SINTEF

SINTEF REPORT

SINTEF ICT

Address: NO-7465 Trondheim
NORWAY
Location Trondheim:
S.P. Andersens v 15
Location Oslo:
Forskningsveien 1
Telephone: +47 73 59 30 00
Fax: +47 73 59 43 02

Enterprise No.: NO 948 007 029 MVA

TITLE

A Method for Model-Driven Information Flow Security

AUTHOR(S)

Fredrik Seehusen and Ketil Stølen

CLIENT(S)

Norwegian Research Council (NCR) and European Commission (EC)

REPORT NO. A11357	CLASSIFICATION Unrestricted	CLIENTS REF. NCR: 152839 and 180052, and EC: FP6-IST-27004	
CLASS. THIS PAGE Unrestricted	ISBN 978-82-14-04434-8	PROJECT NO. 90B230, 90B245, and 403328	NO. OF PAGES/APPENDICES 56/4
ELECTRONIC FILE CODE		PROJECT MANAGER (NAME, SIGN.) Ketil Stølen [illegible]	CHECKED BY (NAME, SIGN.) Mass Soldal Lund [illegible]
FILE CODE	DATE 2009-03-31	APPROVED BY (NAME, POSITION, SIGN.) Bjørn Skjellaug, Research director [illegible]	

ABSTRACT

We present a method for software development in which information flow security is taken into consideration from start to finish. Initially, the user of the method (i.e., a software developer) specifies the system architecture and selects a set of security requirements (in the form of secure information flow properties) that the system must adhere to. The user then specifies each component of the system architecture using UML inspired state machines, and refines/transforms these (abstract) state machines into concrete state machines. It is shown that if the abstract specification adheres to the security requirements, then so does the concrete one provided that certain conditions are satisfied.

KEYWORDS	ENGLISH	NORWEGIAN
GROUP 1	ICT, modelling	IKT, modellering
GROUP 2	Design	Design
SELECTED BY AUTHOR	MDA, Information flow security,	MDA, Sikker informasjonsflyt
	Refinement, Transformation	Raffinering, Transformasjon

A Method for Model-Driven Information Flow Security

Fredrik Seehusen[1,2] and Ketil Stølen[1,2]
[1] SINTEF ICT, Norway
{Fredrik.Seehusen, Ketil.Stolen}@sintef.no
[2] Department of Informatics, University of Oslo, Norway

2009

Abstract

We present a method for software development in which information flow security is taken into consideration from start to finish. Initially, the user of the method (i.e., a software developer) specifies the system architecture and selects a set of security requirements (in the form of secure information flow properties) that the system must adhere to. The user then specifies each component of the system architecture using UML inspired state machines, and refines/transforms these (abstract) state machines into concrete state machines. It is shown that if the abstract specification adheres to the security requirements, then so does the concrete one provided that certain conditions are satisfied.

1 Introduction

Security incidents occur on a daily basis within many companies. In CSI/FBI Computer Crime and Security Survey for 2005 [9], 74% of the companies reported security incidents. Despite the importance of security, careful engineering of security into overall design is often neglected and security features are typically built into an application in an ad-hoc manner or are only integrated during the final phases of system development [22].

Model-Driven Security (MDS) [3] advocates the opposite. MDS aims to raise the level of abstraction in design and development of secure systems by supporting (1) a model-driven development process in which security is taken into account from start to finish, (2) a clear separation of abstract, platform independent models (PIMs) and refined, platform specific models (PSMs), and (3) adherence preserving transformations between PIMs and PSMs.

The central idea of MDS is that systems can be specified and shown to be in adherence with security requirements at different levels of abstraction. Abstraction is believed to simplify analysis, facilitate reuse of designs and early discovery of design flaws. At each level of abstraction, we distinguish between a system specification and a set of security requirements the system specification must adhere to. When we have established that a system specification adheres to the security requirements at a given level, this relationship should also hold

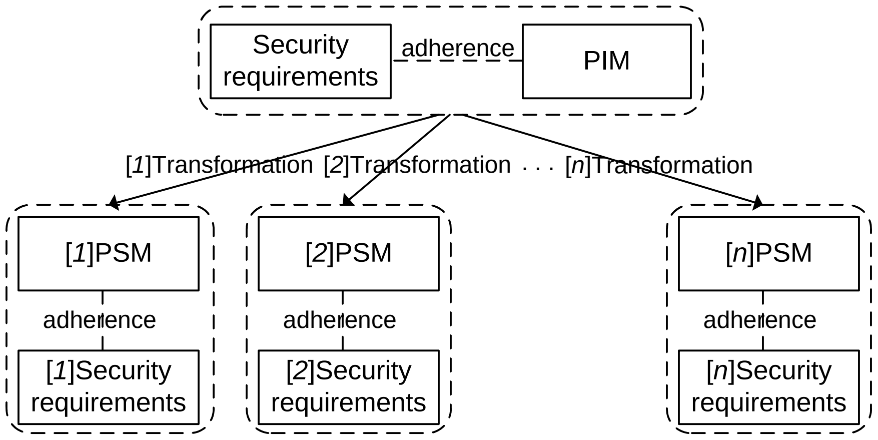

Figure 1: Model-driven security (pictured)

at the next level so that the invested effort to establish adherence is not wasted. Hence, we would like adherence to be *preserved* under transformation. Adding details to a specification may of course require some additional analysis. However, it should not be necessary to recheck the adherence relationship already established at the more abstract level.

The MDS framework is illustrated in Fig.1. Here a platform independent model together with its security requirements are transformed into several platform specific models with associated security requirements.

Already published approaches to MDS include [2, 3, 5, 8, 10, 14, 22, 37]. Although interesting, they are in most cases of a rather informal nature; the semantics of the languages used (at abstract or/and concrete levels) are not sufficiently precise to allow for rigorous reasoning at more than one level of abstraction. Moreover, some of the approaches ([2, 3, 8, 22]) consider transformation of security requirements only, and not transformation of system specifications. These approaches allow adherence checking only at the lowest level of abstraction. Others ([5, 8, 10, 37]) do not clearly characterize what it means for a system to adhere to a security requirement. Instead, security is described in terms of a *security mechanism*.

Security is often defined as the preservation of confidentiality, integrity, and availability [15]. In this report, we, however, focus on security in the more narrow sense of *secure information flow properties* (see e.g., [4, 11, 25, 27, 30, 31, 35, 38]) which provide an elegant way of specifying confidentiality as well as integrity requirements [26].

The notion of transformation is closely related to *refinement*. That is, refinement is the (possibly manual) process of making an abstract specification more concrete, whereas transformation typically is a special case of refinement in which this process is automatic. There are several kinds of refinement. One of the refinement notions considered in this report, is refinement w.r.t. *underspecification*, i.e., the process of removing alternative design choices that are equivalent in the sense that it suffices for an implementation to provide only of them. In the classical literature, this kind of refinement is often referred to as *behavioral refinement* or *property refinement* [6]. It has long been recog-

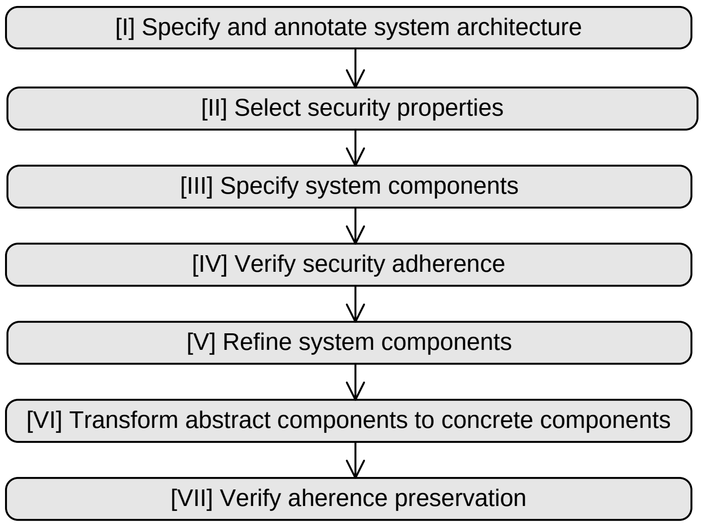
[I] Specify and annotate system architecture
[II] Select security properties
[III] Specify system components
[IV] Verify security adherence
[V] Refine system components
[VI] Transform abstract components to concrete components
[VII] Verify aherence preservation

Figure 2: Overview of method

nized that secure information flow properties are not preserved under standard definitions of refinement w.r.t. underspecification [16]. We avoid this problem by defining refinement in a semantic framework which is more expressive than conventional frameworks. In our method, system specifications are written in a state machine notation inspired by UML. But contrary to UML state machines, we have two constructs of choice; one describing alternative *design choices*, and one describing choices which should be provided by the *system*. The distinction between the two kinds of choices is necessary in order to handle refinement of secure information flow properties [13, 17, 31].

Software systems are almost never built entirely from scratch, i.e., a set of already implemented operations is usually available from the operating systems, runtime environments, or from libraries of programming environments. To take this into account, our state machines are allowed to *reference* events that are already available in a predefined *event library*. These events are later substituted by their definitions by a so-called *event transformation*. The transformation may therefore be understood as a special case of what in the literature is known as *action refinement* [36]. We provide a formal characterization of the syntax and semantics of transformations that are induced by event libraries, and define general conditions under which transformations induced by event libraries preserve arbitrary secure information flow properties.

This report is structured as follows: Sect. 2 gives an overview of our method. In Sect. 3 - 9 the individual steps of our method are presented. Sect. 10 discusses related work, and Sect. 11 provides conclusions and directions of future work. In the appendix, we formally define state machines (App. A) and event transformations (App. B). A glossary of symbols is provided in App. C and proofs are presented in App. D.

2 Overview of method

The goal of our method for model-driven information flow security is to support the specification and development of secure software systems. As illustrated in Fig. 2, the method has seven main steps.

In step I, the user of our method (i.e., a software developer) specifies the system architecture using UML composite structures. The system architecture is an overview of the components of the system and their associated communication channels. The user then partitions the system into security domains by labeling the system architecture with security relevant annotations.

In step II, the user selects a set of secure information flow properties (typically from a library) that the specification must adhere to. The secure information flow properties of the library are assumed to be defined in our security property schema [33, 34] which ensures that the adherence to the properties is preserved when design decisions are resolved by refinement in later steps.

In step III, the user specifies each component of the system architecture using our UML inspired state machines. The state machine notation provides constructs for specifying both design choices and choices that must be offered by the components of the system. The specified state machines may reference events that are already provided in a predefined event library. In this step, new event specifications may also be uploaded to the event library.

In step IV, the user verifies that the state machine based specification adheres to the security properties selected in step II. There are many techniques and methods that can be used for this purpose. We do not go into details on these. However, we provide a precise characterization of what it means for a system specification to adhere to a secure information flow property. This characterization provides a formal foundation for adherence verification.

In step V, the user refines the specification by removing alternative design decisions until all design decisions are decided. The validity of adherence is guaranteed to be preserved under this kind of refinement.

In step VI, the state machine specification of step V is transformed into a more concrete specification by substituting the event references of the specification by their definitions in the event library. We give a formal characterization of the syntax and semantics of these so-called event transformations, and show that they satisfy some desirable properties.

In step VII, the user verifies that the event transformation of step VI preserves adherence to the security properties selected in step II. We present conditions that can be used to check that the transformation preserves adherence.

In the MDS terminology, the specification produced in step V may be seen as a platform independent model (PIM), whereas the state machine produced in step VII may be seen as a platform specific model (PSM). Note that although a fully general approach to MDS would consider the transformation of both (abstract) security properties and system specification into (concrete) security properties and system specifications, we only consider the transformation of the system specifications. The reason for this is that the security properties we consider are essentially parameterized w.r.t. to the different levels of abstraction. Hence, there is normally no need to transform the actual specification of the properties.

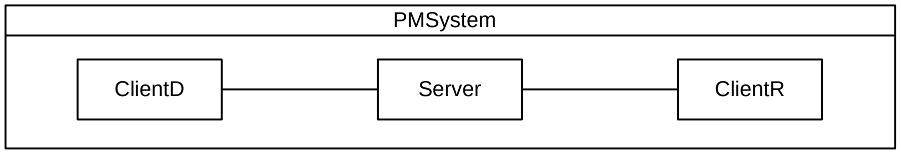

Figure 3: Architecture of the PM system

3 Step I: Specify and annotate the system architecture

In step I of our method, the user specifies the system architecture, i.e., an overview of the *basic* state machines that the system specification consists of and their associated communication channels. We will often refer to the architecture specification as a *composite* state machine. After the architecture has been specified, the user partitions the system into security domains by annotating the specification. In the following, we introduce a running example which will be used to explain this step and the subsequent steps of our method.

Consider a large software developing company that aims to develop a distributed system, the project management system (the PM system), in order to centralize all storage of software development projects. Software developers should be able to retrieve projects from a server to their local machines, edit or add files to the project, and upload any changes back to the server.

Currently, the company has no unified development method, and developers working on different projects are to a large degree given flexibility in the method they choose to adopt. The company wants to assess the different methods in order to recommend improvements, and possibly to introduce a unified development process. This task is assigned to a group of researches. The researchers are to pick a set of sample projects, and assess each project thoroughly with respect to progress, quality of code etc. For convenience, the PM system should be augmented slightly such that the researchers will be able to retrieve projects from the server over the Internet on a regular basis. This additional functionality is not of high priority, thus it will be implemented with little resources. It is not initially known whether the researchers should use the same client as the developers.

To make the assessment as realistic as possible, the developers should not know which projects the researchers have sampled for the assessment. As we will see later, this requirement may be enforced by selection an appropriate secure information flow property.

Fig. 3 shows the architecture specification of the PM system with one client acting on the behalf of the developer (ClientD), one client acting on the behalf of the researcher (ClientR), and one server (Server). The architecture specification is intended to give an overview of the components or basic state machines that the system specification consists of. As a graphical notation, we have chosen to use UML composite structures (see App. A for more details).

After the system architecture has been specified, the user partitions the architecture into security domains by annotating the components with the security domain they belong to. In our running example, developers are not allowed to find out which projects the researchers have sampled for the assessment. Thus

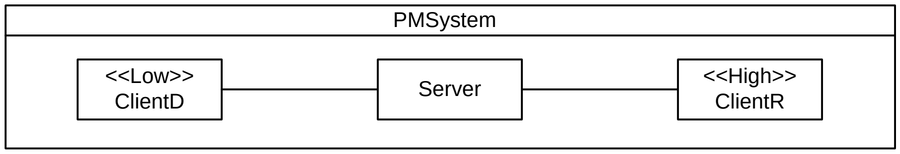

Figure 4: Annotated architecture of the PM system

we let the developers belong to one security domain which we call the *low level domain*, and the researchers belong to another domain, which we call the *high level domain*. We require that information should not flow from the high level domain to the low level domain.

In Fig. 4, the domains that the developer and the researcher clients belong to are specified by the labels $<<$ Low $>>$ and $<<$ High $>>$, respectively. For short, we denote the low level domain by L, and the high level domain by H. Semantically, we interpret a security domain by a set of *events*, where an event represents the transmission or the reception of a message. In the current example, this means that L denotes all events that represent messages that can be sent to or from ClientD, whereas H denotes all events that represent messages that can be sent to or from ClientR.

