A Transformational Approach to Facilitate Monitoring of High-level Policies

Fredrik Seehusen and Ketil Stølen

2008

Contents

1 Introduction 2

2 Step I: Specifying High-level Policies with Sequence Diagrams 4
  2.1 Example – Specifying Policy 1 4
  2.2 Example – Specifying Policy 2 5

3 Step II: Specifying Transformations with Sequence Diagrams 6
  3.1 Example – Specifying a Transformation for Policy 1 7

4 Step III: Transforming Sequence Diagrams to State Machines 7
  4.1 Example – Transforming Policy 2 to a State Machine 9
    4.1.1 Phase 1 9
    4.1.2 Phase 2 12
  4.2 Example – Why the Negation Construct is Useful 14

5 Related Work 14

6 Conclusions 16
  6.0.1 Acknowledgement 16

A UML Sequence Diagrams 18
  A.1 Syntax 18
  A.2 Semantics 19
    A.2.1 LTS of Single Lifeline Diagrams 19
    A.2.2 Trace Semantics of Single Lifeline Diagrams 21
    A.2.3 Synchronous Trace Semantics of (General) Diagrams 23
    A.2.4 Asynchronous Trace Semantics of (General) Diagrams 24
  A.3 Policy Adherence for Sequence Diagrams 25

B State Machines 25
  B.1 Syntax 25
    B.1.1 Basic State Machines 25
    B.1.2 Composite State Machine 26
  B.2 Semantics 27
    B.2.1 Trace Semantics of Basic State Machines 27
    B.2.2 Synchronous Trace Semantics of Composite State Machines 28
Abstract

We present a method for specifying high-level security policies that can be enforced by run-time monitoring mechanisms. The method has three main steps: (1) the user of our method formalizes a set of policy rules using UML sequence diagrams; (2) the user selects a set of transformation rules from a transformation library, and applies these using a tool to automatically obtain a low-level policy (also expressed in UML sequence diagrams); (3) the tool automatically transforms the low-level policy expressed in UML sequence diagrams into a UML state machine that governs the behavior of a run-time policy enforcement mechanism. We believe that the method is both easy to use and useful since it automates much of the policy formalization process.

1 Introduction

Policies are rules governing the choices in the behavior of a system [16]. We consider a particular class of policies, namely security policies that are enforceable by the class of mechanisms that work by monitoring execution steps of some system, herein called the target, and terminating the target’s execution if it is about to violate the security policy being enforced. This class is called EM, for Execution Monitoring [14].

The security policy which is enforced by an EM mechanism is often specified by a state machine that describes exactly those sequences of security relevant actions that the target is allowed to execute. Such EM mechanisms receive an input whenever the target is about to execute a security relevant action. If the state machine of the EM mechanism has an enabled transition on a given input, the current state is updated according to where the transition lands. If the state machine has no enabled transition for a given input, then the target is about to violate the policy being enforced. It is therefore terminated by the EM mechanism.

Security policies are often initially expressed as short natural language statements. Formalizing these statements is, however, time consuming since they often refer to high-level notions such as “opening a connection” or “sending an SMS” which must ultimately be expressed as sequences of security relevant actions of the target. If several policies refer to the same high-level notions, or should be applied to different target platforms, then these must be reformalized for each new policy and each new target platform.

Clearly, it is desirable to have a method that automates as much of the formalization process as possible. In particular, the method should

1. support the formalization of policies at a high level of abstraction;
2. offer automatic generation of low-level policies from high-level policies;
1. INTRODUCTION

3. facilitate automatic enforcement by monitoring of low-level policies;

4. be easy to understand and employ by the users of the method (which we assume are software developers).

The method we present has three main steps which accommodate the above requirements:

Step I The user of our method receives a set of policy rules written in natural language, and formalizes these using UML sequence diagrams.

Step II The user selects a set of transformation rules (expressed in UML sequence diagrams) from a transformation library, and applies these using a tool to automatically obtain a low-level policy (also expressed in UML sequence diagrams).

Step III The tool automatically transforms the low-level policy expressed in UML sequence diagrams into a state machine that governs the behavior of an EM mechanism.

There are two main advantages of using this method as opposed to formalizing low-level policies directly using state machines. First, much of the formalization process is automated due to the transformation from high to low level. This makes the formalization process less time consuming. Second, it will be easier to show that the formalized high-level policy corresponds to the natural language policy it is derived from, than to show this for the low-level policy. The reason for this is that the low-level policy is likely to contain implementation specific details which make the intention of the policy harder to understand.

The choice of UML is motivated by requirement 4. UML is widely used in the software industry. It should therefore be understandable to many software developers which are the intended users of our method. UML sequence diagrams are particularly suitable for policy specification in the sense that they specify partial behavior (as opposed to complete), i.e., the diagrams characterize example runs or snapshots of behavior in a period of time. This is useful when specifying policies since policies are partial statements that often do not talk about all aspects of the target’s behavior. In addition to this, UML sequence diagrams allows for the explicit specification of negative behavior, i.e., behavior that the target is not permitted to engage in. This is useful because the only kind of policies that can be enforced by EM mechanisms are prohibition policies, i.e., policies that state what the target is not allowed to do.

The rest of this report is structured as follows: In Sect. 2 we describe step I of our method by introducing a running example and showing how high-level security policies can be expressed with UML sequence diagrams. Sect. 3 describes step II of our method by showing how a transformation from high-level to low-level policies can be specified using UML sequence diagrams. Step III of our method is described in Sect. 4 which defines a transformation from (low-level) sequence diagram policies to state machine policies that can be enforced by EM mechanisms. Sect. 4 discusses related work, and Sect. 5 provides conclusions and directions of future work. The formal foundation is presented in the appendices. App. A and App. B, present the syntax and the semantics of UML sequence diagrams and state machines, respectively. App. C characterizes the transformation from high- to low-level sequence diagrams. App D defines the transformation from low-level sequence diagrams to state machines.
2 Step I: Specifying High-level Policies with Sequence Diagrams

In the first step of our method, the user receives a set of policy rules written in natural language. The user then formalizes these rules using UML sequence diagrams. In this section we show how to express two security policies. The examples are taken from an industrial case study conducted in the EU project S3MS [15].

As the running example of this report, we consider applications on the Mobile Information Device Profile (MIDP) Java runtime environment for mobile devices. We assume that the runtime environment is associated with an EM mechanism that monitors the executions of applications. Each time an application makes an API-call to the runtime environment, the EM mechanism receives that method call as input. If the current state of the state machines that governs the EM mechanisms has no enabled transitions on that input, then the application is terminated because it has violated the security policy of the EM mechanism.

2.1 Example – Specifying Policy 1

The first policy we consider is

The application is only allowed to establish connections to the address http://s3ms.fast.de.

This policy is specified by the UML sequence diagram of Fig. 1.

Sequence diagrams describe communication between system entities which we will refer to as lifelines. In a diagram, lifelines are represented by vertical dashed lines. An arrow between two lifelines represents a message being sent from one lifeline to the other in the direction of the arrow. Sequence diagrams should be read from top to bottom; a message on a given lifeline should occur before all messages that appear below it on the same lifeline (unless the messages are encapsulated by operators). Communication is asynchronous, thus we distinguish between the occurrence of a message transmission and a message reception. Both kinds of occurrences are viewed as instantaneous and in the following called events.

The two lifelines in Fig. 1 are Application representing the target of the policy, and url representing an arbitrary address that the target can connect to. The
sending of message connect from the Application to url represents an attempt to 
open a connection.

Expressions of the form \{bx\} (where bx is a boolean expression) are called 
constraints. Intuitively, the interaction occurring below the constraint will only 
take place if the constraint evaluates to true.

The constraint of Fig. 1 should evaluate to true if url is not equal to the 
address “http://s3ms.fast.de” (which according to the policy is the only address 
that the application is allowed to establish a connection to).

Interactions that are encapsulated by the neg operator specify negative be-
behavior, i.e., behavior which the target is not allowed to engage in. Thus Fig. 1 
should be read: Application is not allowed to connect to the arbitrary address url 
if url is different from the address “http://s3ms.fast.de”.

UML sequence diagrams are partial in the sense that they typically don’t 
tell the complete story. There are normally other legal and possible behaviors 
that are not considered within the described interaction. In particular, sequence 
diagrams explicitly describe two kinds of behavior: behavior which is positive 
in the sense that it is legal, valid, or desirable, and behavior which is negative 
meaning that it is illegal, invalid, or undesirable. The behavior which is not 
explicitly described by the diagram is called inconclusive meaning that it is 
considered irrelevant for the interaction in question.

We interpret sequence diagrams in terms of positive and negative traces, i.e., 
sequences of events (see App. A). When using sequence diagrams to express 
prohibition policies, we are mainly interested in traces that describe negative 
behavior. If a system is interpreted as a set of traces, then we say that the 
systems adheres to a policy if none of the system’s traces have a negative trace 
of the policy as a sub-trace\(^1\). Thus we take the position that the target is allowed 
to engage in (inconclusive) behavior which is not explicitly described by a given 
policy. This is reasonable since we do not want to use policies to express the 
complete behavior of the target.

Turning back to the example, an application is said to adhere to the policy 
of Fig. 1 if none of its traces contain the transmission of the message connect to 
an address which is different from http://s3ms.fast.de.

### 2.2 Example – Specifying Policy 2

The second natural language policy is

\[ \text{The application is not allowed to send more than } N \text{ SMS messages (where } N \text{ is a natural number).} \]

This policy is specified by the sequence diagram of Fig. 2. Again, the life-
lines of the diagram are Application representing the policy target, and url, this 
time representing an arbitrary recipient address of an SMS message.