4 Step II: Select security properties

In step II, the user selects a set of secure information flow properties, referred to as security properties for short, that the system must adhere to. A security property defines what it *means* that information flows from one security domain to another.

The security requirement of the running example may be formalized by the security property *non-inference* [30] denoted NF. This is one of the most well-known security properties of the literature. It treats all the behavior of the high level domain H (the behavior of the researchers in this example) as confidential, and requires that the low level user (the developer) must not deduce that any event in H has occurred.

We assume (1) that the developer may observe all events in L, i.e., its own communication with the server and (2) that the developer has complete knowledge of the system specification. Therefore, for each observation (i.e., a sequence of events in L) that the developer can make by interacting with the server, the developer can look at the specification and construct the so-called *low level equivalence set* of all traces (i.e., sequences of events describing system executions) that are *compatible* with that observation. The developer will know that one of the traces in this set has occurred, but not which one. However, if all the traces of this set include a high level event in H, then the developer can conclude with certainty that the researcher has done something. In this case, the specification would not be secure w.r.t. the NF property. Conversely, if there is one trace in the low level equivalence set which does not include a high level event, then the developer cannot conclude with certainty that the researcher has done something.

In general, the underlying idea of all secure information flow properties is to demand that each low level equivalence set must contain a trace which prevents the low level user from deducing that some confidential behavior has (or has not) occurred.

The above discussion suggests that all security properties have two essential ingredients:

- a definition of low level equivalence;
- a definition of the high level behavior which should be regarded as confidential.

Formally, two traces s and t are low level equivalent, written $s \sim_{\mathfrak{l}} t$, if they contain the same sequences of low level events, i.e.,

$$s|_L = t|_L \tag{1}$$

where $t|_L$ yields the trace obtained from t by removing all events not in the set of events L. The definition of low level equivalence is the same for all security properties. The definition of confidential behavior, however, differs depending on the property. For the NF property – which treats traces that contain any high level event in H as confidential – the set of confidential behavior C may be defined by the following predicate

$$C(t) \overset{\mathsf{def}}{=} t|_H \neq \langle\rangle \tag{2}$$

where $\langle\rangle$ denotes the empty trace. In other words, a trace is confidential if it is non-empty when all non high level events have been removed from it.

For a system S, whose set of possible traces is denoted by $[\![S]\!]$, to adhere to the NF property, there must, for each low level observation that can be made from $[\![S]\!]$, be a trace in $[\![S]\!]$ which is not confidential (i.e., not included in C) and which is low level equivalent to the observation. Formally we have

$$\text{NF}([\![S]\!]) \overset{\mathsf{def}}{=} \forall t \in [\![S]\!] : \exists u \in [\![S]\!] : \neg C(u) \wedge u \sim_{\mathfrak{l}} t \tag{3}$$

All security properties are of a similar form as (3). In fact, NF may be expressed as an instance of a security predicate schema which is parameterized by a predicate which characterizes confidential behavior (see [34, 33]). We assume that all security properties that can be selected by the user in step II are instances of the security property schema.

5 Step III: Specify system components

In step III of our method, the user specifies each basic state machine of the architecture. It is assumed that a library of predefined event specifications is available to the user. These events can be referenced in the state machine specifications.

Fig. 5 shows the specification of the two clients ClientD and ClientR. ClientD receives an input document from its user (not shown), and passes that document on to the server by sending the message storeDoc. In the state machine notation, the black circle represents the initial state and the boxes with rounded edges represent standard states. Transitions are specified by arrows between states. Transitions may be labeled by *action expressions* of the form $nm.si[bx]/ef$, where $nm.si$ is an *input event*, $[bx]$ (where bx is a boolean expression) is a

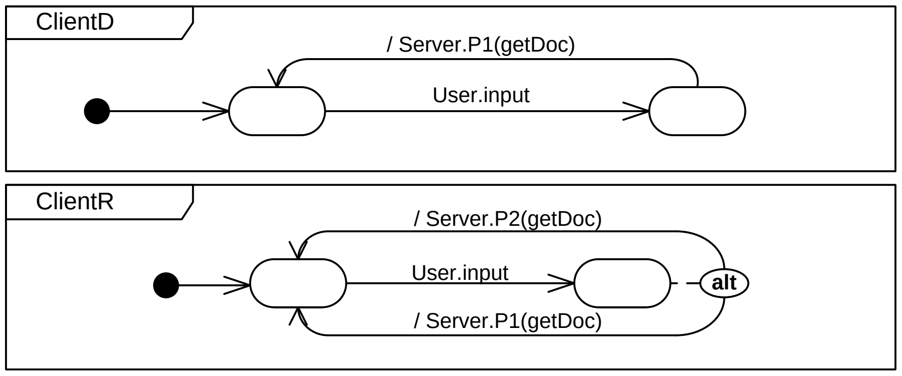

Figure 5: Specification of the clients

guard, and ef is an *effect*. Event triggers, guards, and effects are all optional in the expression.

Intuitively, if the current state of the state machine is q, and the state machine has a transition from q to state q' that is labeled by $nm.si[bx]/ef$, then the state machine will set its current state to q' if it receives signal si from the basic state machine whose name is nm and the guard bx is evaluated to true. The effect ef will be executed when the state machine moves to q'. There are two kinds of effects: output events and assignments. An output event is an expression of the form $nm.si$ that represents the transmission of a signal si to the state machine whose name is nm. Assignments are denoted by expressions of the form $x = ex$ where x is a variable and ex is a term built from common basic types such as booleans, integers, and strings, and common operations for these.

The graphical notation used for specifying state machines is essentially a subset of the UML state machine notation. See App. A for more details.

We assume that the message transmission protocol between the server and the client is provided by the event library. As specified in Fig. 5, the clients invoke the action P1 (for protocol 1) in order to send messages to the server.

The client acting on the behalf of the researcher (ClientR) is similar to ClientD, except for two differences. First, the researcher retrieves documents from the server instead of storing them, and second ClientR has the choice of either using protocol P1 or using protocol P2. This is indicated by the alt-construct which is used to specify design choices that are potential in the sense that one of the choice alternatives can be removed during refinement. In the current example, the design choice is specified because it is not initially known whether or not ClientR will use the same protocol as ClientD.

The specification of the server is shown in Fig. 6. Upon receiving the message storeDoc (resp. getDoc) from the client, it sends the message storeDocument (resp. retrieveDoc) to a data base (not shown) which stores (resp. retrieves) the document. The server, moves into a state in which it waits for an ok message from the data base. If the server receives any message from the clients while waiting for the data base, then these messages are simply dropped. As indicated by the alt-construct, the protocol used in order to transmit and receive messages to the researcher client is an open design choice.

The denotational semantics of standard state machines is usually given as the set of traces that is obtained by recording the events of all paths in the state machine when starting from the initial state. However, we distinguish between two kinds of choice, and interpret state machines as sets of trace sets called *obligations*. Intuitively, the traces of an obligation are equivalent in the sense that an implementation is only required to produce one of them. Thus, each trace of an obligation represent underspecification, or potential choices. However, an implementation is required to produce at least one trace in each obligation. Thus obligations represent explicit choices that have to be present in an implementation.

The obligations described by a basic state machine P is denoted by $[\![P]\!]$ (see App. A.3 for a formal definition). Choices that are not potential will result in new obligations being created, whereas potential choices will result in obligations being collapsed. For instance, the semantics of ClientD is

$$[\![\mathsf{ClientD}]\!] \quad = \quad \{\{\langle\rangle\}, \{\langle ?u.i, !s.p1\rangle\}, \{\langle ?u.i, !s.p1, ?u.i, !s.p1\rangle\}, \ldots\}$$

Here a new obligation is created for each finite iteration of the loop in ClientD. Each obligation consists of a single trace since there are no potential choices in ClientD. Note that state machine names and signal names have been shortened, and that expressions of the form $?nm.si$ and $!nm.si$ represent input and output events, respectively.

The semantics of ClientR is given by

$$\begin{aligned}[\![\mathsf{ClientR}]\!] \quad = \quad & \{\{\langle\rangle\}, \{\langle ?u.i, !s.p1\rangle, \langle ?u.i, !s.p2\rangle\}, \{\langle ?u.i, !s.p1, ?u.i, !s.p1\rangle, \\ & \langle ?u.i, !s.p2, ?u.i, !s.p2\rangle, \ldots\}, \ldots\}\end{aligned}$$

Contrary to ClientD, the obligations of ClientR are not singleton sets because ClientR contains potential choices. For instance, in obligation $\{\langle ?u.i, !s.p1\rangle, \langle ?u.i, !s.p2\rangle\}$, the traces $\langle ?u.i, !s.p1\rangle$ and $\langle ?u.i, !s.p2\rangle$ represent potential choices. Note that $p1$ and $p2$ for the time being are just syntactic suffixes of events. It is first when we make use of event libraries that they come into play.

We assume that basic state machines are autonomous, and therefore executed in parallel. Semantically, parallel composition is defined by *interleaving* traces. For instance, if the two traces $\langle e_1, e_2\rangle$ and $\langle e_1', e_2'\rangle$ describe the execution of two different independent state machines, then the parallel composition of these traces, written $\parallel (\langle e_1, e_2\rangle, \langle e_1', e_2'\rangle)$, is given by

$$\begin{aligned}&\{\langle e_1, e_2, e_1', e_2'\rangle, \langle e_1', e_2', e_1, e_2\rangle, \langle e_1, e_1', e_2, e_2'\rangle, \\ &\langle e_1', e_1, e_2', e_2\rangle, \langle e_1', e_1, e_2, e_2'\rangle, \langle e_1, e_1', e_2', e_2\rangle\}\end{aligned}$$

Each trace describes a possible execution of the two state machines that the traces $\langle e_1, e_2\rangle$ and $\langle e_1', e_2'\rangle$ belong to. Because the state machines are independent and autonomous, all interleavings of the two traces are possible executions.

Basic state machines are usually not completely independent because a state machine that expects an input message from another state machine cannot proceed with execution before the message is received. Hence the execution of one state machine may be influenced by another. Traces that describe communication in which messages are sent before they are received are called *well formed*. We require that all traces in the semantics of a composite state machine to be well formed.

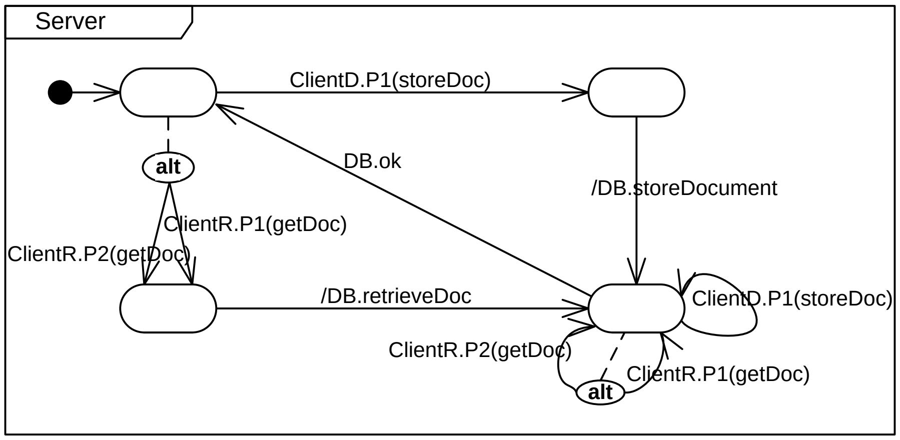

Figure 6: Specification of the server

The semantics of the PM system is the parallel composition of ClientD, Server (as specified in Fig. 6), and ClientR. A fragment of this semantics is shown below

$$\{\{\langle\rangle\}, \{\langle ?u.i, !s.p1, ?cd.p1, !db.s, ?db.o\rangle\},$$
$$\{\langle ?u.i, !s.p1, ?u.i, !s.p2, ?cd.p1, !db.s, !db.o, ?cr.p2, !db.s, ?db.o\rangle, \ldots\}, \ldots\}$$

Note for instance that the trace $\langle ?u.i, !s.p1, ?cr.p2, !db.s, ?db.o\rangle$ is not well formed because the input event $?cdr.p2$ is not preceded by a corresponding output event in the trace. Note also that the behavior of the user and the data base are not shown. We assume that the PM system is composed with a user and data base that generate all possible inputs to the PM system.

6 Step IV: Verify security adherence

In step IV, the user verifies that the specification adheres to the selected security property. Our method describes what it means for a specification to adhere to a security property, but leaves the choice of evaluation process to the user. The evaluation may be formal, e.g., by theorem proving or by model checking if feasible, or it may be by other means like testing if the specifications are too large for full verification. In the following, we argue only informally that our system specification adheres to the NF property.

In the definition of the NF property given in Sect. 4 (see (3)), we considered a system given as a trace set. Since we interpret specifications as sets of obligations, the definition needs to be revisited to take this into account. Instead of requiring that there is a trace u which prevents the low level user from deducing that confidential behavior has occurred, we require that there is an obligation ϕ such that all its traces prevent the low level user from deducing that confidential behavior has occurred. This means that the obligations, as opposed to the individual traces, provide the unpredictability required by the security property. This is in line with the intuition behind the use of obligations. Formally, we have

$$\text{NF}([\![P]\!]) \overset{\text{def}}{=} \forall t \in \widehat{[\![P]\!]} : \exists \phi \in [\![P]\!] : \forall u \in \phi : \neg C(u) \wedge u \sim_{\mathfrak{l}} t \tag{4}$$

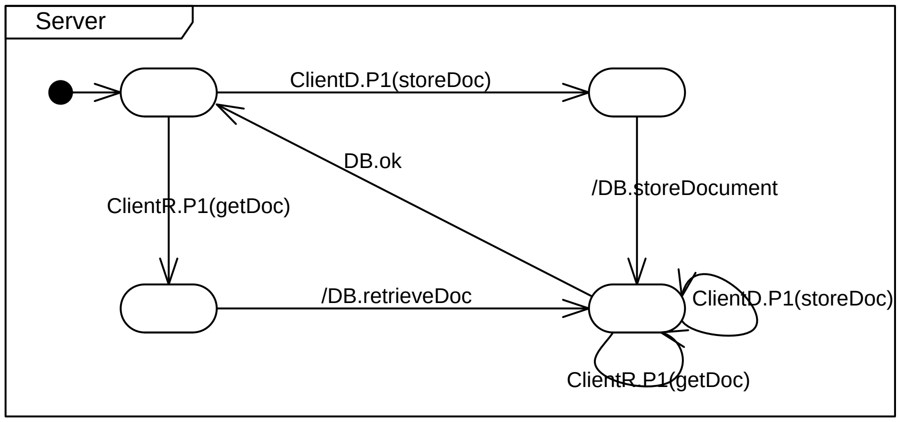

Figure 7: Refined specification of the server

Note that $\widehat{\Omega}$, where Ω is a set of obligations, collapses Ω into a set of traces.