Boxes with rounded edges contain assignments of variables to values. In 
Fig. 2, the variable smsSent is initialized to zero, and incremented by one each 
time the application sends an SMS. The loop operator is used to express the 
iteration of the interaction of its operand.

\(^1\)A trace \( s \) is a sub-trace of trace \( t \) if there is a trace \( t' \) obtained from \( t \) by removing zero 
or more events in \( t \) such that \( s = t' \).
The alternation (-operator is used to express alternative interaction scenarios. In Fig. 2, there are two alternatives. The first alternative is applicable if the variable smsSent is less than or equal to N (representing an arbitrary number). In this case the application is allowed to send an SMS, and the variable sendSMS is incremented by one. The second alternative is applicable when smsSent is greater than N. In this case, the application is not allowed to send an SMS as specified by the neg-operator.

An application adheres to the policy of Fig. 2 if none of its traces contain more than N occurrences of the message smsSent.

3 Step II: Specifying Transformations with Sequence Diagrams

In the second step of our method, the user selects a set of transformation rules from a transformation library. The users then employs a tool which automatically applies the transformation rules to the high-level policy such that a low-level policy is produced. In the following, we show how a transformation to the low-level can be defined using UML sequence diagrams. An advantage of using sequence diagrams for this purpose is that the writer of the transformation rules can express the low-level policy behavior using the same language that is used for writing high-level policies. This will also make it easier for the user of our method to understand or modify the transformation rules if that should become necessary.

A transformation rule is specified by a pair of two diagram patterns (diagrams that may contain meta-variables), one left hand side pattern, and one right hand side pattern. When a transformation rule is applied to a diagram $d$, all fragments of $d$ that matches the left hand side pattern of the rule are replaced by the right hand side pattern. Meta-variables are bound according to the matching. A diagram pattern $dp$ matches a diagram $d$ if the meta-variables of $dp$ can be replaced such that the resulting diagram is syntactically equivalent.
to $d$.

In the following we illustrate the use of transformation rules by continuing the example of the previous section.

3.1 Example – Specifying a Transformation for Policy 1

The policies described in the previous section are not enforceable since the behavior of the target is not expressed in terms of API-calls that can be made to the MIDP runtime environment. Recall the policy of Fig. 1. It has a single message `connect` which represents an attempt to open a connection. In order to make the policy enforceable, we need to express this behavior in terms of API-calls that can be made to the runtime environment.

Fig. 3 illustrates a transformation rule which describes how the `connect` message is transformed into the API-calls which can be made in order to establish a connection via the MIDP runtime environment. The diagram on the left in Fig. 3 represents the left hand side rule, and the diagram on the right represents the right hand side rule. In the diagram, all meta-variables are underlined.

Fig. 4 shows the result of applying the rule of Fig. 3 to the policy of Fig. 1. Here we see that the message `connect` has been replaced by the relevant API-calls of the runtime environment.

Clearly, the high-level policy of Fig. 1 allows for an easier comparison to the natural language description than the low-level policy of Fig. 4. Moreover, if the high-level policy of Fig. 1 should be applied to a different runtime environment than MIDP, say .NET, then a similar transformation rule can be written or selected from a transformation library without changing the high-level policy.

Notice that each message of Fig. 4 contains one or more variables that are not explicitly assigned to any value in the diagram. We call these parameter variables. Parameter variables are bound to a value upon the occurrence of the message they are contained in.

4 Step III: Transforming Sequence Diagrams to State Machines

In the third step of our method, the low-level intermediate sequence diagram obtained from step II is automatically transformed into a state machine that governs the behavior of an EM mechanism. The state machine explicitly describes the (positive) behavior which is allowed by a system. Everything which is not described by the state machine is (negative) behavior which is not allowed. Therefore, the state machine does not have a notion of inconclusive behavior as sequence diagrams do.

The semantics of a state machine is a set of traces describing positive behavior. A system adheres to a state machine if each trace described by the system is also described by the state machine (when the trace is restricted to the alphabet of the state machine). We define adherence like this because this notion of adherence is used by existing EM mechanisms (see e.g. [8, 14]) that are governed by state machines.

The transformation of step III converts a sequence diagram into a state machine such that an arbitrary system adheres to the state machine if and only if
the system adheres to the sequence diagram, i.e., the transformation is adherence preserving. As one would expect, the transformation converts positive and negative behavior of the sequence diagram into positive and negative behavior of the state machine, respectively. However, the inconclusive behavior of the sequence diagram is converted to positive behavior of the state machine. This is because, by definition of adheres for sequence diagrams, a system is allowed to engage in the (inconclusive) behavior which is not described by the sequence
The only kind of policies that can be enforced by EM mechanisms are so-called prohibition policies, i.e., policies that describe what a system is not allowed to do. Sequence diagrams are suitable for specifying these kinds of policies because they have a construct for specifying explicit negative behavior. For this reason, the sequence diagram policies are often more readable than the corresponding state machine policies since the state machine policies can only explicitly describe behavior which is allowed by an application. In the following examples, we clarify this point and explain how the transformation from sequence diagrams to state machines works.

4.1 Example – Transforming Policy 2 to a State Machine

Although our method is intended to transform low-level intermediate sequence diagram policies to state machines, we will use the sequence diagram of Fig. 2 to illustrate the transformation process as this diagram better highlights the transformation phases than the low-level intermediate policy of Fig. 4.

In general, the transformation from a sequence diagram to a state machine yields a composite state machine (i.e., a set of basic state machines) that contains one basic state machine for each lifeline of the diagram. Therefore, the transformation from the sequence diagram of Fig. 2 yields a composite state machine consisting of one basic state machine describing the lifeline Application and one basic state machine describing the lifeline url.

The transformation from a sequence diagram $d$ with one lifeline to a (basic) state machine has two phases. In phase 1, the sequence diagram $d$ is transformed into a state machine $P$ whose trace semantics equals the negative trace set of $d$. In phase 2, $P$ is inverted into the state machine $P'$ whose semantics is the set of all traces that do not have a trace of $P$ as a sub-trace.

In the following we explain the two phases by transforming a diagram describing lifeline Application in Fig. 2 into a basic state machine.

4.1.1 Phase 1

First, in phase 1, we transform the sequence diagram describing the lifeline Application into a state machine whose trace semantics equals the negative traces of the diagram. To achieve this, we make use of the operational semantics of sequence diagrams which is inspired by [11] and described in App. A.

The operational semantics makes use of a labeled transition system (LTS) whose states are diagrams, and whose transitions are labeled by events, assignments, and so-called silent events that indicate which kind of operation has been
executed. If the labeled transition system has a transition from a diagram \(d\) to a diagram \(d'\) that is labeled by, say, event \(e\), then we understand that event \(e\) is enabled in diagram \(d\), and that \(d'\) is obtained by removing \(e\) from \(d\).

To transform a diagram describing Application into a state machine describing its negative traces, we first construct the LTS whose states are exactly those that can be reached from diagram. The result is illustrated in Fig.5. Then, we transform the LTS into a state machine describing the negative traces of the diagram. The states of this state machine are of the form \((q, mo)\) where \(q\) is a state in the LTS and \(mo\) is a mode which used to differentiate between positive and negative traces. There are two kinds of modes: \(pos\) (for positive) or \(neg\) (for negative). A state with mode \(neg\) leads to an accepting state that accepts negative traces. Thus we require that all accepting states have mode \(neg\).

Transitions of state machines are labeled by action expressions of the form \(nm.si[\{bx\}]/ef\), where \(nm.si\) is called an input event, \([bx]\) (where \(bx\) is a boolean expression) is called a guard, and \(ef\) is called an effect. An effect may either be an assignment or an output event.

Since the transitions of state machines are cannot be labeled by silent events, such transitions are skipped when converting the LTS into a state machine. To make this more precise, we define the following notation.

- \(q \Rightarrow q'\) means that the LTS has a sequence of zero or more transitions that are labeled by silent events from state \(q\) to state \(q'\), i.e., \(q \Rightarrow \tau_1 \cdots \Rightarrow \tau_n \Rightarrow q'\).

When an action expression is executed, the parameter values (if any) of the events, assignments, and guards are all bound to the same values. Therefore, when constructing the transitions of the state machine from the LTS, constraints preceding assignments or events are converted into guards for the events or assignment succeeding it. For instance, if the LTS has transitions \(q \Rightarrow q_1 \Rightarrow q_2 \Rightarrow q'\) (where \(\{bx\} and \{bx'\}\) are constraints, and \(a\) is an assignment), then the state machine has a transition labeled by \([bx'']/a\) (where \(bx''\) is the conjunction of \(bx\) and \(bx'\)). To make this more precise, we define the following notation.

- Let \(bx\) be the conjunction of the boolean expressions \(bx_1, \ldots, bx_n\) and \(ae\) be an event or an assignment, then \(q \Rightarrow (bx,ae) q'\) means \(q \Rightarrow (bx_1) q_1 \cdots \Rightarrow (bx_n) q_n \Rightarrow ae \Rightarrow q'\).

We make use of the silent event \(\tau_{neg}\) to distinguish between negative and positive executions. That is, we distinguish between those states that can be reached from transitions labeled by \(\tau_{neg}\) and those states that cannot be reached from such transitions. A state that can be reached from a transition labeled by \(\tau_{neg}\) leads to a state that accepts a negative trace.

The transitions of the state machine obtained from the LTS are constructed by the following rules:

- if the LTS has a transition \(q \Rightarrow (bx,ae) q'\) and none of the transitions from \(q\) to \(q'\) are labeled by \(\tau_{neg}\), then the state machine has a transition \((q, mo) \Rightarrow (bx,ae) (q', mo)\);
- if the LTS has a transition \(q \Rightarrow (bx,ae) q'\) and one of the transitions from \(q\) to
STEP III: TRANSFORMING SEQUENCE DIAGRAMS TO STATE MACHINES

Figure 6: State machine describing negative traces for Application

$q'$ are labeled by $\tau_{neg}$, then the state machine has a transition $(q, ma) \xrightarrow{(bx, ac)} (q', neg)$.

From these rules, we obtain the state machine shown in Fig. 6 from the LTS of Fig. 5.

We assume that transitions that are labeled by action expressions without input or output events are always taken immediately by the state machine. Thus, if a state has one of these “immediate” transitions, it cannot have an additional outgoing transition with input or output events.

The alphabet of a state machine is a set of events. When we transform a sequence diagram to a state machine, the alphabet of the state machine is the set of all events that occur in the sequence diagram.

The graphical notation used for specifying state machines is inspired by the UML statechart diagram notion. See App. B for more details.

Turning back to the example, recall that our goal is to transform the lifeline Application in the sequence diagram of Fig. 2 into a state machine describing all traces that do not have a negative trace of the sequence diagram as a sub-trace. This means that if the sequence diagram describes two negative traces $s$ and $t$ such that $s$ is a prefix of $t$, then we can ignore trace $t$ because every trace that has $t$ as a sub-trace will also have $s$ as a sub-trace. As a result of this, we can remove all states occurring after the accepting state of Fig. 6. We then obtain the state machine of Fig. 7. Notice that we have renamed the states. This is just a matter of convenience since the state names are only used for the purpose of illustration.
4.1.2 Phase 2
In phase 2, we “invert” the state machine of Fig. 7 into the state machine whose semantics is the set of all traces that do not have a trace of the state machine of Fig. 7 as a sub-trace. In general, the inversion $P'$ of a state machine $P$ has the power-set of the states in $P$ as its states\(^2\) (see e.g., Fig. ??), the alphabet of $P$ as its alphabet, all its states as its accepting states, and its transitions are those that are constructed by the following rules

- (a) if $P'$ has a state $Q \cup \{q\}$, and $P$ has a transition from $q$ to $q'$ that has not been previously visited to reach $Q \cup \{q\}$ and whose action expression $\text{act}$ does not contain an event, then $P'$ has a transition from $Q \cup \{q\}$ to $Q \cup \{q'\}$ that is labeled by $\text{act}$;

- (b) if $P'$ has a state $Q \cup \{q\}$ and $P$ has a transition from $q$ to $q'$ (where $q'$ is not an accepting state) that has not been previously visited to reach $Q \cup \{q\}$ and whose action expression $\text{act}$ contains an event, then $P$ has a transition from $Q \cup \{q\}$ to $Q \cup \{q\} \cup \{q'\}$ that is labeled by $\text{act}$;

- (c) if $Q$ is a state in $P'$ such that no states in $Q$ has a transition in $P$ whose action expression does not contain an event, $e$ is in the alphabet of $P$, and $\text{act}$ is an action expression containing $e$ that is not enabled in any state $q$ in $Q$, then $P'$ has a transition from $Q$ to $Q$ that is labeled by $\text{act}$.

Rules (a) and (b) ensure that $P'$ describes the positive traces, but not the negative traces, of the sequence diagram that $P$ was transformed from. Rule (c) ensures that $P'$ describes the inconclusive behavior of the sequence diagram that $P$ was transformed from.

Notice that rule (a) and (b) “collect” states of $P$ that has been previously visited, unless those states have outgoing transitions whose action expressions do not contain events. The reason for this is best illustrated with an example of principle. Consider the state machine $P$ on the left hand side of Fig. 8. The set of traces described by it is $\{(a, a), (b, b)\}$. The inversion of $P$ is the state machine $P'$ shown on the right hand side of Fig. 8. Since all states of $P'$ are understood to be accepting, we have omitted to specify the accepting state in the figure. The trace semantics of $P'$ is the set $\{(), (a), (b), (a, b), (b, a)\}$. Initially, both $a$ and $b$ are enabled. However, if $a$ occurs, then only $b$ is enabled (if we

\(^2\)Each state of $P'$ also needs to keep track of the transition already visited in $P$ to reach that state. We postpone the discussion of this technical detail to App. D
Figure 9: State machine describing positive and inconclusive traces for Application

assume that the alphabet of the state machine is \( \{a, b\} \). Similarly if \( b \) occurs, then only \( a \) is enabled. If both \( a \) and \( b \) occur, then no events are enabled. The accepting states of \( P \) are used to find those events that should not be enabled in \( P' \). Therefore, enabled events in previously visited states must always progress towards accepting states in \( P \). This is the reason why previously visited states are collected. For instance, consider the transition \( \{2\} \xrightarrow{a} \{2, 4\} \) in \( P' \). Here the state 2 is “collected” because we need to make sure that \( b \) is not enabled after \( b \) has occurred in state \( \{2, 4\} \). Since 2 is collected, the occurrence of \( b \) in state \( \{2, 4\} \) leads to state \( \{2, 3, 4\} \) and \( b \) is not enabled in this state because the occurrence of \( b \) in state 3 leads to an accepting state in \( P' \).

Because we assume that states cannot both have outgoing transitions with events and transitions without events, there is no need to collect states whose outgoing transitions are not labeled by events. This is the reason why the state 1 is not collected in state machine \( P' \).

Transitions of the form \( e[false] \) in the state machine \( P' \) of Fig. 8 are all constructed by rule (c) above. These transitions are never enabled because the guard is \( false \). These transitions could have been removed from \( P' \) without affecting the semantics of the state machine. Hence, rule (c) does not play an essential role in the example. An example of in which rule (c) plays an essential role is given in the next section (Sect. 4.2).

Returning back to our running example, Fig. 9 shows the inversion of the state machine of Fig. 7. Since all state in the state machine of Fig. 9 are assumed to be accepting, we have omitted to specify the accepting states. Also, we have omitted to specify transitions whose guards always evaluate to false.

We have that every trace that contain less than or equal to \( N \) occurrences of the sendSMS message are accepted by the state machine of Fig. 9. Indeed, this was the intended meaning of the policy.

The procedure described above can be used to transform the lifeline url of Fig. 2 into a basic state machine whose name is url. The result of transforming the diagram of Fig. 2 into a state machine is then the composite state machine consisting of Application (of Fig. 9) and the basic state machine describing the lifeline url.

The semantics of a composite state machine is obtained by parallel composing the semantics of its basic state machines. In App. B, we define both a notion of synchronous and asynchronous parallel composition of basic state machines. Since the messages of the running example are understood to be method calls, it is appropriate to use the synchronous interpretation.
4.2 Example – Why the Negation Construct is Useful

As noted in the beginning of this section, the sequence diagram construct for specifying explicit negative behavior is useful when specifying policies that can be enforced by EM mechanisms. In this section, we illustrate this with an example.

Consider the policy shown in Fig. 10. It may be seen as a composition of three policies. The upper most policy states that after the lifeline A has transmitted a, it is not allowed to transmit b. The two other policies are similar except that the messages are different.

To transform a diagram describing the lifeline A into a basic state machine, we first (in phase 1) construct the basic state machine that describes the negative traces of the diagram. The resulting state machine is shown in Fig. 11. Then, we invert the state machine of Fig. 11 to obtain the state machine of Fig. 12. Note that we have omitted to specify the transitions whose guards always evaluate to false. Note also that all reflexive transitions correspond to inconclusive behavior of the sequence diagram of Fig. 10. These transitions are all generated by rule (c) in the procedure for inverting a state machine described in Sect. 4.1.

Clearly, it is more difficult to understand the meaning of the state machine policy of Fig. 12 than the sequence diagram policy of Fig. 10. The reason for this is that the state machine policy have to describe all behavior which is allowed. However, the process of inverting a state machine may lead to a state explosion. This shows why it is useful to have a construct for specifying negative behavior.

5 Related Work

Previous work that address the transformation of policies or security requirements are [1, 2, 3, 4, 6, 7, 12, 10, 13, 18]. All these differ clearly from ours in that the policy specifications, transformations, and enforcement mechanisms are different from the ones considered in this report.
5 RELATED WORK

Figure 11: Intermediate state machine for Application

Figure 12: Intermediate state machine for Application

[1] considers authorization policies, [13] considers security policies for Web Services, [12] considers policies in the form of logical conditions, [1, 2, 3, 6, 7, 10, 18] consider access-control requirements, and [4] focuses on transformation techniques rather than any particular kind policies. Of the citations above, [3] gives the most comprehensive account of policy transformation. In particular it shows how platform independent role based access control requirements can be expressed in UML diagrams, and how these requirements can be transformed to platform specific access control requirements.

The transformation of sequence diagrams (or a similar language) to state machines has been previously addressed in [5, 9, 19]. However, these do not consider policies, nor do they offer a way of changing the granularity of interactions during transformation.

The only paper that we are aware of that considers UML sequence diagrams for policy specification is [17]. However, in that paper, transformations from high- to low-level policies or transformation to state machines is not considered. The paper argues that sequence diagrams must be extended with customized expressions for deontic modalities to support policy specification. While this is true in general, this is not needed for the kind of prohibition policies that can be enforced by EM mechanisms.
6 Conclusions

We claim that it is desirable to automate as much as possible of the process of formalizing security policies. To this end we have presented a method which (1) supports the formalization of policies at a high level of abstraction, (2) offers automatic generation of low-level policies from high-level policies, and (3) facilitates automatic enforcement by monitoring of low-level policies. Enforcement mechanisms for the kind of policies considered in this report have been developed in S³MS EU project[15]. Thus the method fulfills the first three requirements that were presented in the Sect. 1. Empirical investigation of whether the method satisfies the fourth requirement, namely that it should be easy to understand by software developers, is beyond the scope of this report. However, we have used UML as a policy language, and using UML for specifying policies, we believe, is not much harder than using UML to specify software systems (in particular, since we focus on execution monitoring and do not have to take other modalities than prohibition into consideration).