Again, this definition of the NF property may be expressed as an instance of a security property schema which is defined for specifications that are interpreted as sets of obligations [33].

The PM system is secure w.r.t. the NF property because the developer can never be sure that the researcher has done something regardless of the low level observations he can make. There are two reasons for this. First it is always possible for the researchers to do nothing. Second, the developers cannot influence the behavior of the researcher and vice versa because neither the developers nor the researchers receive input from the server.

To see this, note that the semantics of ClientR includes the obligation $\{\langle\rangle\}$. By composing this obligation in parallel with the obligations of ClientD (which describe the observations that the developers can make), and Server, we obtain obligations whose traces do not contain any high level events, i.e., message transmission to or from ClientR. Therefore, for every observation that the developer can make, he can never be sure that the researchers have done something.

7 Step V: Refine system components

In Step V, the user refines the specification by removing alternative design choices until all choices are resolved. The security property selected in step II (defined by (Eq. (4))) is preserved under refinement because it is defined in a semantic model which distinguishes between potential choices and inherent choices that should be provided by a system.

Recall that in our running example, it was not initially known whether or not the researcher would use the same communication protocol as the developers for communicating with the server. Assume that it is decided that the researcher clients will use the same protocol as the developer clients. We then refine the specifications of the previous section by removing all the choice alternatives that involve protocol 2. Fig. 7 shows the resulting specification for the server. The same process has to be carried out for the researcher client.

We now make precise what is meant by refinement. Intuitively, obligations are seen as providing alternative choices which must be provided by an implementation, whereas each trace in an obligation provides a design choice that may be removed under refinement. Formally, a state machine Q is a refinement of a state machine P, written $P \rightsquigarrow Q$, iff

$$(\forall \phi \in [\![P]\!] : \exists \phi' \in [\![Q]\!] : \phi' \subseteq \phi) \wedge (\forall \phi' \in [\![Q]\!] : \exists \phi \in [\![P]\!] : \phi' \subseteq \phi) \qquad (5)$$

The reason why Nf property is preserved under this notion of refinement is that it is defined (see (4)) such that *obligations* (as opposed to traces which are used in most standard definitions) may be seen as providing the unpredictability required by the security property. Since choices provided by an obligation cannot be removed under refinement to an implementable specification, this means that the required unpredictability is preserved. Note that a specification whose semantics includes an empty set is not considered implementable. These kinds of specifications may therefore safely be ignored.

In [33] we show that any security property expressed in the security property schema is preserved under refinement.

8 Step VI: Event transformation

In this step VI, the specification (call it abstract) obtained in step V is transformed into a concrete one by replacing the events of the abstract specification by the specification of these events in the event library. At this stage, all design choices are assumed to be decided. In the following we therefore interpret a specification in the standard way as a trace set (see App. A for a formal definition).

An event library is a set of *event specifications*. An event specification consists of an *event definition* and a *state machine pattern* defining the behavior of the event. An event definition is a pair consisting of a kind (! or ?) and a signal pattern of the form $si(fp_1, \ldots, fp_n)$ where $fp_1, \ldots, fp_n$ are formal parameters. A state machine pattern is a state machine that may contain signal patterns.

In the running example, it is assumed that the event specifications for P1 (which takes care of message transmission) are already specified in the library. The specifications of these events are shown in Fig. 8. The figure shows two specifications, one for the receive event (the signal name which is marked by ?) and one for the send event (the signal name which is marked by !). Both specifications have one formal parameter called *msg*.

The P1 protocol works as follows. After a message has been transmitted, the sender starts a timer, and waits for an acknowledgement (i.e., an ack message) from the receiver. If an acknowledgement is not received within a certain time frame, then a timeout is triggered (the full details of this is not shown), and the sender retransmits the message. If an acknowledgement has not been received after the message has been transmitted 10 times, then the sender gives up. Note that the label *to* in the output event specification (the topmost specification of Fig. 8) and the label *from* (the lowermost specification of Fig. 8) are parameters that are bound to the transmitter and the receiver of the event being replaced, respectively. For instance, Fig. 9 shows the result of replacing the events containing P1 in the upper most specification of Fig. 5 by the event specification of Fig. 8. Here we see that the *to* parameter has been bound to Server which is the recipient of the message.

We let T_{EL} be the function that takes a basic state machine P and yields the state machine $T_{EL}(P)$ obtained from P by replacing the events occurring in each action expression of P by their definition in event library EL. For instance, if EL is the event library of Fig 8, and ClientD is the topmost specification of Fig. 5, then $T_{EL}(\mathsf{ClientD})$ yields the specification shown in Fig. 9.

To ensure that transformations preserve semantic equality, i.e., that two abstract specifications that are (semantically) equal are not transformed into two (semantically) different concrete specifications, we require all variables in event specifications to be local. To ensure this, we let T_{EL} rename variables of the event specification such that no name clashes occur when events are being replaced by their definitions. For instance, if the topmost specification of Fig. 8 is applied in a context where the variable c is used, then this variable is given a fresh name which is not used in the context.

In App. B, we show that any event transformation T_{EL} preserve semantic equality, i.e.,

$$[\![P]\!] = [\![Q]\!] \implies [\![T_{EL}(P)]\!] = [\![T_{EL}(Q)]\!]$$

It is also shown that an event transformation is entirely characterized by its behavior on events. Technically, this means that the transformation is homomorphic w.r.t. the union and concatenation of trace sets.

The transformation T_{EL} is defined for *basic* state machines. The transformation induced by an event library EL which takes a *composite* state machine P as input, is simply the function that applies T_{EL} to all basic state machines that P consists of.

The transformation of a composite state machine may produce concrete traces that have no abstract equivalent due to the granularity we get when events are substituted by state machines. We say that the *image* of a transformation T_{EL}, written Im_{EL}, is the set of all concrete traces that have an abstract equivalent. In App. B.2, we show that any event transformation T^C_{EL} for composite state machines preserve semantic equality when restricting attention to the image, i.e.,

$$[\![P]\!] = [\![Q]\!] \implies ([\![T^C_{EL}(P)]\!] \cap Im_{EL}) = ([\![T^C_{EL}(Q)]\!] \cap Im_{EL})$$

It is also shown that the transformation T^C_{EL} for composite state machines is homomorphic w.r.t. union of trace sets when restricting attention to the image.

9 Step VII: Verify adherence preservation

In step VII, the user verifies that the transformation performed in step VI is adherence preserving. Again, we do not consider any particular method of evaluation (e.g., model checking), instead we present the conditions that need to be evaluated.

We say that a transformation preserves a security property $SecP$ if the image of every abstract state machine P that is secure w.r.t. $SecP$ is secure w.r.t. $SecP$, i.e.,

$$SecP([\![P]\!]) \implies SecP([\![T_{EL}(P)]\!] \cap Im_{EL}) \tag{6}$$

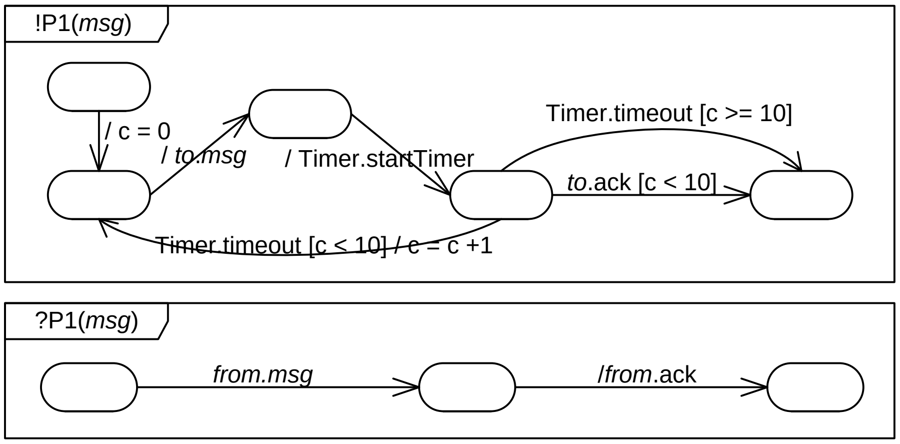

Figure 8: Specification of the P1 event

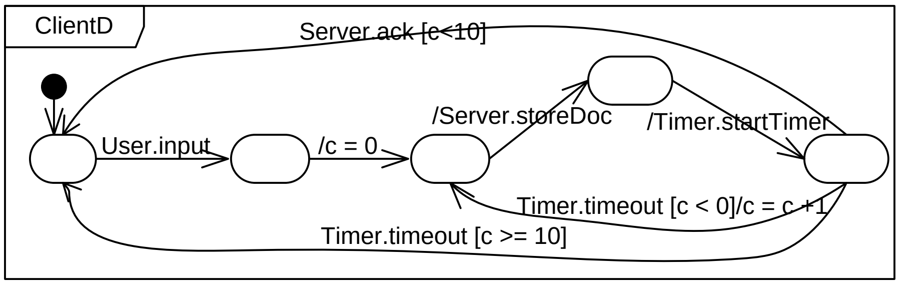

Figure 9: Concrete specification of the developer client

We restrict attention to image since we cannot exploit the fact that the abstract specification is secure to ensure that concrete traces that do not have any abstract equivalent do not violate security. This means that additional security analysis may be needed at the concrete level to ensure that those traces that do not have any abstract equivalent do not violate security. However, it should not be necessary to recheck the adherence relationship already established at the more abstract level.

In App. B.3, we define a general condition under which an event transformation preserves a given security property. To obtain the specific condition under which the event library of step VII preserves the NF property, the general condition must be instantiated by the definition of the NF property. We then obtain the following condition:

(a) if an abstract trace contains no high level events, then no concrete traces it is transformed to can contain any high level events;

(b) if two abstract traces are low level equivalent, then no the concrete traces they are transformed to must also be low level equivalent.

Condition (a) is satisfied because the set of high level events is assumed to be all events that can occur in ClientR, and because event transformations cannot change the transmitters or receivers of events.

To check whether condition (b) is satisfied, we may check whether the parallel composition of all corresponding state machine patterns of event specifications in EL (i.e., one send and one receive event) yields the same trace set regardless of context. In the current example, the event library of Fig. 8 satisfies this condition. For instance, the parallel composition of the two event specifications in Fig. 8 when substituted for event $!s.p1$ (in ClientD of Fig .5) and $?cd.p1$ (in Server of Fig. 6), is

$$\{\langle !s.sd, ?cd.sd, !cd.a, ?s.a\rangle, \langle !s.sd, !s.sd, ?cd.sd, !cd.a, ?s.a\rangle, \dots\}$$

Since none of the traces in this set contains a context dependent input event, all traces are well formed regardless of the context they appear in.

10 Related work

This report is related to previous work by the authors [33, 34]. In particular, the security property schema (referred to in Sect. 4) was introduced in [34, 33]. There it was also shown that all security properties of the schema are preserved under refinement. The main difference between this report and [33, 34], is that [33] does not consider transformations at all, and [34] is mostly semantics based; specifications are treated as trace sets, or sets of trace sets. Thus state machines are not considered at all, and a rigorous characterization of syntactic transformations and their interpretation is not given. The main contribution of this report is the definition of UML inspired state machines (Sect. 5 and App. A) and the definition of event transformations (Sect. 8 and App. B).

The state machine notation we use is inspired by UML. Several approaches have given UML state machines a formal semantics (see e.g., [1, 20, 29]). However, none of the works we are aware of distinguishes between explicit and potential nondeterminism.

The semantics of state machines that we propose is based on STAIRS [12], and our notion of refinement corresponds to so-called limited refinement in STAIRS. The main difference between our work and STAIRS is that STAIRS gives a semantics for UML sequence diagrams, whereas we consider state machines. Another difference is that the STAIRS denotational semantics with data [32] describes traces that will never be produced during execution because the values of the data states are allowed to change at any time between assignments. In our denotational semantics, we do not allow this. In this sense, our denotational semantics is more similar to the operational semantics of STAIRS [24].

Our work is related to two somewhat distinct areas of research: model-driven security (MDS) and information flow security. Already published approaches to MDS include [2, 3, 5, 8, 10, 14, 22, 37]. Although interesting, they differ from our work in that they are in most cases of a rather informal nature; the semantics of the languages used (at abstract or/and concrete levels) are not sufficiently precise to allow for rigorous reasoning at more than one level of abstraction. Moreover, some of the approaches ([2, 3, 8, 22]) consider transformation of security requirements only, and not transformation of system specifications. These approaches allow adherence checking only at the lowest level of abstraction. Furthermore, others ([5, 8, 10, 37]) do not clearly characterize what it means for a system to adhere to a security requirement. Instead, security is described in terms of a *security mechanism*.

Information flow security was originally introduced [35] as a generalization of the so-called *non-interference* property [11] from deterministic to nondeterministic systems. Since then, a number of different information flow properties (see e.g., [27, 30, 38]), as well general information flow frameworks (see e.g., [4, 25, 28, 31, 38]), have been proposed in various semantic models. The security schema considered in this report (App. B.3) is inspired by the framework of [25]. One of the main differences between the frameworks is that the distinction of potential and explicit choice is not made in [25].

Preservation of secure information flow properties under refinement was first considered in [16]. There it was shown that information flow properties are not in general preserved under a notion of refinement based on inverse trace set inclusion. It has later been observed (see e.g., [13, 17, 18, 31]) that the problem occurs in approaches that do not take the distinction of potential and explicit nondeterminism into account. All of the above citations consider a less general notion of refinement than we do.

We are not aware of any work which consider preservation of information flow properties under a syntactic notion of transformation.

11 Conclusions and future work

We have presented a method for model-driven information flow security. The method has 7 main steps that accommodate the integration of security into design artifacts above the code level. In particular, the method supports (1) specification of systems at different levels of abstraction; (2) rigorous adherence verification at different levels of abstraction; (3) adherence preserving refinement and transformation.

The method is based on UML inspired state machines that we have extended with a construct for specifying potential choice. These state machines, as well as event transformations from abstract to concrete state machines have been given formal syntax and semantics. We have shown that the event transformations satisfy properties that simplify the conditions under which adherence is preserved.

This report is related to previous work of the authors [34, 33]. However, state machines were not considered in [34, 33], and a formal definition of syntactic transformations and their interpretation was not given. The main contribution of this report is the definition of UML inspired state machines (Sect. 5 and App. A) and the definition of event transformations (Sect. 8 and App. B).

Many approaches to model-driven security exist, but all of them are of a rather informal nature. This makes it difficult to precisely define the meaning of adherence or preservation of adherence under refinement and transformation.

We are not aware of any work that address the preservation of information flow security properties under a syntactic notion of transformation.

In future work, we would like to develop syntactic type checking rules which can be used by a tool to automatically check preservation of adherence under transformations.

Acknowledgement The research of this report has partly been funded by the Research Council of Norway project SECURIS (152839/220) and DIGIT (180052/S10), and partly by the EU through the S^{3}MS (contract no. 27004) project.

References

[1] D. B. Aredo. Semantics of UML Statecharts in PVS. In *Proc. of the 7th International Multi-conference on Systemics, Cybernetics and Informatics (SCI2003)*, 2003.

[2] D. Basin, J. Doser, and T. Lodderstedt. Model driven security for process-oriented systems. In *Proc. of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT'03)*, pages 100–109. ACM, 2003.

[3] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering Methodologies*, 15(1):39–91, 2006.

[4] A. Bossi, R. Focardi, D. Macedonio, C. Piazza, and S. Rossi. Unwinding in information flow security. *Electronic Notes in Theoretical Computer Science*, 99:127–154, 2004.

[5] R. Breu, M. Hafner, B. Weber, and A. Novak. Model driven security for inter-organizational workflows in e-government. In *Proc. of the 2005 International Conference on E-Government: Towards Electronic Democracy (TCGOV'05)*, volume 3416 of *Lecture Notes in Computer Science*, pages 122–133. Springer, 2005.

[6] M. Broy and K. Stølen. *Specification and development of interactive systems. FOCUS on streams, interface, and refinement.* Springer, 2001.

[7] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[8] C. C. Burt, B. R. Bryant, R. R. Raje, A. M. Olson, and M. Auguston. Model driven security: unification of authorization models for fine-grain access control. In *Proc. of the 7th International Enterprise Distributed Object Computing Conference (EDOC'03)*, pages 159–173. IEEE Computer Society, 2003.

[9] CSI/FBI. *Computer Crime and Security Survey*, 2005.

[10] E. Fernández-Medina and M. Piattini. Extending OCL for secure database development. In *Proc. of the 7th International Conference on The Unified Modelling Language: Modelling Languages and Applications (UML'04)*, volume 3273 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 2004.

[11] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proc. of the 1982 IEEE Symposium on Security and Privacy (S&P'82)*, pages 11–20. IEEE Computer Society, 1982.

[12] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. STAIRS towards formal design with sequence diagrams. *Journal of Software and Systems Modeling*, 4(4):355–367, 2005.

[13] M. Heisel, A. Pfitzmann, and T. Santen. Confidentiality-preserving refinement. In *Proc. of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 295–306. IEEE Computer Society, 2001.

[14] R. Heldal and F. Hultin. Bridging Model-Based and Language-Based Security. In *Proc. of the 8th European Symposium (ESORICS'03)*, volume 2808 of *Lecture Notes in Computer Science*, pages 235–252. Springer, 2003.

[15] ISO/IEC 17799:2005. *Information technology - security techniques - code of practice for information security management (ISO27002)*, 2005.

[16] J. Jacob. On the derivation of secure components. In *Proc. of the IEEE Symposium on Security and Privacy (S&P'89)*, pages 242–247. IEEE Computer Society, 1989.

[17] J. Jürjens. Secrecy-preserving refinement. In *Proc. of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FFME'01)*, volume 2021 of *Lecture Notes in Computer Science*, pages 135–152. Springer, 2001.

[18] J. Jürjens. *Secure systems development with UML*. Springer, 2005.

[19] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata studies*, pages 3–40, 1956.

[20] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In *Proc. of the 4th International Conference on The Unified Modeling Language (UML'01)*, volume 2185 of *Lecture Notes in Computer Science*, pages 241–256, 2001.

[21] P. Linz. *An introduction to formal languages and automata*. D. C. Heath and Company, 1990.

[22] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In *Proc. of the 5th International Conference on The Unified Modeling Language (UML'02)*, volume 2460 of *Lecture Notes in Computer Science*, pages 426–441. Springer, 2002.

[23] M. S. Lund. *Operational analysis of sequence diagram specifications*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2008.

[24] M. S. Lund and K. Stølen. A Fully General Operational Semantics for UML 2.0 Sequence Diagrams with Potential and Mandatory Choice. In *Proc. of the 14th International Symposium on Formal Methods (FM'06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 380–395. Springer, 2006.

[25] H. Mantel. Possibilistic Definitions of Security - An Assembly Kit. In *Proc. of the IEEE Compuer Security Foundations Workshop (CSFW'00)*, pages 185–199. IEEE Computer Society, 2000.

[26] H. Mantel. Preserving Information Flow Properties under Refinement. In *Proc. of the IEEE Symposium on Security and Privacy (S&P'01)*, pages 78–91. IEEE Computer Society, 2001.

[27] D. McCullough. Specifications for Multi-Level Security and a Hook-Up Property. In *Proc. of the IEEE Symposium on Security and Privacy (S&P'87)*, pages 161–166. IEEE Computer Society, 1987.

[28] J. McLean. A General Theory of Composition for Trace Sets Closed under Selective Interleaving Functions. In *Proc. of the IEEE Symposium on Research in Security and Privacy (S&P'94)*, pages 79–93. IEEE Computer Society, 1994.

[29] J. Merseguer, S. Bernardi, J. Campos, and S. Donatelli. A compositional semantics for UML state machines aimed at performance evaluation. In *Proc. of the 6th International Workshop on Discrete Event Systems*, pages 295–302. IEEE Computer Society Press, 2002.

[30] C. O'Halloran. A calculus for information flow. In *Proc. of the 1st European Symposium on Research in Computer Security (ESORICS'90)*, pages 147–159. AFCET, 1990.

[31] A. W. Roscoe. CSP and determinism in security modelling. In *Proc. of the IEEE Symposium on Security and Privacy (S&P'95)*, pages 114–127. IEEE Computer Society, 1995.

[32] R. K. Runde and K. Stølen. Refining UML Interactions with Underspecification and Nondeterminism. Research Report 325, Department of Informatics, University of Oslo, 2007.

[33] F. Seehusen, B. Solhaug, and K. Stølen. Adherence preserving refinement of trace-set properties in STAIRS: exemplified for information flow properties and policies. *Journal of Software and Systems Modeling*, 8(1):45–65, 2009.

[34] F. Seehusen and K. Stølen. Secure Information Flow, Abstraction, and Composition. To appear in Journal of IET Information Security, 2009.

[35] D. Sutherland. A model of information. In *Proc. of the 9th National Computer Security Conference*, pages 175–183, 1986.

[36] R. J. van Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4/5):229–327, 2001.

[37] B. Vela, E. Fernández-Medina, E. Marcos, and M. Piattini. Model driven development of secure XML databases. *SIGMOD Record*, 35(3):22–27, 2006.

[38] A. Zakinthinos and E. S. Lee. A general theory of security properties. In *Proc. of the IEEE Computer Society Symposium on Research in Security and Privacy (S&P'97)*, pages 94–102. IEEE Computer Society, 1997.

A State machines

In this section, we present the syntax and the semantics of our UML inspired state machines.

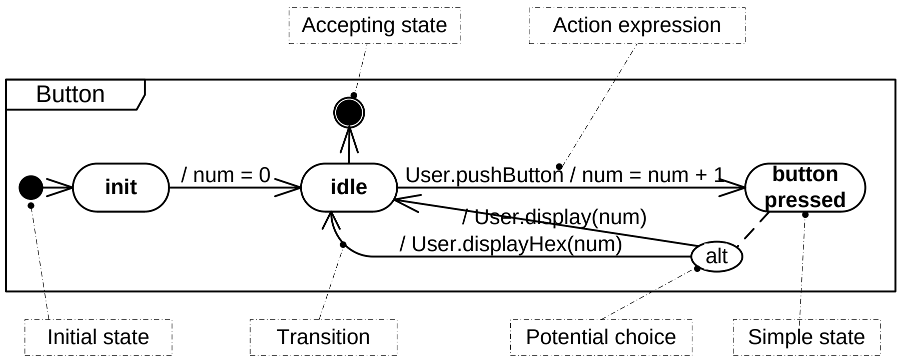

Figure 10: Example of a basic state machine

A.1 Syntax

A state machine is either *basic* in the sense that it does not consist of other state machines or *composite* in the sense that it does consist of other state machines. We first present the syntax of basic state machines, then we define the syntax of composite state machines.

A.1.1 Basic state machines

In this section, we first present the graphical syntax of basic UML inspired state machines. Then we define a textual syntax.

We make use of the following syntactic categories:

$$\begin{array}{lcll} ax & \in & \mathbf{AExp} & \text{arithmetic expressions} \\ bx & \in & \mathbf{BExp} & \text{boolean expressions} \\ sx & \in & \mathbf{SExp} & \text{string expressions} \end{array}$$

Here ax, bx, and sx are syntactic variables of the three syntactic categories, respectively.

We let **Exp** denote the set of all arithmetic, boolean, and string expressions, and we let ex range over this set. Moreover, we assume that the following sets are given:

$$\begin{array}{lcll} x, y & \in & \mathbf{Var} & \text{variables} \\ n & \in & \mathbf{Num} & \text{numerals} \\ st & \in & \mathbf{Str} & \text{strings} \end{array}$$

We let **Val** denote the set of all values, i.e., numerals, strings, and booleans (`t` or `f`).

As illustrated in Fig. 10, the constructs which are used for specifying state machines are *initial state*, *simple state*, *accepting state*, *transition*, *action expression*, and *potential choice*.

The state machine illustrated in Fig. 10 specifies a button that whenever pressed, displays the number of times it has been pressed to a user. The number is displayed by sending the signal $display(num)$ or $displayHex(num)$ (where num represents the number of times the button has been pressed) to the user. It is intended that $display(num)$ is used for displaying the number in base 10, and that $displayHex(num)$ is used for displaying the number in base 16 (hexadecimal). Whether the button sends the signal $display(num)$ or the signal $displayHex(num)$ should be interpreted as a design choice as specified by the potential choice construct (alt) attached to the branching transitions.

A state describes a period of time during the life of component. The three kinds of states, *initial state*, *simple state*, and *accepting state*, are graphically represented by a black circle, a box with rounded edges, and a black circle encapsulated by another circle, respectively.

A *transition* represents a move from one state to another. Transitions are labeled by *action expressions* of the form

$$nm.si[bx]/ef$$

Here the expression $nm.si$, where nm is a state machine name and si is a signal, is called a *trigger*. The expression $[bx]$ where bx is a boolean expression is called a *guard*, and ef is called an *effect*. Intuitively, the action should be understood as follows: when signal si is received from a state machine with name nm and the boolean expression bx evaluates to true, then the effect ef is executed. An effect is either an assignment or an output expression of the form $nm.si$ representing the transmission of signal si to the state machine with name nm.

An assignment is a pair (x, ex) consisting of a variable x and an expression ex. The set of all assignments is defined by

$$\mathbf{Assign} \stackrel{\mathsf{def}}{=} \mathbf{Var} \times \mathbf{Exp}$$

In diagrams, assignments are written $x = ex$ instead of (x, ex).

We let **ActExp** denote the set of all action expressions and we let $actx$ range over this set. Triggers, guards, and effects are all optional parts of an action expression. We sometimes indicate the absence of these parts by ϵ, or we omit to write them all together. For instance, the expressions $\epsilon[bx]/ef$ and $[bx]/ef$ are valid and equal action expressions without any trigger.

We let **Nm** denote the set of all state machine names and we let nm range over this set. We represent names by strings, thus we have $\mathbf{Nm} \subseteq \mathbf{Str}$. A signal is a tuple $(st, ex_1, \ldots, ex_n)$ where st denotes the signal name, and $ex_1, \ldots, ex_n$ are the parameters of the signal. We usually write $st(ex_1, ex_2, ..., ex_n)$ instead of $(st, ex_1, ex_2, ..., ex_n)$. Formally, the set of all signals is defined

$$\mathbf{SI} \stackrel{\mathsf{def}}{=} \mathbf{Str} \times \mathbf{Exp}^*$$

The *potential choice* construct (graphically represented by an alt) may be attached to branching transitions in a state machine specification to specify alternative design choices.

The graphical notation used for specifying state machines is essentially a subset of the UML statechart notation. However, there are two differences:

- We allow the sender of an input signal to be specified on the transition labels. UML does not have any particular construct for specifying this.
- The potential choice construct is not part of the UML statechart notation.

In theory, the former difference could have been avoided by letting the name of the sender of an input signal be part of the name of the signal. However, by definition of event transformations, we would then have to specify one event definition for each possible sender or receiver of a signal. This would be impractical. To see this, consider for instance the example of Sect. 8, where the state machine of Fig. 9 is obtained by replacing all signals whose name is $P1$ in Fig. 5 according to the event definition of Fig. 8. Since the name of the sender or receiver of signals is not part of the signal name, we only have to specify two event definitions: one for the transmission of $P1$ and one for the reception of $P1$. However, if the sender or receiver names had been part of the signal name, then we would have to specify one event definition for each possible sender or receiver. This is clearly not a practical solution.

Note that we do not consider our manner of specifying transmission of signals in state machines a deviation from the UML standard. The reason for this is that the standard does not impose any particular restriction on the kind of effects that can be specified or the language they are specified in.

Having defined the graphical syntax of basic state machines, we now move on to define the textual syntax of basic state machines.

Definition A.1 (Basic state machine expression) *The set of all basic state machine expressions* $\mathbf{P}$ *is defined by the following grammar*

$$P \quad ::= \quad act \,|\, P^* \,|\, P.P \,|\, P+P \,|\, P|P$$

The base cases implies that any action (act) is a basic state machine expression. Any other state machine expression is constructed from the basic ones through the application of the operators for iteration (P^*), sequential composition ($P.P$), standard choice ($P+P$), and potential choice ($P|P$).

To define the semantics more conveniently, we do not represent actions exactly as they are represented in the graphical diagrams. Instead, we define the set of all actions $\mathbf{Act}$ by

$$\mathbf{Act} \stackrel{\mathsf{def}}{=} \mathbf{IE} \cup \{\epsilon\} \times \mathbf{BExp} \cup \{\epsilon\} \times \mathbf{OE} \cup \{\epsilon\} \times \mathbf{Assign} \cup \{\epsilon\} \tag{7}$$

where $\mathbf{IE}$ and $\mathbf{OE}$ denote the set of all input events and output events, respectively. We require that every action must contain at most one event, i.e.,

$$(ie_\epsilon, bx_\epsilon, oe_\epsilon, a_\epsilon) \in \mathbf{Act} \implies (ie_\epsilon = \epsilon \vee oe_\epsilon = \epsilon) \tag{8}$$

Here we have used ie_ϵ to denote an ϵ or an input event, i.e., $ie_\epsilon \in \mathbf{IE} \cup \{\epsilon\}$. The same convention is used for boolean expressions, output events, and assignments.