Previous work in the literature has addressed policy transformation, but differ clearly from ours in that the policy specifications, transformations, and enforcement mechanisms different.

6.0.1 Acknowledgement

Partially funded by the IST FP6 programme of the EC, under the IST-FP6-STREP-27004 S3MS project [15].

References


REFERENCES


A  UML SEQUENCE DIAGRAMS


A  UML Sequence Diagrams

In this section, we first present the syntax (Sect. A.1) and semantics of UML sequence diagrams (Sect. A.2). Then, in Sect. A.3, we define what it means for a system to adhere to a sequence diagram policy.

A.1 Syntax

We use the following syntactic categories to define the textual representation of sequence diagrams:

- $ax \in AExp$ arithmetic expressions
- $bx \in BExp$ boolean expressions
- $sx \in SExp$ string expressions

We let $Exp$ denote the set of all arithmetic, boolean, and string expressions, and we let $ex$ range over this set. We denote the empty expression by $\epsilon$. We let $Val$ denote the set of all values, i.e., numerals, strings, and booleans ($t$ or $f$).

**Definition 1 (Sequence diagram expression)** Let $e$, $bx$, $l$, $x$, and $ex$ denote events, boolean expressions, lifelines, variables, and expressions. The set of all syntactically correct sequence diagram expressions $D$ is defined by the following grammar:

$$d ::= \text{skip} \mid e \mid \text{constr}(bx, l) \mid \text{assign}(x, ex, l) \mid \text{refuse}(d) \mid \text{loop}(0..*)(d) \mid d_1 \text{seq} d_2 \mid d_1 \text{alt} d_2 \mid d_1 \text{par} d_2$$

The base cases imply that any event ($e$), constraint ($\text{constr}(bx, l)$), or assignment ($\text{assign}(x, ex, l)$) is a sequence diagram. Any other sequence diagram is constructed from the basic ones through the application of operators for negation ($\text{refuse}(d)$), iteration ($\text{loop}(0..*)(d)$), weak sequencing ($d_1 \text{seq} d_2$), choice ($d_1 \text{alt} d_2$), and parallel execution ($d_1 \text{par} d_2$).

An event is a pair $(k, m)$ of a kind $k$ and a message $m$. An event of the form $(!, m)$ represents a transmission of message $m$, whereas an event of the form $(?, m)$ represents a reception of $m$. We let $E$ denote the set of all events:

$$e \in E \overset{def}{=} \{!, ?\} \times M$$

Messages are of the form $(l_t, l_r, si)$ where $l_t$ represents the transmitter lifeline of the message, $l_r$ represents the receiver lifeline of the message, and $si$ represents the signal of the message. We let $L$ denote the set of all lifelines, and $SI$ denote the set of all signals. The set $M$ of all messages is then defined by

$$m \in M \overset{def}{=} L \times L \times SI$$

A signal is a tuple $(nm, ex_1, \ldots, ex_n)$ where $nm$ denotes the signal name, and $ex_1, \ldots, ex_n$ are the parameters of the signal. We usually write $nm(ex_1, ex_2, \ldots, ex_n)$ instead of $(nm, ex_1, ex_2, \ldots, ex_n)$. Formally, the set of all signals is defined

$$SI \overset{def}{=} Nm \times Exp \times \cdots \times Exp$$
Signals may contain special so-called parameter variables that are bound to a value upon the occurrence of the signal. We let $px \in \text{PVar}$ denote the set of all parameter values. We assume $\text{PVar} \subseteq \text{Var}$.

The expression $\text{assign}(x, ex, l)$ represents the assignment of expression $ex$ to variable $x$ on lifeline $l$. The expression $\text{constr}(bx, l)$ represents a constraint on lifeline $l$. We let $a \in \text{A}$ and $c \in \text{C}$ denote the set of all assignments and constraints, respectively.

We say that an output event $e$ is on lifeline $l$ if the transmitter of the message of $e$ is $l$. Similarly, an input event $e$ is on lifeline $l$ if the receiver of its message is $l$. Assignments and constraints are on lifeline $l$ if the lifeline of the assignment or the constraint is $l$. We define this formally be the predicate $\text{onl} \in (\text{E} \cup \text{A} \cup \text{C}) \times \text{L} \rightarrow \text{BExp}$:

$$
\text{onl}((!, (l_t, l_r, m)), l_t) \equiv t
\text{onl}((?!, (l_t, l_r, m)), l_r) \equiv t
\text{onl}(\text{assign}(x, ex, l), l) \equiv t
\text{onl}(\text{constr}(bx, l), l) \equiv t
\text{onl}(eca) \equiv f
$$

where $eca$ is a member of $\text{E} \cup \text{A} \cup \text{C}$. Note that each rule has precedence over the rules below it.

By $\text{E}'$, $\text{A}'$, and $\text{C}'$, we denote the set of all events, assignments, and constraints on lifeline $l$, respectively. A diagram $d$ is on lifeline $l$ iff all its events, assignments, and constraints are on $l$. We let $\text{D}'$ denote the set of all diagrams on lifeline $l$.

### A.2 Semantics

In this section, we define the semantics of UML sequence diagrams. First, in Sect. A.2.1, we define the so-called labeled transition system of a sequence diagram with a single lifeline. Then, in Sect. A.2.2, we define the trace semantics of single lifeline diagrams. Finally, in Sect. A.2.3 and Sect. A.2.3 we define the synchronous and asynchronous trace semantics of sequence diagrams with one or more lifelines.

The definition of the semantics if similar to the operational semantics of UML sequence diagrams given in [11]. However, there are two essential differences. First, the operational semantics of [11] is defined for UML sequence diagrams that are extended with a construct for specifying so-called explicit non-deterministic choices. We do not consider explicit non-deterministic choices in this report. Therefore, our semantics has been simplified accordingly. Second, we allow messages of sequence diagrams to contain parameter variables that are bound to a value upon the occurrence of the message they are contained in. The reason for this is that most of the example policies we have tried to specify with sequence diagrams would not have been possible to specify without this expressibility. The sequence diagrams considered in [11] do not parameter variables. Therefore, our semantics differs from the one considered in [11].

#### A.2.1 LTS of Single Lifeline Diagrams

In this section, we define the labeled transition system (LTS) that is induced by sequence diagrams on a specific lifeline.
Definition 2 (Labeled transition system (LTS)) A labeled transition system over the set of labels \( LE \) is a pair \( (Q, R) \) consisting of

- a set \( Q \) of states;
- a ternary relation of \( R \subseteq (Q \times LE \times Q) \), known as a transition relation.

We usually write \( q \xrightarrow{le} q' \in (Q, R) \) if \( (q, le, q') \in R \), or just \( q \xrightarrow{le} q' \) if \( (Q, R) \) is clear from the context.

We use a notion of structural congruence in order to define the LTS that is induced by sequence diagrams.

Definition 3 (Structural congruence) Structural congruence over sequence diagrams, written \( \equiv \), is the congruence over \( D \) determined by the following equations:

1. \( d_\text{seq} \equiv d, \text{skip}_\text{seq} \equiv d \)
2. \( d_\text{par} \equiv d, \text{skip}_\text{par} \equiv d \)
3. \( \text{skip} \equiv \text{skip} \)
4. \( \text{loop}(0..*) (\text{skip}) \equiv \text{skip} \)

The LTS of a single lifeline diagram \( d \) is obtained by executing \( d \) without evaluating its constraints and assignments. Thus, the LTS has diagrams as its states, and its transitions are labeled by events, assignments, and constraints. Transitions may also be labeled by so-called silent events that indicate the kind of construct that has been executed. We let \( T \) denote the set of all silent events:

\[
\tau \in T \overset{\text{def}}{=} \{ \tau_\text{alt}, \tau_\text{loop}, \tau_\text{refuse} \}
\]

Definition 4 (LTS of single lifeline diagrams) The LTS of sequence diagrams on lifeline \( l \), written \( \text{LTS}(l) \), has \( D' \) as its states, \( A' \cup C' \cup E' \cup T \) (we let \( \alpha \) range over this set) as its labels, and its transitions are exactly those that are generated by the following rules

\[
\begin{align*}
\text{refuse}(d) & \xrightarrow{\tau_\text{refuse}} d \\
\text{loop}(0..*) (d) & \xrightarrow{\tau_\text{loop}} \text{skip alt} (d \text{seq loop}(0..*) (d)) \\
d_1 \text{alt} d_2 & \overset{\tau_\text{alt}}{\xrightarrow{\alpha}} d_1 \\
d_1 \text{alt} d_2 & \overset{\tau_\text{alt}}{\xrightarrow{\alpha}} d_2 \\
d_1 \text{seq} d_2 & \overset{\alpha}{\xrightarrow{\text{seq}}} d_1' \text{seq} d_2 \\
d_1 \text{par} d_2 & \overset{\alpha}{\xrightarrow{\text{par}}} d_1' \text{par} d_2 \\
d_1 & \overset{\alpha}{\xrightarrow{\text{seq}}} d_1' \\
d_2 & \overset{\alpha}{\xrightarrow{\text{par}}} d_2' \\
d_1 & \overset{\alpha}{\xrightarrow{\text{par}}} d_1' \\
d_2 & \overset{\alpha}{\xrightarrow{\text{par}}} d_1' \\
\end{align*}
\]
A.2.2 Trace Semantics of Single Lifeline Diagrams

In this section, we define the trace semantics of single lifeline diagrams. We first define the execution graph of sequence diagrams that is obtained by executing diagrams and evaluating constraints and assignments. Then, we define the trace semantics in terms of the execution graph.