The sets of all input events and output events are defined by

$$\mathbf{IE} \stackrel{\mathsf{def}}{=} \{?\} \times \mathbf{M} \qquad\qquad \mathbf{OE} \stackrel{\mathsf{def}}{=} \{!\} \times \mathbf{M}$$

where $\mathbf{M}$ denotes the set of all messages. An event of the form $(!, m)$ is an output event representing a transmission of message m, whereas an event of the form $(?, m)$ is an input event representing a reception of m. We let $\mathbf{E}$ denote the set of all events, i.e.,

$$\mathbf{E} \stackrel{\mathsf{def}}{=} \mathbf{IE} \cup \mathbf{OE}$$

Messages are of the form (nm_t, nm_r, si), where nm_t is the name of the transmitter state machine of the message, nm_r is the name of the receiver state machine of the message, and si is the signal of the message. The set of all messages is thus defined by

$$\mathbf{M} \stackrel{\mathsf{def}}{=} \mathbf{Nm} \times \mathbf{Nm} \times \mathbf{SI}$$

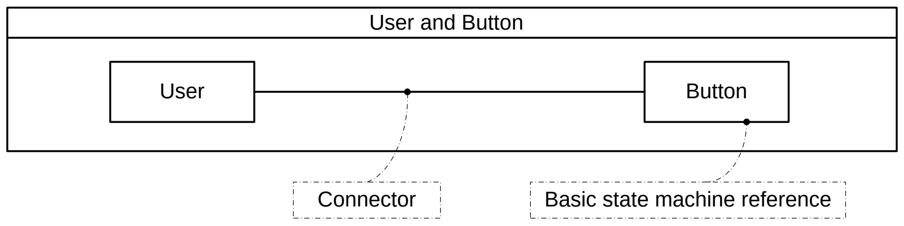

Figure 11: Example of a composite state machine

We require that the transmitters of output events and receivers of input events in a basic state machine must have the same name. To make this precise, we define the name of an event by the function $nm \in \mathbf{E} \to \mathbf{Nm}$ as follows

$$\begin{aligned} nm((!, (nm_t, nm_r, si))) &\overset{\mathsf{def}}{=} nm_t \\ nm((?, (nm_t, nm_r, si))) &\overset{\mathsf{def}}{=} nm_r \end{aligned}$$

We let $\mathbf{E}_{nm}$ denote the set of all events with name nm, i.e.,

$$\mathbf{E}_{nm} \overset{\mathsf{def}}{=} \{e \in \mathbf{E} \mid nm(e) = nm\} \tag{9}$$

We let $nm \in \mathbf{P} \to \mathbf{Nm} \cup \{\bot\}$ be the function that yields the name of a state machine expression. The function is defined such that $nm(P) = nm$ if all events in the actions of P are members of $\mathbf{E}_{nm}$ for some name nm. Otherwise $nm(P) = \bot$.

A basic state machine expression (when we ignore the potential choice) is essentially a regular expression. It is well know that the language of regular expressions is equal to the language defined by a finite state machine [19]. Several procedures exist for deriving a regular expression from a finite state machine (see, e.g., [7, 19, 21]). Although state machine expressions have two kinds of choices (instead of just one as in regular expressions), similar techniques could be used to convert a graphical state machine into a textual state machine expression. Of course, we have to translate action expressions of graphical diagrams to actions of textual state machine expressions, but this is trivial.

As an example, the state machine of Fig. 10 is represented by the following state machine expression

$$\begin{gathered} (\epsilon, \epsilon, \epsilon, num = 0) \,.\, ((User, Button, pushButton), \epsilon, \epsilon, num = num + 1) \,. \\ ((\epsilon, \epsilon, (Button, User, display(num)), \epsilon) | (\epsilon, \epsilon, (Button, User, displayHex(num)))^* \end{gathered}$$

A.1.2 Composite state machines

A composite state machine is a state machine that consists of one or more (basic) state machines. There are two constructs for specifying composite state machines: *connector* and *basic state machine reference*. The connector is used to indicate that the basic state machines it is connected to may communicate with each other. Connectors are graphically drawn as lines. The basic state machine reference construct is a reference to the basic state machine whose name equals the name of the reference.

An example of a composite state machine is shown in Fig. 11. The composite state machine consists of two basic state machines ($User$ and $Button$) and one connector indicating that $User$ and $Button$ may communicate with each other.

The notation we use for specifying composite state machines is a subset of the UML *composite structure* notation.

We are now ready to define the textual syntax of composite state machine expressions.

Definition A.2 (Composite state machine expression) *The set of all syntactically correct composite state machine expressions* $\mathbf{P}^C$ *is defined by the following syntax*

$$P ::= P_1 \parallel \cdots \parallel P_n$$

where $P_1, \ldots, P_n$ *are basic state machine expressions with single distinct names i.e.,* $nm(P_i) \neq \perp$ *for all* $i \in \{1, \ldots, n\}$*, and* $nm(P_i) \neq nm(P_j)$ *if* $i \neq j$ *for all* $i, j \in \{1, \ldots, n\}$*.*

A.2 Semantics of state machines without potential choice

In this section, we define the semantics of state machines that do not contain the potential choice construct.

A.2.1 Basic state machines

In this section, we define the semantics of basic state machine expressions. Roughly speaking, the semantics of a state machine is the set of sequences obtained by recording all input and output events produced in each possible execution of the state machine. To obtain only those sequences that can be produced during execution, we need a way of evaluating expressions, and a way of storing the variables that are assigned to values during execution. The following auxiliary functions are needed to define this more precisely. The definitions of these functions are inspired by [23].

We let $var \in \mathbf{Exp} \to \mathbb{P}(\mathbf{Var})$ be the function that yields the set of all variables in an expression. An expression $ex \in \mathbf{Exp}$ is closed if $var(ex) = \emptyset$. We let **CExp** denote the set of closed expressions, defined as:

$$\mathbf{CExp} \overset{\mathsf{def}}{=} \{ex \in \mathbf{Exp} \mid var(ex) = \emptyset\}$$

We assume the existence of a function $eval(\_) \in \mathbf{CExp} \to \mathbf{Val}$ that evaluates any closed expression to a value. In the obvious way, we lift $eval$ to signals containing closed expressions as arguments and to events that contain such signals. In addition, we lift $eval$ to actions as follows

$$\begin{array}{lcl} eval((ie_\epsilon, bx_\epsilon, oe_\epsilon, \epsilon)) & \overset{\mathsf{def}}{=} & (eval(ie_\epsilon), eval(bx_\epsilon), eval(oe_\epsilon), \epsilon) \\ eval((ie_\epsilon, bx_\epsilon, oe_\epsilon, (x, ex))) & \overset{\mathsf{def}}{=} & (eval(ie_\epsilon), eval(bx_\epsilon), eval(oe_\epsilon), (x, eval(ex))) \end{array}$$

Let $\sigma \in \mathbf{Var} \to \mathbf{Exp}$ be a (total) mapping from variables to expressions. We denote such a mapping $\sigma = \{x_1 \mapsto ex_1, x_2 \mapsto ex_2, \ldots\}$ for distinct $x_1, x_2, \dots \in \mathbf{Var}$ and for $ex_1, ex_2, \dots \in \mathbf{Exp}$. If $ex_1, ex_2, \dots \in \mathbf{Val}$ we call it a data state. We let Σ denote the set of all mappings and $\widehat{\Sigma}$ denote the set of all data states.

We let $\sigma[x \mapsto ex]$ be the mapping σ except that it maps x to ex, i.e.,

$$\begin{array}{lcl} \{x_1 \mapsto ex_1, x_2 \mapsto ex_2, \ldots\}[x \mapsto ex] & \overset{\mathsf{def}}{=} & \{x_1 \mapsto ex_1, \ldots, x_i \mapsto ex, \ldots\} \\ & & \quad \text{where } x = x_i \text{ for some } i \in \{1, \ldots, n\} \end{array}$$

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

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

We let $\sigma(x)$ denote the expression that x maps to. 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 σ. We lift σ to signals and events in the same way, and to actions as follows

$$\begin{array}{lcl}
\sigma((ie_\epsilon, bx_\epsilon, oe_\epsilon, \epsilon)) & \stackrel{\text{def}}{=} & (\sigma(ie_\epsilon), \sigma(bx_\epsilon), \sigma(oe_\epsilon), \epsilon) \\
\sigma((ie_\epsilon, bx_\epsilon, oe_\epsilon, (x, ex))) & \stackrel{\text{def}}{=} & (\sigma(ie_\epsilon), \sigma(bx_\epsilon), \sigma(oe_\epsilon), (x, \sigma(ex)))
\end{array}$$

To define the semantics of a basic state machine, i.e., the sequences of events that can be produced during execution, it is necessary to record the change in data state. We define an intermediate semantics for this purpose. The intermediate semantics is defined in terms of sequences of so-called action-state triples. Action-state triples are of the form

$$(act, \sigma, \sigma')$$

where σ denotes the so-called *pre-state* before act is executed, and σ' denotes the *post-state* after act has been executed, e.g., if act contains an assignment (x, ex), then σ' is the state σ except that x maps to ex.

To define the intermediate semantics, we make use of the concatenation operator $\frown$. If s and t are sequences, then $s \frown t$ yields the sequence obtained by concatenating s and t. We lift the concatenation operator to sequence sets such that $\Phi \frown \Phi'$ yields the set obtained by concatenating all sequences of Φ with all sequences of Φ'. Also, we denote by Φ^n, the set of sequences obtained by concatenating Φ n times. In particular, Φ^0 is defined by $\{\langle\rangle\}$. For example, we have that Φ^5 yields Φ concatenated five times, i.e.,

$$\Phi^5 = \Phi \frown \Phi \frown \Phi \frown \Phi \frown \Phi$$

We are now ready to define the intermediate semantics of state machines.

Definition A.3 (Intermediate semantics of basic state machines) *The intermediate semantics of a basic state machine expression P, written $(\!| P |\!)$, is defined*

$$\begin{array}{lcl}
(\!| (ie_\epsilon, bx_\epsilon, oe_\epsilon, \epsilon) |\!) & \stackrel{\text{def}}{=} & \{\langle eval(\sigma((ie_\epsilon, bx_\epsilon, oe_\epsilon, \epsilon)), \sigma, \sigma)\rangle \mid \sigma \in \widehat{\Sigma}\} \\
(\!| (ie_\epsilon, bx_\epsilon, oe_\epsilon, (x, ex) |\!) & \stackrel{\text{def}}{=} & \{\langle eval(\sigma((ie_\epsilon, bx_\epsilon, oe_\epsilon, (x, ex)))), \sigma, \\
 & & \quad \sigma[x \mapsto eval(\sigma(ex))])\rangle \mid \sigma \in \widehat{\Sigma}\} \\
(\!| P.Q |\!) & \stackrel{\text{def}}{=} & (\!| P |\!) \frown (\!| Q |\!) \\
(\!| P + Q |\!) & \stackrel{\text{def}}{=} & (\!| P |\!) \cup (\!| Q |\!) \\
(\!| P^* |\!) & \stackrel{\text{def}}{=} & \bigcup_{i \in \mathbb{N}} ((\!| P |\!))^i
\end{array}$$

The intermediate semantics is defined in a modular way in the sense that the meaning of an expression is not affected by the context it appears in. This is achieved by letting actions be evaluated under all possible data states. A consequence of this is that the intermediate semantics describes action sequences that will never be produced during execution. There are two reasons for this:

(a) The sequences of the intermediate semantics may contain actions whose guards are evaluated to false. During execution, however, these actions will never occur.

(b) Because the pre-state of a triple is not required to be equal to the post-state of the triple preceding it, values of the data states may implicitly change at any time between assignments, while this would never occur during execution.

If a sequence s of action-state triples does not exhibit problems (a) and (b), we say that it is *well formed*, and write $wft(s)$ to indicate this. Formally, well formed sequences of action-state triples are inductively defined by the following rules

$$\frac{-}{wft(\langle\rangle)} \qquad \frac{-}{wft(\langle((ie_\epsilon, \mathtt{t}, oe_\epsilon, a_\epsilon), \sigma, \sigma')\rangle)}$$

$$\frac{wft(s \frown \langle(act, \sigma, \sigma')\rangle) \quad wft(\langle(act', \sigma', \sigma'')\rangle)}{wft(s \frown \langle(act, \sigma, \sigma'), (act', \sigma', \sigma'')\rangle)}$$

We let $\mathcal{A}$ denote the set of all well formed action-state triple sequences, i.e.,

$$\mathcal{A} \overset{\text{def}}{=} \{s \in (\mathbf{Act} \times \widehat{\Sigma} \times \widehat{\Sigma})^* \mid wft(s)\} \tag{10}$$

We are only interested in the input-output behavior of state machines. To make this precise we let $io \in (\mathbf{Act} \times \widehat{\Sigma} \times \widehat{\Sigma})^* \to \mathbf{E}^*$ be the function that takes a sequence s of action-state triples and produces the sequence of events obtained from s by removing all data stats, assignments, and guards of s. The io function is formally defined by the following rules:

$$\begin{array}{lcl} io(\langle\rangle) & \overset{\text{def}}{=} & \langle\rangle \\ io(\langle((ie, bx_\epsilon, \epsilon, a_\epsilon), \sigma, \sigma')\rangle \frown t) & \overset{\text{def}}{=} & \langle ie\rangle \frown io(t) \\ io(\langle((\epsilon, bx_\epsilon, oe, a_\epsilon), \sigma, \sigma')\rangle \frown t) & \overset{\text{def}}{=} & \langle oe\rangle \frown io(t) \\ io(\langle((\epsilon, bx_\epsilon, \epsilon, a_\epsilon), \sigma, \sigma')\rangle \frown t) & \overset{\text{def}}{=} & io(t) \end{array} \tag{11}$$

We lift io to sets of action-state triple sequences such that $io(\Phi)$ yields the set of sequences obtained by applying io to each sequence in Φ. For example, we have that

$$\begin{array}{l} io(\{\langle(((?, m), bx, \epsilon, (x, ex)), \sigma_1, \sigma_1'), ((\epsilon, bx', (!, m'), \epsilon), \sigma_2, \sigma_2')\rangle\}) = \\ \quad \{\langle(?, m), (!, m')\rangle\} \end{array}$$

Definition A.4 (Semantics of basic state machines) *The semantics of a basic state machine expression P, written $[\![P]\!]$, is defined*

$$[\![P]\!] \overset{\text{def}}{=} io((\!| P |\!) \cap \mathcal{A})$$

A.2.2 Composite state machines

We assume that basic state machines are autonomous, and therefore executed in parallel. Semantically, parallel composition is defined by *interleaving* traces.

A trace t is an interleaving of the traces $s_1, \ldots, s_n$, written $inl((s_1, \ldots, s_n), t)$, iff $s_1, \ldots, s_n$ are sub-sequences of t, i.e,

$$\frac{-}{inl((\langle\rangle, \ldots, \langle\rangle), \langle\rangle)} \qquad \frac{inl((s_1, \ldots, s_i, \ldots, s_n), t)}{inl((s_1, \ldots, \langle e\rangle \frown s_i, \ldots, s_n), \langle e\rangle \frown t)}$$

We are only interested in interleavings that are well formed in the sense that they describe communication in which messages are sent before they are received. We write $wft(s)$ if trace s is well formed. The predicate is formally defined by the rules:

$$\frac{-}{wft(\langle\rangle)} \qquad \frac{wft(s \frown t)}{wft(s \frown \langle e\rangle \frown t)}\text{if } e \in \mathbf{OE} \qquad \frac{wft(s \frown t)}{wft(s \frown \langle(!,m)\rangle \frown t \frown \langle(?,m)\rangle)}$$

We let $\mathcal{T}$ denote the set of all well formed traces, i.e.,

$$\mathcal{T} \overset{\text{def}}{=} \{s \in \mathbf{E}^* \mid wft(s)\}$$

We define the parallel composition operator as the set of all well formed interleavings of its arguments, i.e.,

$$\parallel (s_1, \ldots, s_n) \overset{\text{def}}{=} \{t \in \mathcal{T} \mid inl((s_1, \ldots, s_n), t)\} \tag{12}$$

The operator is lifted to trace sets as follows

$$\parallel (\Phi_1, \ldots, \Phi_n) \overset{\text{def}}{=} \bigcup_{s_1 \in \Phi_1, \ldots, s_n \in \Phi_n} \parallel (s_1, \ldots, s_n) \tag{13}$$

Definition A.5 (Semantics of composite state machines) *The semantics of a composite state machine expression $P = (P_1 \parallel \ldots \parallel P_n)$, written $[\![P]\!]$, is defined by*

$$[\![P]\!] \overset{\text{def}}{=} \parallel ([\![P_1]\!], \ldots, [\![P_n]\!])$$

A.3 Semantics of state machines with potential choice

The semantics of a state machine expression with potential choice is a set of trace sets called *obligations*. Intuitively, the traces of an obligation are equivalent in the sense that an implementation is only required to produce one of them. Thus, each trace of an obligation represents underspecification, i.e., a potential choice. However, an implementation is required to produce at least one trace in each obligation. Thus obligations represent explicit choices that have to be present in an implementation.