Some auxiliary functions are needed to define the execution graph. Let \( \sigma \in \text{Exp} \rightarrow \mathcal{P}(\text{Var}) \) be a function that returns the variables of an expression. We lift the function to signals such that \( \sigma(si) \) yields all variables occurring in the parameters of signal \( si \). Furthermore, we let \( \sigma(e) \) yield all variables occurring in the signal of event \( e \). Similarly, \( \text{pvar} \in \text{Exp} \rightarrow \mathcal{P}(\text{PVar}) \) yields all parameter variables in an expression. The function \( \text{pvar} \) is lifted to signals and events in the same way as the function \( \sigma \).

An expression \( ex \in \text{Exp} \) is closed if \( \sigma(ex) = \emptyset \). We let \( \text{CExp} \) denote the set of closed expressions, defined as:

\[
\text{CExp} \overset{\text{def}}{=} \{ ex \in \text{Exp} \mid \sigma(ex) = \emptyset \}
\]

We assume the existence of a function \( \text{eval} \in \text{CExp} \rightarrow \text{Val} \cup \{ \bot \} \) that evaluates any closed expression to a value. If an expression \( ex \) is not well-formed, then \( \text{eval}(ex) = \bot \). The evaluation function is lifted to signals such that \( \text{eval}(si) \) yields the signal obtained from \( si \) by evaluating all expressions in \( si \). If one expression \( ex \) in \( si \) is not well-formed, i.e., \( \text{eval}(ex) = \bot \), then \( \text{eval}(si) = \bot \). The evaluation function is similarly lifted to events.

For simplicity, we assume that all expressions that occur in a sequence diagram are well-formed.

Let \( \sigma \in \text{Var} \rightarrow \text{Exp} \) be a (partial) mapping from variables to expressions. We denote such a mapping \( \sigma = \{ x_1 \mapsto ex_1, x_2 \mapsto ex_2, \ldots, x_n \mapsto ex_n \} \) for distinct \( x_1, x_2, \ldots, x_n \in \text{Var} \) and \( ex_1, ex_2, \ldots, ex_n \in \text{Exp} \). If \( ex_1, ex_2, \ldots, ex_n \in \text{Val} \) we call it a data state. We let \( \Sigma \) denote the set of all mappings and \( \hat{\Sigma} \) the set of all data states.

The empty mapping is denoted by \( \emptyset \). \( \text{Dom}(\sigma) \) denotes the domain of \( \sigma \). \( \sigma[x \mapsto ex] \) is the mapping \( \sigma \) except that it maps \( x \) to \( ex \), i.e.:

\[
\{ x_1 \mapsto ex_1, \ldots, x_n \mapsto ex_n \}[x \mapsto ex] \overset{\text{def}}{=} \begin{cases} 
\{ x_1 \mapsto ex_1, \ldots, x_n \mapsto ex_n, x \mapsto ex \} & \text{if } x \neq x_i \text{ for all } i \in \{1, \ldots, n\} \\
\{ x_1 \mapsto ex_1, \ldots, x_i \mapsto ex_i, \ldots, x_n \mapsto ex_n \} & \text{if } x = x_i \text{ for some } i \in \{1, \ldots, n\}
\end{cases}
\]

We generalize \( \sigma[x \mapsto ex] \) to \( \sigma[\sigma'] \) in the following way:

\[
\sigma[\{ x_1 \mapsto ex_1, \ldots, x_n \mapsto ex_n \}] \overset{\text{def}}{=} \sigma[\{ x_1 \mapsto ex_1 \} \cdots [x_n \mapsto ex_n]
\]

\( \sigma(x) \) returns the expression that \( x \) maps to, and is only defined for \( x \in \text{Dom}(\sigma) \). The mapping is lifted to expressions such that \( \sigma(ex) \) yields the expression obtained from \( ex \) by simultaneously substituting the variables of \( ex \) with the expressions that these variables map to in \( \sigma \). We furthermore lift \( \sigma \) to events such that \( \sigma(e) \) yields the event obtained from \( e \) by substituting all expressions in \( e \) according to \( \sigma \).

The execution graph of a sequence diagram on lifeline \( l \) is defined in terms of the LTS \( \text{LTS}(l) \). Most of the silent events of \( \text{LTS}(l) \) are not needed, and
are therefore removed when constructing the execution graph. The exception is the silent event \( \tau_{\text{refuse}} \), which is used to separate between positive and negative executions. To make this precise, we define the notion of positive and negative experiment relations.

**Definition 5 (Experiment relations)** The positive relations \( \Rightarrow_p \) and \( \xrightarrow{\text{assign}(\text{expr}, \text{val})} \), the negative relations \( \Rightarrow_n \) and \( \xrightarrow{\text{assign}(\text{expr}, \text{val})} \), and the relation \( \Rightarrow \) for sequence \( s \), are defined as follows

1. \( q \Rightarrow_p q' \) means that there is a sequence of zero or more transitions \( q \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_n} q' \) such that \( \tau_i \neq \tau_{\text{refuse}} \) for all \( i \in \{1, \ldots, n\} \);
2. \( q \Rightarrow_n q' \) means that there is a sequence of one or more transitions \( q \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_n} q' \) such that \( \tau_i = \tau_{\text{refuse}} \) for some \( i \in \{1, \ldots, n\} \);
3. Let \( bx \) be the conjunction of the boolean expressions \( bx_1, \ldots, bx_n \) and \( ae \) be an event or an assignment, then \( q \xrightarrow{\text{assign}(bx_1 \land \cdots \land bx_n, l)} q_1 \cdots \xrightarrow{\text{assign}(bx_n, l)} q_n \Rightarrow_p q' \); and the relation \( \Rightarrow \) for sequence \( s \) is

\[
A \cup \{ \text{assign}(\text{expr}, \text{val}) \} \}
\]

A node of the execution graph is a triple \((d, \sigma, m_0)\) consisting of a diagram \( d \), a data state \( \sigma \), and a mode \( m_0 \). The mode \( m_0 \) is used to separate between positive and negative executions. The data state \( \sigma \) holds the variables and the values that have been assigned to them during execution. A variable \( x \) may be bound to a value in two ways: either \( x \) is explicitly bound to a value when an assign construct is executed, or \( x \) is bound to an arbitrary value upon the occurrence of an event whose signal contains \( x \) as parameter variable.

For simplicity, we assume that all diagrams are well-formed in the sense that all expressions that are evaluated under the execution of a diagram are closed, i.e., that the diagram does not contain any free variables that are not assigned to a value.

**Definition 6 (Execution graph of single lifeline diagrams)** The execution graph of sequence diagrams on lifeline \( l \), written \( EG(l) \), is the LTS that has \( D \times X \times \{ \text{pos, neg} \} \) as its states, \( E \cup \{ \tau_{\text{assign}} \} \) as its labels, and its transitions are exactly those that are generated by the following rules:

\[
\begin{align*}
\dfrac{d \xrightarrow{\text{assign}([x, \text{val}])} d' \in LTS(l)}{\Rightarrow} & \quad \text{if } \text{eval}(\sigma(bx)) = t & \\
\dfrac{[d, \sigma, m_0] \xrightarrow{\tau_{\text{assign}}} [d', \sigma[x \mapsto \text{eval}(\sigma(ex))], m_0]}{\Rightarrow} & \quad \text{if } \text{eval}(\sigma(bx)) = t & \\
\dfrac{d \xrightarrow{\text{assign}([x, \text{val}])} d' \in LTS(l)}{[d, \sigma, m_0] \xrightarrow{\tau_{\text{assign}}} [d', \sigma[x \mapsto \text{eval}(\sigma(ex))], \text{neg}]} & \quad \text{if } \text{eval}(\sigma(bx)) = t & \\
\end{align*}
\]
The trace semantics of a sequence diagram is defined by

\[
\frac{d (bx, e) \rightarrow_p d'}{[d, \sigma, mo] \xrightarrow{eval(\sigma'[e])} [d', \sigma[\sigma'], mo]} \quad \text{if } cond(\sigma, \sigma', bx, e)
\]

\[
\frac{d (bx, e) \rightarrow_n d'}{[d, \sigma, mo] \xrightarrow{eval(\sigma'[e])} [d', \sigma[\sigma'], neg]} \quad \text{if } cond(\sigma, \sigma', bx, e)
\]

where the predicate \(cond\) is defined by

\[
cond(\sigma, \sigma', bx, e) \iff eval(\sigma'[e]) = t \land \text{Dom}(\sigma') = p\text{var}(e) \land eval(\sigma'[e]) \neq \bot
\]

The trace semantics of a sequence diagram \(d\) is a pair \((H_p, H_n)\) where \(H_p\) is a set of traces describing the positive executions of \(d\), and \(H_n\) is a set of traces describing the negative executions of \(d\).

**Definition 7 (Trace semantics of single lifeline diagrams)** The trace semantics of a sequence diagram \(d\) on lifeline \(l\), written \([d]\), is defined by

\[
[d] \overset{\text{def}}{=} \{(s \in E^* | [d, 0, pos] \Rightarrow [skip, \sigma, pos] \in EG(l)), \{s \in E^* | [d, 0, pos] \Rightarrow [skip, \sigma, neg] \in EG(l)]\}
\]

### A.2.3 Synchronous Trace Semantics of (General) Diagrams

In this section, we define the synchronous semantics of sequence diagrams in general, i.e., diagrams with more than one lifeline. We assume that all diagrams are on standard form.

**Definition 8 (Standard form)** A diagram \(d\) is on standard form iff

\[
d = d_1 \, \text{pard}_2 \, \text{par} \cdots \text{pard}_n, \quad \text{var}(d_1) \cap \cdots \cap \text{var}(d_n) = \emptyset
\]

where \(d_1 \in D^{l_1}, \ldots, d_2 \in D^{l_2}, \ldots, d_n \in D^{l_n}\) for distinct lifelines \(l_1, l_2, \ldots, l_n \in L\).

We do not lose generality by requiring that diagrams be on standard form. All implementable sequence diagrams \(d\) can be brought to the standard form by parallel composing each projection of \(d\) to a lifeline in \(d\).

As we did in the previous section, we first define the execution graph, then we define the trace semantics in terms of the execution graph.

**Definition 9 (Synchronous execution graph of sequence diagrams)** The synchronous execution graph of (general) sequence diagrams, written \(EGS\), is the LTS that has \(D \times \Sigma \times \{\text{pos}, \text{neg}\}\) as its states, \(M \cup \{\tau_{\text{assign}}\}\) as its labels, and its transitions are exactly those that can be derived from the following rules:

\[
\begin{align*}
[d_1, \sigma, mo] \xrightarrow{(l,m)} & [d_2', \sigma', mo'] \in EG(l_2) \\
[d_1 \, \text{par} \cdots \text{pard}_i \, \text{par} \cdots \text{pard}_n, \sigma, mo] \xrightarrow{\tau_i} & [d_1 \, \text{par} \cdots \text{pard}_i \, \text{par} \cdots \text{pard}_n, \sigma', mo'] \\
[d_1, \sigma, mo] \xrightarrow{(l,m)} & [d_2', \sigma_j, mo_j] \in EG(l_2) \\
[d, \sigma, mo] \xrightarrow{m} & [d', \sigma[\sigma_j]], mo']
\end{align*}
\]

with

\[
\begin{align*}
\text{d} & = d_1 \, \text{par} \cdots \text{pard}_i \, \text{par} \cdots \text{pard}_j \cdots \text{pard}_n \\
\text{d'} & = d_1 \, \text{par} \cdots \text{pard}_i \, \text{par} \cdots \text{pard}_j \cdots \text{pard}_n \\
\text{mo'} & = \text{neg} \quad \text{if } mo_i = \text{neg} \text{ or } mo_j = \text{neg} \\
\text{mo'} & = \text{pos} \quad \text{if } mo_i = \text{pos} \text{ and } mo_j = \text{pos}
\end{align*}
\]
The synchronous trace semantics of a sequence diagram is defined in terms of sequences of messages since the distinction of message transmission and message reception is not need for synchronous communication.

**Definition 10 (Synchronous trace semantics of sequence diagrams)** The synchronous trace semantics of a (general) sequence diagram, written $[d]$, is defined by

\[
[d] = (\{s \in M^* \mid [d, \emptyset, \text{pos}] \xrightarrow{\text{add}} [d', \sigma, \text{pos}] \in EGS \land d' \equiv \text{skip}, \\
\{s \in M^* \mid [d, \emptyset, \text{pos}] \xrightarrow{\text{rm}} [d', \sigma, \text{neg}] \in EGS \land d' \equiv \text{skip}\})
\]

### A.2.4 Asynchronous Trace Semantics of (General) Diagrams

To define the asynchronous trace semantics of (general) sequence diagrams, we assume that there is some communication medium $B$ which is used by the lifelines of a diagram in order to send messages to each other. We do not assume a predefined structure of the communication medium. The only requirement is that the following functions are defined

- $\text{add} \in B \times M \to B$: Adds a message to the communication medium.
- $\text{rm} \in B \times M \to B$: Removes a message from the communication medium.
- $\text{ready} \in B \times M \to \text{BExp}$: Returns $t$ if the communication medium is in a state where it can deliver the message and $f$ otherwise.

We make use of these functions to define the asynchronous execution graph induced by sequence diagrams.

**Definition 11 (Asynchronous execution graph of sequence diagrams)**

The asynchronous execution graph of (general) sequence diagrams, written $EGA$, is the LTS that has $B \times D \times \Sigma \times \{\text{pos}, \text{neg}\}$ as its states, $E \cup T$ as its labels, and its transitions are exactly those that can be derived from the following rules:

\[
[d_i, \sigma, \text{mo}] \xrightarrow{\text{add}(\beta, m)} [d'_i, \sigma', \text{mo}] \in \text{EGA}
\]

\[
[d_i, \sigma, \text{mo}] \xrightarrow{\text{rm}(\beta, m)} [d'_i, \sigma', \text{mo}] \in \text{EGA}
\]

\[
[d_i, \sigma, \text{mo}] \xrightarrow{\text{ready}(\beta, m)} [d'_i, \sigma', \text{mo}] \in \text{EGA}
\]

The asynchronous trace semantics is defined in terms of the execution graph as we did in the previous sections.

**Definition 12 (Asynchronous trace semantics)** The trace semantics of a (general) sequence diagram with initial buffer $\beta$, written $[d]_\beta$, is defined by

\[
[d]_\beta = (\{s \in E^* \mid [\beta, d, \emptyset, \text{pos}] \xrightarrow{\text{add}} [\beta', d', \sigma, \text{pos}] \in EGA \land d' \equiv \text{skip}, \\
\{s \in E^* \mid [\beta, d, \emptyset, \text{pos}] \xrightarrow{\text{rm}} [\beta', d', \sigma, \text{neg}] \in EGA \land d' \equiv \text{skip}\})
\]
A.3 Policy Adherence for Sequence Diagrams

In this section, we define what it means for a system to adhere to a policy expressed by a sequence diagram.

Let the semantics of a system $S$ be the set of traces $\Phi$. Then system $S$ adheres to a sequence diagram $d$ describing a policy if none of the traces of $\Phi$ has a negative trace of $d$ as a sub-trace. A trace $s = \langle e_1, \ldots, e_n \rangle$ is a sub-trace of $t$, written $s \prec t$, iff

$$s_1 \preceq (e_1) \cdots \preceq s_n \preceq (e_n) \preceq s_{n+1} = t$$

for some $s_1, \ldots, s_{n+1} \in E^*$.

**Definition 13 (Policy adherence for sequence diagrams)** Let $d$ be a sequence diagram representing a policy whose trace semantics is given by $(H_p, H_n)$ and let $S$ be a system that is interpreted by a trace set $\Phi$. Then system $S$ adheres to policy $d$ iff

$$(s \in H_n \land t \in \Phi) \implies \neg(s \prec t)$$

B State Machines

In this section, we define the syntax and semantics of UML inspired state machines. We also define what it means for a system to adhere to a policy expressed as a state machine.

B.1 Syntax

A state machine is either *basic* in the sense that is does not consist of other state machines or *composite* in the sense that it consists of other state machines. In Sect. B.1.1 we define the syntax of basic state machines. The syntax of composite state machines is given in Sect. B.1.2.

B.1.1 Basic State Machines

To define the alphabet of basic state machines, we make use of the equivalence relation $\equiv$ on signals, messages, and events:

\[
\begin{align*}
\text{nm}(e_1, \ldots, e_n) &\equiv \text{nm}(e'_1, \ldots, e'_n) \\
(l_t, l_r, si) &\equiv (l_t, l_r, si') \quad \text{if } si = si' \\
(k, m) &\equiv (k, m') \quad \text{if } m = m'
\end{align*}
\]

With each equivalence class of signals, we associate a unique so-called identity signal of the form

$$\text{nm}(px_1, \ldots, px_n)$$

where $\text{nm}$ is a name and $px_1, \ldots, px_n$ are parameter variables.

We let $\downarrow \in \text{Si} \to \text{Si}$ be a function that takes a signal $si$ and yields the identity signal of the equivalence class of $si$. We have,

$$si = si' \iff si \downarrow = si' \downarrow$$
The function $\downarrow$ is lifted to messages, events, and sets of events as follows:

\[
\begin{align*}
(l_t, l_r, si) \downarrow &= (l_t, l_r, si \downarrow) \\
(k, m) \downarrow &= (k, m \downarrow) \\
E \downarrow &= \{e \downarrow \mid e \in E\}
\end{align*}
\]

For convenience, we require that the alphabet of state machines consist of identity events only.

Transitions of a state machine are labeled by actions of the form $(e, bx, a)$ where $e$ is an event trigger, $bx$ is a guard, and $a$ is an assignment. Intuitively, the action means: if the event trigger occurs and the guard evaluates to true, then execute the assignment.

The set of actions induced by a set of events $E$, written $\text{Act}_E$, is defined by

\[
\text{Act}_E \overset{\text{def}}{=} \{(e \in E \mid \exists e' \in E : e = e') \cup \{\epsilon\} \times \text{BExp} \times \text{A}\}
\]

Note that event triggers are optional in an action. An action without an event trigger is of the form $(\epsilon, bx, a)$.

We are now ready to define the syntax of basic state machines.

**Definition 14 (Syntax of basic state machines)** A state machine is a tuple $(E, Q, R, q_i, F)$ consisting of

- an alphabet $E \subseteq E \downarrow$;
- a set of states $Q$;
- a transition relation $R \subseteq Q \times \text{Act}_E \times Q$;
- an initial state $q_i \in Q$;
- a set of final states $F \subseteq F$

The set of all basic state machines is denoted by $P$.