To define the semantics formally, we lift the concatenation operator $\frown$ to sets of obligations as follows

$$\Omega \frown \Omega' \overset{\text{def}}{=} \{\phi \frown \phi' \mid \phi \in \Omega \wedge \phi' \in \Omega'\}$$

We denote by Ω^n, the set of obligations Ω concatenated n times. We define Ω^0 by $\{\{\langle\rangle\}\}$.

We define the *inner union* of two sets of obligations, written $\Omega \uplus \Omega'$, by

$$\Omega \uplus \Omega \overset{\text{def}}{=} \{\phi \cup \phi' \mid \phi \in \Omega \wedge \phi' \in \Omega'\}$$

As we did in the previous section, we first define the intermediate semantics.

Definition A.6 (Intermediate semantics of basic state machines) *The intermediate semantics of a basic state machine P with potential choice, written $(\!| P |\!)$, is defined*

$$\begin{array}{lll}
(\!|\, (ie_\epsilon, bx_\epsilon, oe_\epsilon, \epsilon) \,|\!) & \overset{\mathsf{def}}{=} & \{\{\langle eval(\sigma((ie_\epsilon, bx_\epsilon, oe_\epsilon, \epsilon)), \sigma, \sigma)\rangle \,|\sigma \in \widehat{\Sigma}\}\} \\
(\!|\, (ie_\epsilon, bx_\epsilon, oe_\epsilon, (x, ex)) \,|\!) & \overset{\mathsf{def}}{=} & \{\{\langle eval(\sigma((ie_\epsilon, bx_\epsilon, oe_\epsilon, (x, ex)))), \sigma, \\
 & & \quad \sigma[x \mapsto eval(\sigma(ex))])\rangle \,|\sigma \in \widehat{\Sigma}\}\} \\
(\!|\, P.Q \,|\!) & \overset{\mathsf{def}}{=} & (\!|\, P \,|\!) \frown (\!|\, Q \,|\!) \\
(\!|\, P + Q \,|\!) & \overset{\mathsf{def}}{=} & (\!|\, P \,|\!) \cup (\!|\, Q \,|\!) \\
(\!|\, P|Q \,|\!) & \overset{\mathsf{def}}{=} & (\!|\, P \,|\!) \uplus (\!|\, Q \,|\!) \\
(\!|\, P^* \,|\!) & \overset{\mathsf{def}}{=} & \bigcup_{i\in\mathbb{N}}((\!|\, P \,|\!))^i
\end{array}$$

We are now ready to define the semantics of state machine expressions with potential choices.

Definition A.7 (Semantics of basic state machines) *The semantics of a basic state machine expression P with potential choice, written $[\![P]\!]$, is defined by*

$$[\![P]\!] \overset{\mathsf{def}}{=} \{io(\phi \cap \mathcal{A}) \mid \phi \in (\!|\, P \,|\!)\}$$

The semantics of a composite state machine expression is the parallel composition of the (basic) state machine expressions it consists of.

Definition A.8 (Semantics of composite state machines) *The semantics of the composite state machine expression $P = (P_1 \parallel \cdots \parallel P_n)$ with potential choice, written $[\![P]\!]$, is defined*

$$[\![P]\!] \overset{\mathsf{def}}{=} \bigcup_{\phi_1\in[\![P_1]\!],\ldots,\phi_n\in[\![P_n]\!]} \{\parallel (\phi_1, \ldots, \phi_n)\}$$

B Event transformations

In this section, we give a formal characterization of event transformations and show that these transformations satisfy some desirable properties. In Sect. B.1, we consider transformation of basic state machines, whereas in Sect. B.2, transformation of composite state machines is considered.

B.1 Basic state machines

In this section, we give a formal characterization of the transformation which is induced by an event library. The transformation is defined for state machines without potential choices.

An event library is a set of *event specifications*. An event specification is a pair (e^d, P^p) consisting of an event definition e^d and a state machine pattern P^p defining the behavior of the event. An event definition is of the form (k, si^d) where k is a kind (! or ?), and si^d is a *signal definition* of the form $st(fp_1, \ldots, fp_n)$ where $fp_1, \ldots, fp_n$ are formal parameters. We let **FP** denote the set of all formal parameters. The set of all signal definitions $\mathbf{SI}^d$ and event definitions $\mathbf{E}^d$ are defined by

$$\mathbf{SI}^d \overset{\mathsf{def}}{=} \mathbf{Str} \times \mathbf{FP}^* \qquad \mathbf{E}^d \overset{\mathsf{def}}{=} \{!, ?\} \times \mathbf{SI}^d$$

A state machine pattern P^p is a state machine expression that may contain formal parameters. To make this more precise, we define the notion of expression pattern, event pattern, and action pattern.

An expression pattern ex^p is an expression that may contain formal parameters. We let $\mathbf{Exp}^p$ denote the set of all expression patterns, and $\mathbf{BExp}^p$ denote the set of all boolean expression patterns. Note that $\mathbf{Exp} \subset \mathbf{Exp}^p$ and $\mathbf{BExp} \subset \mathbf{BExp}^p$.

An event pattern e^p is an event whose transmitter and receiver may contain special formal parameters and whose signals may contain expression patterns. The set of all expression patterns $\mathbf{E}^p$ is defined by

$$\mathbf{E}^p \stackrel{\text{def}}{=} \{!, ?\} \times ((\mathbf{Nm} \cup \{to\}) \times (\mathbf{Nm} \cup \{from\}) \times (\mathbf{FP} \cup (\mathbf{Str} \times (\mathbf{ExP}^p)^*)))$$

where to and $from$ are special formal parameters, i.e., $to, from \in \mathbf{FP}$.

We let $\mathbf{IE}^p$ and $\mathbf{OE}^p$ denote the set of all input event patterns and output event patterns, respectively. These sets are defined by

$$\mathbf{IE}^p \stackrel{\text{def}}{=} \{(k, m^p) \in \mathbf{E}^p \mid k = ?\} \qquad \mathbf{OE}^p \stackrel{\text{def}}{=} \{(k, m^p) \in \mathbf{E}^p \mid k = !\}$$

The set of all action patterns $\mathbf{Act}^p$ is defined by

$$\mathbf{Act}^p \stackrel{\text{def}}{=} (\mathbf{IE}^p \cup \{\epsilon\}) \times (\mathbf{BExp}^p \cup \{\epsilon\}) \times (\mathbf{OE}^p \cup \{\epsilon\}) \times ((\mathbf{Var} \times \mathbf{Exp}^p) \cup \{\epsilon\})$$

We are now ready to define state machine patterns precisely.

Definition B.1 (State machine pattern) *The set of all state machine patterns* $\mathbf{P}^p$ *is defined by the following grammar*

$$P^p ::= act^p \mid (P^p)^* \mid P^p.P^p \mid P^p + P^p$$

where $act^p \in \mathbf{Act}^p$.

The notion of an event library is made precise by the following definition.

Definition B.2 (Event library) *The set of all event libraries* $\mathbf{EL}$ *is given by*

$$\mathbf{EL} \stackrel{\text{def}}{=} \mathbf{E}^d \times \mathbf{P}^p$$

For every event library EL in $\mathbf{EL}$*, we require the signal definitions in EL to be unique in the following sense*

$$((k, st(fp_1, \ldots, fp_j)), P^p_1), ((k, st(fp'_1, \ldots, fp'_k)), P^p_2) \in EL \implies j \neq k$$

In addition, we require that each event specification $((k, (si^d, P^p)))$ *in EL must satisfy*

$$ffp(P^p) \subseteq ffp(si^d)$$

where the function ffp yields the set of all formal parameters in a signal definition or a state machine pattern.

When a transformation induced by an event library EL is applied to a state machine P, all events in P for which there is a matching event definition in EL, are replaced by their specification in EL. An event $(k, (nm_t, nm_r, si))$ matches an event definition (k, si^d) if there is a substitution sub that replaces formal parameters by expressions such that $si = sub(si^d)$. More precisely, a substitution $sub \in \mathbf{FP} \rightarrow \mathbf{Exp}$ is a function that replaces formal parameters by expressions. We let $\mathcal{S}ub$ denote the set of all substitutions. Any substitution

sub is lifted to expression patterns such that $sub(ex^p)$ yields the expression obtained from ex^p by replacing each formal parameter fp in ex^p by $sub(fp)$. If ex^p contains no formal parameters, then $sub(ex^p) = ex^p$. Substitutions are further lifted to event patterns and assignment patterns as follows

$$\begin{array}{lcl} sub((k,(nm^p_t, nm^p_r, si(ex^p_1,\ldots,ex^p_n)))) & \overset{\mathsf{def}}{=} & (k,(sub(nm^p_t), sub(nm^p_r), \\ & & \quad si(sub(ex^p_1),\ldots,sub(ex^p_n)))) \\ sub((k,(nm^p_t, nm^p_r, fp))) & \overset{\mathsf{def}}{=} & (k,(sub(nm^p_t), sub(nm^p_r), sub(fp))) \\ sub((x, ex^p)) & \overset{\mathsf{def}}{=} & (x, sub(ex^p)) \end{array}$$

and to state machine patterns as follows

$$\begin{array}{lcl} sub((k,(ie^p_\epsilon, bx^p_\epsilon, oe^p_\epsilon, a^p_\epsilon))) & \overset{\mathsf{def}}{=} & (k,(sub(ie^p_\epsilon), sub(bx^p_\epsilon), sub(oe^p_\epsilon), sub(a^p_\epsilon))) \\ sub(P^p_1.P^p_2) & \overset{\mathsf{def}}{=} & sub(P^p_1).sub(P^p_2) \\ sub(P^p_1 + P^p_2) & \overset{\mathsf{def}}{=} & sub(P^p_1) + sub(P^p_2) \\ sub((P^p)^*) & \overset{\mathsf{def}}{=} & sub(P^p)^* \end{array}$$

Note that ie^p_ϵ denotes an input event pattern or an ϵ, i.e., $bx^p_\epsilon \in \mathbf{IE}^p \cup \{\epsilon\}$. The same convention is also used for boolean expression patterns, output event patterns, and assignment patterns.

We let $te_{\_}(\_) \in \mathbf{EL} \rightarrow (\mathbf{E} \rightarrow \mathbf{P})$ be the function that replaces events by their definition in an event library. Formally, the function is defined by

$$\begin{array}{lcl} te_{EL}((k,(nm_t, nm_r, si))) & \overset{\mathsf{def}}{=} & sub[\{to \mapsto nm_t, from \mapsto nm_r\}](P^p) \\ & & \text{if there are } ((k, si^d), P^p) \in EL \\ & & \text{and } sub \in \mathcal{S}ub \text{ such that } sub(si^d) = si \end{array} \tag{14}$$

Note that the special formal parameters to and $from$ are bound to the transmitter and the receiver of messages, respectively. As we did in App. A.2, we let $sub[fp \mapsto ex]$ denote the mapping sub except that fp maps to ex.

The function $tact_{\_}(\_) \in \mathbf{EL} \rightarrow (\mathbf{Act} \rightarrow \mathbf{P})$ that substitutes events in actions by their specification in an event library is defined as follows

$$\begin{array}{lcl} tact_{EL}((ie, bx_\epsilon, \epsilon, a_\epsilon)) & \overset{\mathsf{def}}{=} & (\epsilon, bx_\epsilon, \epsilon, \epsilon).te_{EL}(ie).(\epsilon,\epsilon,\epsilon,a_\epsilon) \\ & & \quad \text{if } ie \in \mathcal{D}om(te_{EL}) \\ tact_{EL}((\epsilon, bx_\epsilon, oe, a_\epsilon)) & \overset{\mathsf{def}}{=} & (\epsilon, bx_\epsilon, \epsilon, \epsilon).te_{EL}(oe).(\epsilon,\epsilon,\epsilon,a_\epsilon) \\ & & \quad \text{if } oe \in \mathcal{D}om(te_{EL}) \\ tact_{EL}((ie_\epsilon, bx_\epsilon, oe_\epsilon, a_\epsilon)) & \overset{\mathsf{def}}{=} & (ie_\epsilon, bx_\epsilon, oe_\epsilon, a_\epsilon) \\ & & \quad ie_\epsilon \notin \mathcal{D}om(te_{EL}) \text{ and} \\ & & \quad oe_\epsilon \notin \mathcal{D}om(te_{EL}) \end{array} \tag{15}$$

We let T_{EL} be the function that takes a basic state machine P and yields the state machine $T_{EL}(P)$ obtained from P by replacing the events occurring in each action of P by their definition in EL. We let T_{EL} rename variables of the event specification such that no name clashes occur when events are being replaced by their definitions. To make this precise, we make use of a renaming function $rnm(\_,\_) \in \mathbf{P} \times \mathbf{EL} \rightarrow \mathbf{EL}$. That is, $rnm(P, EL)$ yields the event library obtained from EL by renaming all its variables such that the following condition is satisfied

$$var(rnm(P, EL)) \cap var(P) = \emptyset$$

where the function var is lifted to event libraries and state machines such that $var(EL')$ yields the set of all variables occurring in the state machine patterns of EL', and $sub(P)$ yields the set of all variables occurring in P.

The event transformation T_{EL} that applies $tact_{EL}$ to all actions of a basic state machine is given by the following definition.