We define the functions for obtaining the alphabet, states, transitions, initial state, and final states of a basic state machine:

\[
\begin{align*}
\text{alph}((E, Q, R, q_i, F)) &\overset{\text{def}}{=} E \\
\text{states}((E, Q, R, q_i, F)) &\overset{\text{def}}{=} Q \\
\text{trans}((E, Q, R, q_i, F)) &\overset{\text{def}}{=} R \\
\text{init}((E, Q, R, q_i, F)) &\overset{\text{def}}{=} q_i \\
\text{final}((E, Q, R, q_i, F)) &\overset{\text{def}}{=} F
\end{align*}
\]

**B.1.2 Composite State Machine**

A composite state machine is a state machine that consists of basic state machines.

**Definition 15 (Composite state machine)** A composite state machine $P_{C}$ is a set of basic state machines. The set of all composite state machines is denoted by $P_{C}$.
We define the functions for obtaining the alphabet, states, initial state, and final
states of a composite state machine:

\[
\begin{align*}
\text{alphabet}(\{P_1, \ldots, P_n\}) & \overset{\text{def}}{=} \text{alphabet}(P_1) \cup \ldots \cup \text{alphabet}(P_n) \\
\text{states}(\{P_1, \ldots, P_n\}) & \overset{\text{def}}{=} \{q_1, \ldots, q_n\} | q_i \in \text{states}(P_i), \ldots, q_n \in \text{states}(P_n) \\
\text{initial}(\{P_1, \ldots, P_n\}) & \overset{\text{def}}{=} \{\text{initial}(P_1), \ldots, \text{initial}(P_n)\} \\
\text{final}(\{P_1, \ldots, P_n\}) & \overset{\text{def}}{=} \{q_1, \ldots, q_n\} | q_i \in \text{final}(P_i), \ldots, q_n \in \text{final}(P_n) 
\end{align*}
\]

### B.2 Semantics

In this section, we define the trace semantics of basic and composite state machines.

#### B.2.1 Trace Semantics of Basic State Machines

We define the trace semantics of basic state machines as we defined the semantics of single lifeline sequence diagrams in Sect. A.2.2. First, we define the execution graph of basic state machines, then we define the trace semantics in terms of the execution graph.

**Definition 16 (Execution graph of basic state machines)** The execution graph of basic state machine \(P = (E, Q, R, q_i, F)\), written \(\text{EG}(P)\), is the LTS \((Q \times \Sigma, R')\) over \(E \cup \{\tau_i, \tau_{\text{assign}}\}\) whose transition relation \(R'\) is defined by the rules

\[
\begin{align*}
\frac{q \xrightarrow{(e,bx,e)} q'}{[q, \sigma] \xrightarrow{(e,bx,e)} [q', \sigma]} & \quad \text{if } \text{eval}(\sigma(bx)) = t \\
\frac{q \xrightarrow{(e,bx,e)} q'}{[q, \sigma] \xrightarrow{(e,bx,e)} [q', \sigma]} & \quad \text{if } \text{cond}(\sigma, \sigma', bx, e) \\
\frac{q \xrightarrow{(e,bx,\text{assign}(x,ex,l))} q'}{[q, \sigma] \xrightarrow{(e,bx,\text{assign}(x,ex,l))} [q', \sigma]} & \quad \text{if } \text{eval}(\sigma(bx)) = t \\
\frac{q \xrightarrow{(e,bx,\text{assign}(x,ex,l))} q'}{[q, \sigma] \xrightarrow{(e,bx,\text{assign}(x,ex,l))} [q', \sigma]} & \quad \text{if } \text{cond}(\sigma, \sigma', bx, e)
\end{align*}
\]

where the predicate \(\text{cond}\) is defined by

\[
\text{cond}(\sigma, \sigma', bx, e) \iff \text{eval}(\sigma[\sigma](bx)) = t \land \text{Dom}(\sigma') = p \land e \land \text{eval}(\sigma[\sigma'](e)) \neq \bot
\]

The trace semantics of a basic state machine is the set of sequences obtained by recording the events occurring in each path from the initial state to a final state of the state machine.

**Definition 17 (Trace semantics of basic state machines)** The trace semantics of a basic state machine \(P = (E, Q, R, q_i, F)\), written \([P]\), is defined by

\[
[P] \overset{\text{def}}{=} \{s \in E^* | [q_i, 0] \Rightarrow [q', \sigma] \in \text{EG}(P) \land q' \in F\}
\]
We assume that transitions that are labeled by action expressions that do not contain any events are always taken immediately by the state machine. Thus, for simplicity, we do not consider state machines that has one or more states whose outgoing transitions both a labeled by action that contain events and action that do not contain events. Also, we assume that all the actions of all transitions leading to accepting states contain events.

B.2.2 Synchronous Trace Semantics of Composite State Machines

We define the synchronous trace semantics of a composite state machine in the same way as we defined the synchronous trace semantics of sequence diagrams (see Sect. A.2.3).

Definition 18 (Synchronous execution graph of composite state machines)
The synchronous execution graph of a composite state machine \( P_C = \{P_1, \ldots, P_n\} \), written \( EGS(P_C) \), is the LTS whose states are \( \{\{q_1, \sigma_1\}, \ldots, \{q_n, \sigma_n\}\} \mid q_1 \in states(P_1) \land \cdots \land q_n \in states(P_n) \land \sigma_1, \ldots, \sigma_n \in \hat{\Sigma} \} \), whose labels are \( M \cup \{\tau_{\text{r}}, \tau_{\text{assign}}\} \), and whose transitions are defined by the rules

\[
q_i \xrightarrow{\tau} q'_i \in EG(P_i) \\
\frac{}{[Q \cup \{q_i\}] \xrightarrow{\tau} [Q \cup \{q'_i\}]}
\]

\[
q_i \xrightarrow{(\cdot,m)} q'_i \in EG(P_i) \quad q_j \xrightarrow{(\cdot,m)} q'_j \in EG(P_j) \\
\frac{}{[Q \cup \{q_i, q_j\}] \xrightarrow{m} [Q \cup \{q'_i, q'_j\}]}
\]

The trace semantics is given by the following definition.

Definition 19 (Synchronous trace semantics of composite state machines)
The synchronous trace semantics of state machine \( P_C = \{P_1, \ldots, P_n\} \), written \( [P_C] \), is defined by

\[
[P_C] \triangleq \{s \in M^* \mid Q_i \xrightarrow{Q_j} Q_f \in EGS(P_C)\}
\]

where

\[
Q_i = \{\text{init}(P_1), \emptyset, \ldots, \text{init}(P_n), \emptyset\} \\
Q_f = \{q_1, \sigma_1\}, \ldots, \{q_n, \sigma_n\} \quad \text{for} \quad \{q_1, \ldots, q_n\} \in \text{final}(P_C)
\]

B.2.3 Asynchronous Trace Semantics of Composite State machines

To define the asynchronous trace semantics of composite state machines, we assume the functions \( \text{ready}, \text{rm} \), and \( \text{add} \) (see Sect. A.2.4).

Definition 20 (Asynchronous execution graph of composite state machines)
The asynchronous execution graph of a composite state machine \( P_C = \{P_1, \ldots, P_n\} \), written \( EGA(P_C) \), is the LTS whose states are \( B \times \{\{q_1, \sigma_1\}, \ldots, \{q_n, \sigma_n\}\} \mid q_1 \in states(P_1) \land \cdots \land q_n \in states(P_n) \land \sigma_1, \ldots, \sigma_n \in \Sigma\} \), whose labels are \( E \cup \{\tau_{\text{r}}, \tau_{\text{assign}}\} \), and whose transitions are defined by the rules

\[
q_i \xrightarrow{\tau} q'_i \in EG(P_i) \\
\frac{}{[\beta, Q \cup \{q_i\}] \xrightarrow{\tau} [\beta, Q \cup \{q'_i\}]}
\]
The trace semantics is given by the following definition.

**Definition 21 (Asynchronous trace semantics of composite state machines)**

The asynchronous trace semantics of composite state machine $P_C = \{P_1, \ldots, P_n\}$, written $\llbracket P_C \rrbracket_\beta$, is defined by

$$\llbracket P_C \rrbracket_\beta = \{ s \in E^* \mid [\beta, Q_i] \xrightarrow{s} [\beta', Q_f] \in EGS(P_C) \}$$

where

$$Q_i = \{ [init(P_1), \emptyset], \ldots, [init(P_n), \emptyset] \}$$

$$Q_f = \{ [q_1, \sigma_1], \ldots, [q_n, \sigma_n] \} \quad \text{for } \{q_1, \ldots, q_n\} \in \text{final}(P_C)$$

### B.3 Policy Adherence

In this section, we define what it means for a system to adhere to a policy expressed as a state machine.

Let $S$ be a system whose semantics is given by a set of traces $\Phi$, then $S$ adheres to a state machine policy $P_C$ if each trace of $\Phi$ (when projected to the alphabet of $P_C$) is a trace in the semantics of $P_C$.

To make this precise, we make use of the projection function $|_A$ which takes a set $A$ and a sequence $s$ and yields the sequence $s|_A$ obtained from $s$ by removing all elements not in $A$.

**Definition 22** Let $P_C$ be a state machine defining a policy, let $\Phi_p$ denote the semantics of $P_C$, and let $S$ be a system whose semantics is a set of traces $\Phi_s$, then $S$ adheres to $P_C$ iff

$$s \in \Phi_s \implies s|_E \in \Phi_p$$

with $E = \{ e \mid e' \in \text{alph}(P_C) \land e = e' \}$.

### C From High-level to Low-level Sequence Diagrams

In this section, we define a simple and useful method for expressing transformations in terms of sequence diagram patterns.

**Definition 23 (Sequence diagram pattern)** The set of sequence diagram patterns $DP$ is defined by the following syntax

$$dp ::= mv | ep | \text{constr}(mb, l) | \text{assign}(x, mx, l) | \text{refuse}(dp) | \text{loop}(0..*) (dp) | dp_1 \text{ seq } dp_2 | dp_1 \text{ alt } dp_2 | dp_1 \text{ par } dp_2$$
A sequence diagram pattern is either a meta-variable (mv), an event pattern (ep), a constraint with a condition that may contain meta-variables, and assignment whose expression may contain meta variables, or the composition of one or more diagram patterns.

We let \( mv \in MVar \) denote the set of all meta variables. The set \( EP \) of all event patterns is defined

\[
ep \in EP \overset{\text{def}}{=} K \times ((L \cup MVar) \times (L \cup MVar) \times (M \cup MVar))
\]

An expression with meta variables and a boolean expression with meta variables is denoted by \( mx \) and \( mb \), respectively.

A substitution is a function \( \phi \in MVar \rightarrow (D \cup Exp) \) that replaces meta variables by diagrams or expressions. Any substitution \( \phi \) is lifted to diagram patterns such that \( \phi(dp) \) yields the diagram obtained from \( dp \) by simultaneously replacing all meta variables in \( dp \) by diagrams or expressions according to \( \phi \).

The set of all substitution is denoted by \( Sub \).

A diagram pattern \( dp \) matches a diagram \( d \) if there is a substitution \( \phi \) such that \( \phi(dp) = d \)

The domain of a pattern \( dp \), written \( Dom(dp) \), is the set of all diagrams that are matched by \( dp \), i.e.

\[
Dom(dp) \overset{\text{def}}{=} \{ \phi(dp) \mid \phi \in Sub \}
\]

A transformation rule is a pair \( (dp, dp') \) of two patterns; one left hand side (lhs) pattern \( dp \), and one right hand side (rhs) pattern \( d \).

**Definition 24 (Transformation specification)** A transformation specification is a set of transformation rules. We require that the domain of the transformation rules are disjoint. The transformation induced by a transformation specification \( ts = \{(dp_1, dp'_1), (dp_2, dp'_2), \ldots, (dp_n, dp'_n)\} \), written \( T_{ts} \), is defined

\[
T_{ts}(d) \overset{\text{def}}{=} \phi(dp') \quad \text{if} \quad \phi(dp) = d \quad \text{for some} \quad (dp, dp') \in ts \quad \text{and} \quad \phi \in Sub
\]

\[
T_{ts}(op(d, d')) \overset{\text{def}}{=} op(T_{ts}(d), T_{ts}(d')) \quad \text{for} \quad op \in \{\text{seq, alt, par}\}
\]

\[
T_{ts}(op(d)) \overset{\text{def}}{=} op(T_{ts}(d)) \quad \text{for} \quad op \in \{\text{loop, refuse}\}
\]

D From Sequence Diagrams to State Machines

In this section, we define a transformation from sequence diagrams to state machines. We assume that sequence diagrams are on standard form (Def. 8).

First, we define the transformation from single lifeline diagrams to basic state machines, then we define the transformation from (general) sequence diagrams to composite state machines.

The transformation from a single lifeline diagram to a basic state machine has two phases. In phase 1, the sequence diagram is transformed into a state machine \( P \) whose trace semantics equals the negative trace set of \( d \). In phase 2, \( P \) is inverted into the state machine \( P' \) whose semantics is the set of all traces that do not have a trace of \( P \) as a sub-trace.
Definition 25 (Single lifeline sequence diagram to basic state machine)
The transformation \( d_{2p} \) from single lifeline diagrams to basic state machines is defined by
\[
d_{2p} = \text{ph}_2 \circ \text{ph}_1
\]
where \( \text{ph}_1 \) and \( \text{ph}_2 \) represent phase 1 and 2 (as defined below).

The transformation of phase 1 is given by the following definition.

Definition 26 (Phase 1) We let \( \text{ph}_1 : \mathcal{D} \to \mathcal{P} \) be the function that takes a sequence diagram on lifeline \( l \) and yields a state machine describing the negative traces of \( d \). The state machine \( \text{ph}_1(d) = (E, Q, R, q_i, F) \) obtained from sequence diagram \( d \) on lifeline \( l \) is defined by the rules

- \( E = \text{events}(d) \upharpoonright \) where \( \text{events}(d) \) yields all events in diagram \( d \);
- \( Q = D \times \{\text{pos}, \text{neg}\} \);
- \( q_i = (d, \text{pos}) \);
- \( F = \{(\text{skip}, \text{neg})\} \);
- \( R \) is defined by the rules

\[
\begin{align*}
\frac{d \xrightarrow{(b,x,e)} d'}{d \xrightarrow{(b,x,e)} d'} \in \text{LTS}(l) & \quad \frac{d \xrightarrow{(b,x,a)} d'}{d \xrightarrow{(b,x,a)} d'} \in \text{LTS}(l) \\
\frac{(d, mo) \xrightarrow{(e,bx,e)} (d', mo)}{(d, mo) \xrightarrow{(e,bx,a)} (d', mo)} & \quad \frac{(d, mo) \xrightarrow{(e,bx,e)} (d', neg)}{(d, mo) \xrightarrow{(e,bx,a)} (d', neg)}
\end{align*}
\]

In phase 2, the state machine obtained from phase 1 is inverted into the state machine \( P' \) whose semantics is the set of all traces that do not have a trace of \( P \) as a sub-trace.

Definition 27 (Phase 2) We let \( \text{ph}_2 : P \to P \) be the function that takes a basic state machine and yields its inversion. The state machine \( \text{ph}_2(P) = (E', Q', R', q'_i, F') \) for \( P = (E, Q, R, q_i, F) \) is the state machine \( (E', Q', R', q'_i, F') \) defined by

- \( E = E' \)
- \( Q' = \mathcal{P}(Q) \times \mathcal{P}(R) \)
- \( q'_i = (\{q_i\}, \emptyset) \)
- \( F' = \mathcal{P}(Q) \)
- \( R' \) is defined by

\[
\begin{align*}
q \xrightarrow{(e,bx,a)} q' \in (R \setminus V) & \quad q' \notin F \\
(Q \cup \{q\}, V) & \xrightarrow{(e,bx,a)} (Q \cup \{q'\}, V \cup \{(q, (e, bx, a), q')\}) \\
q \xrightarrow{(e,bx,a)} q' \in (R \setminus V) & \quad q' \notin F \\
(Q \cup \{q\}, V) & \xrightarrow{(e,bx,a)} (Q \cup \{q\} \cup \{q'\}, V \cup \{(q, (e, bx, a), q')\}) \\
Q & \xrightarrow{(e,bx,e)} \bot \in R \quad e \in E \\
(Q, V) & \xrightarrow{(e,bx,e)} (Q, V)
\end{align*}
\]
The first two rules ensure that $P'$ describes the positive traces, but not the negative traces, of the sequence diagram that $P$ was transformed from. The last rule ensures that $P'$ describes the inconclusive behavior of the sequence diagram that $P$ was transformed from.

The condition $(Q, V) \xrightarrow{(e, bx, a)} \bot \in R$ should hold if $(e, bx, e)$ is not enabled in $P$ for any state in $Q$. To ensure this, we require that $bx$ is the normal form of the negation of the disjunction of all guards occurring on transitions that are enabled on $e$ in $Q$. We need some auxiliary function to define this more precisely.

We let $\text{guards}$ be the function that collects the set of boolean expressions occurring in the actions of all transitions that are enabled on event $e$ in the states of state set $Q$ in transition relation $R$. Formally,

$$\text{guards}(Q, e, R) \overset{\text{def}}{=} \{ bx \in B\text{Exp} | q \in Q \land q \xrightarrow{(e', bx, a)} q' \in R \land e = e' \}$$

To ensure that the guards that are enabled on event $e$ contain the same parameter variables as $e$, we make use of a renaming function $rnm : \text{PVar} \rightarrow \text{PVar}$. The function is lifted to boolean expressions such that $rnm(bx)$ yields the expression obtained from $bx$ by renaming all parameter variables of $bx$ according to the function $rnm$. We let $Rnm$ denote the set of all renaming functions.

The functions that renames the parameter variables in a set of boolean expressions $Bx$ to the parameter variables of event $e$ is defined as follows

$$rnmBx(e, Bx) \overset{\text{def}}{=} \{ rnm(bx) | bx \in Bx \land rnm \in Rnm \land pvar(rnm(bx)) \subseteq pvar(e) \}$$

We let $nf : \mathcal{P}(B\text{Exp}) \rightarrow \mathcal{P}(B\text{Exp})$ be the function that yields the normal form of the negation of a set of boolean expressions. In particular, we let $nf(\emptyset) = t$.

We write $te(Q, R)$ if no transition from a state in $Q$ in $R$ has a transition that does not contain an event, i.e,

$$te(Q, R) \iff q \xrightarrow{(e, bx, a)} q' \notin R$$

for all $q \in Q$, states $q'$, boolean expressions $bx$, and assignments $a$.

An action $(bx, e, e)$ is not enabled for state set $Q$ and transition relation $R$, written $Q \xrightarrow{(bx, e, e)} \bot \in R$, iff

$$(bx = nf(rnmBx(\text{guards}(Q, e, R)))) \land te(Q, R)$$

The transformation that takes a (general) sequence diagram on standard form and yields a composite state machine is given by the following definition.

**Definition 28 (From (general) sequence diagrams to composite state machines)**

The transformation $d2pc : D \rightarrow \mathcal{P}^2$ which takes a sequence diagram and yields a composite state machine, is defined by

$$d2pc(d_1 \text{par} \cdots \text{par} d_n) \overset{\text{def}}{=} \{ d2p(p_1), \ldots, d2p(p_n) \}$$