Definition B.3 (Basic event transformation) *The event transformation* $T_{\_}(\_) \in \mathbf{EL} \rightarrow (\mathbf{P} \rightarrow \mathbf{P})$ *for basic state machines is defined by*

$$T_{EL}(P) \overset{\mathsf{def}}{=} T'_{rnm(P,EL)}(P)$$

where $T'_{\_}(\_) \in \mathbf{EL} \rightarrow (\mathbf{P} \rightarrow \mathbf{P})$ *is defined by*

$$\begin{array}{lcl} T'_{EL'}(act) & \overset{\mathsf{def}}{=} & tact_{EL'}(act) \\ T'_{EL'}(P.Q) & \overset{\mathsf{def}}{=} & T'_{EL'}(P).T'_{EL'}(Q) \\ T'_{EL'}(P+Q) & \overset{\mathsf{def}}{=} & T'_{EL'}(P)+T'_{EL'}(Q) \\ T'_{EL'}(P^*) & \overset{\mathsf{def}}{=} & T'_{EL'}(P)^* \end{array}$$

Definition B.4 (Semantics of basic event transformations) *The semantics of a transformation* T_{EL}*, written* $[\![T_{EL}]\!]$*, is defined by*

$$\{([\![P]\!], [\![T_{EL}(P)]\!]) \mid P \in \mathcal{D}om(T_{EL})\}$$

Here $\mathcal{D}om(T_{EL})$ yields the domain of the function T_{EL}.

The renaming of variables in event libraries ensures that event specifications are side effect free. This has the consequence that event transformations preserve semantic equality and are therefore functional.

Theorem B.1 *The relation* $[\![T_{EL}]\!]$ *is a function.*

The following lemmas tells us that the semantics of an event transformation is entirely characterized by its behavior on events. This property is useful when defining conditions under which event transformations are security preserving.

Lemma B.1 *If* EL *is an event library, and* P *and* Q *be basic state machines whose variables are disjoint from those in* EL*, then* $[\![T_{EL}{}']\!]$ *is homomorphic w.r.t. union, i.e.,*

$$[\![T'_{EL}]\!]([\![P]\!] \cup [\![Q]\!]) = [\![T'_{EL}]\!]([\![P]\!]) \cup [\![T'_{EL}]\!]([\![Q]\!])$$

Lemma B.2 *If* EL *is an event library, and* P *and* Q *be basic state machines whose variables are disjoint from those in* EL*, then* $[\![T'_{EL}]\!]$ *is homomorphic w.r.t. concatenation, i.e.,*

$$[\![T'_{EL}]\!]([\![P]\!]) \frown [\![T'_{EL}]\!]([\![Q]\!]) = [\![T'_{EL}]\!]([\![P]\!] \frown [\![Q]\!])$$

B.2 Composite state machines

The transformation induced by an event library EL which takes a *composite* state machine P as input, written T^C_{EL}, is simply the function that applies T_{EL} to all basic state machines that P consists of.

Definition B.5 (Composite event transformations) *The composite transformation $T^C_{\_}(\_) \in \mathbf{EL} \to (\mathbf{P}^C \to \mathbf{P}^C)$ induced by an event library is defined*

$$T^C_{EL}(P_1 \parallel \cdots \parallel P_n) \overset{\mathsf{def}}{=} T'_{EL'}(P_1) \parallel \cdots \parallel T'_{EL'}(P_n)$$

where $EL' = rnm(P_1 \parallel \cdots \parallel P_n, EL)$.

The semantics of T^C_{EL} is defined exactly in the same way as for T_{EL}.

Definition B.6 (Semantics of composite event transformations) *The semantics of the transformation T^C_{EL}, written $[\![T^C_{EL}]\!]$, is the relation defined by*

$$\{([\![P]\!], [\![T^C_{EL}(P)]\!]) \mid P \in \mathcal{D}om(T^C_{EL})\}$$

A transformation T^C_{EL} may in general transform non well formed traces resulting from parallel composition at the abstract level into well formed traces at the concrete level. We say that the *image* of a transformation T^C_{EL}, written Im_{EL}, is the set of all concrete traces that have an abstract equivalent, i.e., the set of all concrete traces that correspond to well formed abstract traces. Formally, Im_{EL} is the least set satisfying the following condition

$$\bigwedge_{i\in\{1,\ldots,n\}}(s_i \in (\mathbf{E}_{nm_i})^* \wedge s'_i \in [\![T_{EL}]\!](\{s_i\})) \wedge \parallel(s_1,\ldots,s_n) \neq \emptyset \implies \parallel(s'_1,\ldots,s'_n) \subseteq Im_{EL} \tag{16}$$

for all $\{nm_1, \ldots, nm_n\} \subseteq \mathbf{Nm}$.

The semantics of transformation T^C_{EL} restricted to its image, written $\widehat{T}_{EL}$, is defined

$$\widehat{T}_{EL} \overset{\mathsf{def}}{=} \{([\![P]\!], [\![T^C_{EL}(P)]\!] \cap Im_{EL}) \mid P \in \mathcal{D}om(T^C_{EL})\} \tag{17}$$

Theorem B.2 *The relation $[\![T^C_{EL}]\!]$ is a function when restricted to its image, i.e.,*

$$[\![P]\!] = [\![Q]\!] \implies [\![T^C_{EL}(P)]\!] \cap Im_{EL} = [\![T^C_{EL}(Q)]\!] \cap Im_{EL}$$

Lemma B.3 *The semantics of the event transformation for composite state machines induced by an event library EL is homomorphic w.r.t. the union operator on trace sets when restricted to its image, i.e.,*

$$\widehat{T}_{EL}([\![P]\!] \cup [\![Q]\!]) = \widehat{T}_{EL}([\![P]\!]) \cup \widehat{T}_{EL}([\![Q]\!])$$

B.3 Adherence preservation under event transformations

In this section, we present general conditions under which security properties are preserved under event transformations. We then show how these general conditions can be instantiated into specific conditions for particular security properties.

We recall from [34, 33], that a security property *SecP* is a conjunction of basic security predicates $\text{BSP}_{\mathfrak{r}\mathfrak{h}}(\Phi)$ of the form

$$\forall s, t \in \Phi : s \xrightarrow{\mathfrak{r}} t \implies \exists u \in \Phi : s \xrightarrow{\mathfrak{h}} u \sim_{\mathfrak{l}} t \tag{18}$$

for the restriction relation $\mathfrak{r}$, the high level relation $\mathfrak{h}$, and the low-level equivalence relation $\mathfrak{l}$ on traces (i.e., $\mathfrak{r}, \mathfrak{h}, \mathfrak{l} \subseteq \mathbf{E}^* \times \mathbf{E}^*$). Note that if $\mathfrak{c}$ is a relation, we write $s \xrightarrow{\mathfrak{c}} t$ for $(s,t) \in \mathfrak{c}$, and $s \sim_{\mathfrak{c}} t$ for $(s,t) \in \mathfrak{c}$ if $\mathfrak{c}$ is an equivalence relation.

We will henceforth use the following definition of the low level equivalence relation $\mathfrak{l}$

$$s \sim_{\mathfrak{l}} t \Leftrightarrow s|_L = t|_L \tag{19}$$

for a set of low level events L.

A transformation is secure w.r.t. a security property $SecP$ if the image of an abstract specification P is secure w.r.t. $SecP$ if P is secure w.r.t. $SecP$. We restrict attention to the image since we cannot exploit the fact that the abstract specification is secure to ensure that concrete traces that do not have any abstract equivalent do not violate security. This means that additional security analysis is needed at the concrete level to ensure that those traces that do not have any abstract equivalent do not violate security.

Definition B.7 (Security preservation) *Let EL be an event library and $SecP$ be a security property. Then transformation T^C_{EL} induced by EL preserves security property $SecP$ for specification P iff*

$$SecP([\![P]\!]) \implies SecP(\widehat{T}_{EL}([\![P]\!]))$$

The following theorem presents general conditions under which a basic security predicate is preserved by a transformation.

Theorem B.3 *Let $\mathfrak{r}$ be a restriction, $\mathfrak{h}$ be a high level relation, and T^C_{EL} be the transformation induced by event library EL. Then T^C_{EL} preserves $\text{BSP}_{\mathfrak{r}\mathfrak{h}}$ for specification P if the following conditions are satisfied for all $s, t, u \in [\![P]\!], s' \in \widehat{T}_{EL}(\{s\}), t' \in \widehat{T}_{EL}(\{t\})$*

$$s' \xrightarrow{\mathfrak{r}} t' \implies s \xrightarrow{\mathfrak{r}} t \tag{20}$$

$$s \xrightarrow{\mathfrak{r}} t \wedge s \xrightarrow{\mathfrak{h}} u \sim_{\mathfrak{l}} t \implies \exists u' \in \widehat{T}_{EL}(\{u\}) : s' \xrightarrow{\mathfrak{h}} u' \sim_{\mathfrak{l}} t' \tag{21}$$

Note that we have exploited Lemma B.3 in the definition of the conditions since the property ensures that the transformation $\widehat{T}_{EL}$ is defined for singleton sets. Obviously, a transformation preserves a security property $SecP$ if the conditions are satisfied for all basic security predicates that $SecP$ consists of.

Corollary B.1 *Let $SecP$ be a security property, i.e., a conjunction of basic security predicates. Then $SecP$ is preserved under a transformation T^C_{EL} if each basic security predicate of $SecP$ satisfies the conditions (20) and (21) of Theorem B.3.*

The conditions of Theorem B.3 may be used to derive specific conditions that can be used to check that a transformation preserves a given security property. The procedure for deriving such conditions for an arbitrary security property $SecP$ and event library EL is as follows. For each basic security predicate $\text{BSP}_{\mathfrak{r}\mathfrak{h}}$ that $SecP$ consists of

- define two conditions C_1 and C'_1 and show that C_1 implies (20) and C'_1 implies (21) for the transformation T_{EL} of basic state machines induced by EL;

- define two conditions C_2 and C'_2 and show that $C_1 \wedge C_2$ implies (20) and $C'_2 \wedge C'_2$ implies (21) for the composite transformation T^C_{EL} induced by EL.

In the following we illustrate the procedure for the *non-inference* property [30].

Definition B.8 *The security property non-inference, written* Nf*, consists of the basic security predicate* $\text{Bsp}_{\mathfrak{rh}}$ *whose restriction* $\mathfrak{r}$ *and high level relation* $\mathfrak{h}$ *are defined*

$$\begin{array}{rcl} s \xrightarrow{\mathfrak{r}} t & \Leftrightarrow & true \\ s \xrightarrow{\mathfrak{h}} t & \Leftrightarrow & t|_{\mathcal{H}} = \langle\rangle \end{array}$$

According to the procedure, we must first define two conditions C_1 and C'_1 and show that C_1 implies (20) and C'_1 implies (21), respectively. By definition of $\mathfrak{r}$, any transformation will trivially satisfy condition (20). Thus we need only consider the latter condition (21). We first instantiate (21) for the basic security predicate defining the Nf property. We get for all traces t and u, and all $t' \in \widehat{T}_{EL}(\{t\})$

$$u|_H = \langle\rangle \wedge u|_L = t|_L \implies \exists u' \in \widehat{T}_{EL}(\{u\}) : u'|_H = \langle\rangle \wedge u'|_L = t'|_L \qquad (22)$$

To define a condition C'_1 that implies (22) for basic transformations, we first examine the case in which $[\![T_{EL}]\!]$ is applied to single events. We can do so because $[\![T_{EL}]\!]$ is homomorphic w.r.t. union and concatenation of trace sets (by Lemma B.1 and Lemma B.2). It is easy to see that any transformation that transforms a non high level event into a set of traces that contain a high level event will fail to satisfy condition (22). Hence, we define a condition that ensures that this never occurs:

$$e \notin H \implies \forall s \in [\![T_{EL}]\!](\{e\}) : s|_H = \langle\rangle \qquad (23)$$

The same consideration holds for low level events; if a non low level event is transformed into a set of traces that contain a low level event, then condition (22) is broken. Therefore we must require

$$e \notin L \implies \forall s \in [\![T_{EL}]\!](\{e\}) : s|_L = \langle\rangle \qquad (24)$$

When the set L of low level events, and the set H of high level events are defined as all events that may occur in particular state machines, then conditions (23) and (24) are trivially satisfied because event transformations cannot change the transmitters or receivers of events. By exploiting Lemma B.2, it follows by induction over the length of traces u and t that any transformation on *basic state machines* satisfying conditions (23) and (24) will satisfy the condition (22).

The final step of the procedure is to define a condition C'_2 and to show that this condition together with (23) and (24) implies (22) for transformations of *composite* state machines. To obtain C'_2 we observe that the first part of condition (22) ($u'|_H = \langle\rangle$) is ensured by (23). However, this is not the case for the second part ($u'|_L = t'|_L$) which requires low level equality to be preserved. This condition can be rephrased as

$$\widehat{T}_{EL}(\{t|_L\}) = \widehat{T}_{EL}(\{t\})|_L \qquad (25)$$

for all traces t.

The reason why condition (25) may not be satisfied is that low level observations on the abstract level may be transformed into several concrete low level observations that may be part of traces that are not well formed. To see this, consider the upper most composite state machine of Fig. 12. Let the set of low level events L be the set of events that occur in state machine $\texttt{P}_1$, and the set H of high level events be $\{?d, ?f\}$. The semantics of $\texttt{A}$ is

$$[\![\,\texttt{A}\,]\!] = \| \, (\{\langle !a \rangle\}, \{\langle ?a, !c \rangle, \langle ?a, !d \rangle\}, \{\langle ?c \rangle, \langle ?d \rangle\}) = \{\langle !a, ?a, !c, ?c \rangle, \langle !a, ?a, !d, ?d \rangle\}$$

The specification is secure w.r.t. the NF-property because when the low level user observes $\langle !a \rangle$ (this is the only observation the low level user can make), he will not know whether the high level event $?d$ will occur.

Now let EL be an event library defined such that $T^{C}_{EL}(\texttt{A}) = \texttt{C}$ where $\texttt{C}$ is the lower most specification of Fig. 12. Specification $\texttt{C}$ has the following semantics

$$[\![\,\texttt{C}\,]\!] = \| \, (\{\langle !a_1, !a_1 \rangle, \langle !a_2, !a_2 \rangle\}, \{\langle ?a_1, ?a_1, !e \rangle, \langle ?a_1, ?a_2, !f \rangle, \langle ?a_2, ?a_1, !e \rangle,$$
$$\langle ?a_2, ?a_2, !f \rangle\}, \{\langle ?e \rangle, \langle ?f \rangle\}) = \{\langle !a_1, !a_1, ?a_1, ?a_1, !e, ?e \rangle, \langle !a_2, !a_2, ?a_2, ?a_2, !f, ?f \rangle\}$$

The specification $\texttt{C}$ is not secure w.r.t. the NF property because the low level user will know that the high level event $?f$ occurs when the observation $\langle !a_2, !a_2 \rangle$ is made. The problem is that the low level observation $!a$ has been transformed into the two observations $\langle !a_1, !a_1 \rangle$ and $\langle !a_2, !a_2 \rangle$. But the trace in which both observation $\langle !a_2, !a_2 \rangle$ and event $!e$ (instead of the high level event $!f$) occurs is not well formed, and therefore not part of the concrete specification.

The check that condition (25) is satisfied for a state machine P, we may for instance, check that the traces obtained by transforming two corresponding events of the event library are not affected by the context they appear in, i.e.,

$$((!, m), (?, m) \in \mathcal{D}om(te_{EL}) \wedge t \in [\![\, P \,]\!] \wedge \langle (!, m), (?, m) \rangle \lhd t) \implies$$
$$((s' \in \widehat{T}_{EL}(\{\langle (!, m), (?, m) \rangle\})) \Leftrightarrow (\exists t' \in \widehat{T}_{EL}(\{t\}) : s' \lhd t'))$$

where $s \lhd t$ means that s is a sub-trace of t, i.e., a trace $s = \langle e_1, \ldots e_n \rangle$ is a sub-trace of t iff

$$s_1 \frown \langle e_1 \rangle \frown \cdots \frown s_n \frown \langle e_n \rangle \frown s_{n+1} = t$$

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

The conditions (23), (24), and (25) implies that (22) holds. By Theorem B.3 and definition of NF, this means that any transformation satisfying (23), (24), and (25) will preserve the NF property.

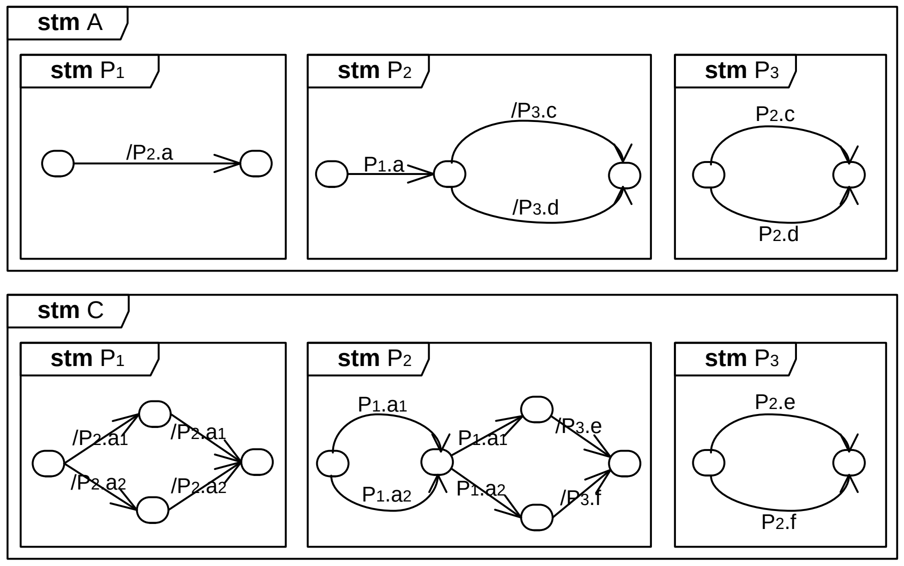

Figure 12: Example

C Syntactic categories

*Set**			*Meaning*
$\mathbb{P}(A)$	$\stackrel{\text{def}}{=}$	$\{X \mid X \subseteq A\}$	The power set of A.
A^*			The set of all finite sequences over the set A.
$ax \in \mathbf{AExp}$			Arithmetic expressions
$bx \in \mathbf{BExp}$			Boolean expressions
$sx \in \mathbf{SExp}$			String expressions
$ex \in \mathbf{Exp}$	$\stackrel{\text{def}}{=}$	$\mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{SExp}$	Expressions
$\mathbf{Val}$	$\subseteq$	$\mathbf{Exp}$	The set of all values.
$x, y \in \mathbf{Var}$	$\subseteq$	$\mathbf{Exp}$	The set of all variables.
$a \in \mathbf{Assign}$	$\stackrel{\text{def}}{=}$	$\mathbf{Var} \times \mathbf{Exp}$	The set of all assignments.
$nm \in \mathbf{Nm}$			The set of all names.
$si \in \mathbf{SI}$			The set of all signals.
$m \in \mathbf{M}$	$\stackrel{\text{def}}{=}$	$\mathbf{Nm} \times \mathbf{Nm} \times \mathbf{SI}$	The set of all messages.
$ie \in \mathbf{IE}$	$\stackrel{\text{def}}{=}$	$\{?\} \times \mathbf{M}$	The set of all input events.
$oe \in \mathbf{OE}$	$\stackrel{\text{def}}{=}$	$\{!\} \times \mathbf{M}$	The set of all output events.
$e \in \mathbf{E}$	$\stackrel{\text{def}}{=}$	$\mathbf{IE} \cup \mathbf{OE}$	The set of all events.
$act \in \mathbf{Act}$	$\stackrel{\text{def}}{=}$	$\mathbf{IE} \cup \{\epsilon\} \times \mathbf{BExp} \cup \{\epsilon\} \times \mathbf{OE} \cup \{\epsilon\} \times \mathbf{Assign} \cup \{\epsilon\}$	The set of all actions.
$P \in \mathbf{P}$			Set of all basic state machine expressions.
$P \in \mathbf{P}^C$			Set of all composite state machine expressions.

* By the notation $a \in A$ we understand that A is ranged over by a.

D Proofs

D.1 Auxiliary definitions

In this section, we make some definitions that are needed in the proofs.

Definition D.1 *We let the function* $|\_| \in \mathbf{P} \rightarrow \mathbb{P}(\mathbf{Act}^*)$ *yield the set of action sequences described by a basic state machine without potential choice. This function is defined by*

$$\begin{array}{lcl} |act| & \overset{\mathsf{def}}{=} & \{\langle act \rangle\} \\ |P.Q| & \overset{\mathsf{def}}{=} & |P| \frown |Q| \\ |P+Q| & \overset{\mathsf{def}}{=} & |P| \cup |Q| \\ |P^*| & \overset{\mathsf{def}}{=} & \bigcup_{i \in \mathbb{N}} (|P|)^i \end{array}$$

Definition D.2 *We let the functions* $preState(\_), postState(\_) \in (\mathbf{Act} \times \widehat{\Sigma} \times \widehat{\Sigma})^+ \rightarrow \widehat{\Sigma}$ *yield the first and the last state in a non-empty sequence of action-state triples, respectively. These functions are defined by*

$$\begin{array}{lcl} preState(\langle (act, \sigma, \sigma') \rangle \frown as) & \overset{\mathsf{def}}{=} & \sigma \\ postState(as \frown \langle (act, \sigma, \sigma') \rangle) & \overset{\mathsf{def}}{=} & \sigma' \end{array}$$

In addition, we let the functions $preState(\_, \_), postState(\_)_{\_} \in (\mathbf{Act} \times \widehat{\Sigma} \times \widehat{\Sigma})^+ \times \mathbb{P}(\mathbf{Var}) \rightarrow \widehat{\Sigma}$ *be defined by*

$$\begin{array}{lcl} preState(as, V) & \overset{\mathsf{def}}{=} & preState(as) \setminus (V \times \mathbf{Val}) \\ postState(as, V) & \overset{\mathsf{def}}{=} & postState(as) \setminus (V \times \mathbf{Val}) \end{array}$$

Definition D.3 *We let* $pr(\_)$ *be the function that yields the prefix closure of a set of sequences. The function is defined by*

$$pr(A) \overset{\mathsf{def}}{=} \{s \mid t \in A \wedge s \sqsubseteq t\}$$

where $\sqsubseteq$*, the prefix predicate, is defined by*

$$s \sqsubseteq t \Leftrightarrow \exists u : s \frown u = t$$

D.2 Basic transformations

Theorem B.1 The relation $[\![T_{EL}]\!]$ is a function.

Proof of Theorem B.1

ASSUME: 1. $[\![P]\!] = [\![Q]\!]$ for some $P, Q \in \mathbf{P}$
2. $EL \in \mathbf{EL}$

PROVE: $[\![T_{EL}(P)]\!] = [\![T_{EL}(Q)]\!]$

⟨1⟩1. ASSUME: 1.1 $EL_1 = rnm(P, EL)$ and $EL_2 = rnm(Q, EL)$ for $EL_1, EL_2 \in$ **EL**

PROVE: $[\![T'_{EL_1}(P)]\!] = [\![T'_{EL_2}(Q)]\!]$

⟨2⟩1. ASSUME: 2.1 $s' \in [\![T'_{EL_1}(P)]\!]$

PROVE: $s' \in [\![T'_{EL_2}(Q)]\!]$

⟨3⟩1. Choose $as'_1 \in (\!| T'_{EL_1}(P) |\!) \cap \mathcal{A}$ such that $io(as'_1) = s'$

PROOF: By assumption 2.1 and definition of $[\![\ ]\!]$ (Def. A.4).

$\langle 3\rangle 2.$ Choose $\langle act_{1.1}, \ldots, act_{1.j}\rangle \in |P|$ and $as'_{1.1} \in (\!|\, T'_{EL_1}(act_{1.1}) \,|\!), \ldots, as'_{1.j} \in (\!|\, T'_{EL_1}(act_{1.j}) \,|\!)$ such that $as'_1 = as'_{1.1} \frown \cdots \frown as'_{1.j}$

PROOF: By $\langle 3\rangle 1$ and Lemma T.B.1.1.

$\langle 3\rangle 3.$ Choose $as_1 \in \mathcal{A}$ and $as_{1.1} \in (\!|\, act_{1.1} \,|\!), \ldots, as_{1.j} \in (\!|\, act_{1.j} \,|\!)$ such that $as_1 = as_{1.1} \frown \cdots \frown as_{1.j}$ and $\forall i \in \{1, \ldots, j\} : preState(as_{1.i}, var(EL_1)) = preState(as'_{1.i}, var(EL_1)) \wedge postState(as_{1.i}, var(EL_1)) = postState(as'_{1.i}, var(EL_1))$

PROOF: By assumptions 1.1 and 2, $\langle 3\rangle 1$, $\langle 3\rangle 2$, and Lemma T.B.1.2.

$\langle 3\rangle 4.$ $io(as_1) \in [\![\, P \,]\!]$

PROOF: By $\langle 3\rangle 2$, $\langle 3\rangle 3$, definition of $(\!|\, \_ \,|\!)$ (Def. A.3), definition of $[\![\, \_ \,]\!]$ (Def. A.4), and definition of *var* and *rnm* (App. B.1).

$\langle 3\rangle 5.$ Choose $as_2 \in (\!|\, Q \,|\!) \cap \mathcal{A}$ such that $io(as_2) = io(as_1)$

PROOF: By $\langle 3\rangle 4$, assumption 1 and definition of $[\![\, \_ \,]\!]$ (Def. A.4).

$\langle 3\rangle 6.$ Choose $\langle act_{2.1}, \ldots, act_{2.k}\rangle \in |Q|$ and $as_{2.1} \in (\!|\, act_{2.1} \,|\!), \ldots, as_{2.k} \in (\!|\, act_{2.k} \,|\!)$ such that $as_2 = as_{2.1} \frown \cdots \frown as_{2.k}$

PROOF: By $\langle 3\rangle 5$, definition of $|\_|$ (Def. D.1) and definition of $(\!|\, \_ \,|\!)$ (Def. A.3).

$\langle 3\rangle 7.$ Choose $as'_2 \in (\!|\, T'_{EL_2}(act_{2.1}) \,|\!) \frown \cdots \frown (\!|\, T'_{EL_2}(act_{2.k}) \,|\!)$ such that $as'_2 \in \mathcal{A}$ and $io(as'_2) = io(as'_1)$

PROOF: By assumption 1 and 1.1, $\langle 3\rangle 1$, $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 3\rangle 4$, $\langle 3\rangle 5$, $\langle 3\rangle 6$, and Lemma T.B.1.3.

$\langle 3\rangle 8.$ $as'_2 \in (\!|\, T'_{EL_2}(Q) \,|\!) \cap \mathcal{A}$

PROOF: By $\langle 3\rangle 6$, $\langle 3\rangle 7$, and definition of $(\!|\, \_ \,|\!)$ (Def. A.3) and definition of T'_{EL_2} (Def. B.3).

$\langle 3\rangle 9.$ Q.E.D.

PROOF: By $\langle 3\rangle 1$, $\langle 3\rangle 7$, $\langle 3\rangle 8$ and definition of (Def. A.4).

$\langle 2\rangle 2.$ ASSUME: 1.1 $s' \in [\![\, T'_{EL_2}(Q) \,]\!]$
PROVE: $s' \in [\![\, T'_{EL_1}(P) \,]\!]$

PROOF:By $\langle 2\rangle 1$ and symmetry of $=$.

$\langle 2\rangle 3.$ Q.E.D.

PROOF:By $\langle 2\rangle 1$ and $\langle 2\rangle 2$.

$\langle 1\rangle 2.$ Q.E.D.

PROOF:By $\langle 1\rangle 1$ and definition of T_{EL} (Def.B.3).

Lemma T.B.1.1 If

- $as' \in (\!|\, T'_{EL}(P) \,|\!)$

then

- $\exists \langle act_1, \ldots, act_n\rangle \in |P| : \exists as'_1 \in (\!|\, T'_{EL}(act_1) \,|\!) : \cdots : \exists as'_n \in (\!|\, T'_{EL}(act_n) \,|\!) : as' = as'_1 \frown \cdots as'_n$

Proof of Lemma T.B.1.1

ASSUME: 1. $EL \in \mathbf{EL}$ and $P \in \mathbf{P}$

PROVE: $as' \in (\!|\, T'_{EL}(P) \,|\!) \implies (\exists \langle act_1, \ldots, act_n\rangle \in |P| : \exists as'_1 \in (\!|\, T'_{EL}(act_1) \,|\!) : \cdots : \exists as'_n \in (\!|\, T'_{EL}(act_n) \,|\!) : as' = as'_1 \frown \cdots as'_n)$

$\langle 1\rangle 1.$ CASE: 1.1 $P = act$

$\langle 2\rangle 1.$ ASSUME: 2.1 $as' \in (\!|\, T'_{EL}(act) \,|\!)$
PROVE: $\exists \langle act_1, \ldots, act_n\rangle \in |act| : \exists as'_1 \in (\!|\, T'_{EL}(act_1) \,|\!) : \cdots : \exists as'_n \in (\!|\, T'_{EL}(act_n) \,|\!) : as' = as'_1 \frown \cdots as'_n$

$\langle 3\rangle 1.$ Choose $\langle act_1\rangle \in |act|$
PROOF: By definition of $|\_|$ (Def. D.1).
$\langle 3\rangle 2.$ $as' \in (\!| T'_{EL}(act_1) |\!)$
PROOF: By $\langle 3\rangle 1$ and assumption 2.1.
$\langle 3\rangle 3.$ Q.E.D.
PROOF: By $\langle 3\rangle 1$ and $\langle 3\rangle 2$.
$\langle 2\rangle 2.$ Q.E.D.
PROOF: By $\langle 2\rangle 1$ and logical implication.
$\langle 1\rangle 2.$ CASE: 1.1 $P = P_1.P_2$
1.2 $as' \in (\!| T'_{EL}(P_1) |\!) \implies (\exists \langle act_1, \ldots, act_n\rangle \in |P_1| : \exists as'_1 \in (\!| T'_{EL}(act_1) |\!) : \cdots : \exists as'_n \in (\!| T'_{EL}(act_n) |\!) : as' = as'_1 \frown \cdots as'_n)$
1.3 $as' \in (\!| T'_{EL}(P_2) |\!) \implies (\exists \langle act_1, \ldots, act_n\rangle \in |P_2| : \exists as'_1 \in (\!| T'_{EL}(act_1) |\!) : \cdots : \exists as'_n \in (\!| T'_{EL}(act_n) |\!) : as' = as'_1 \frown \cdots as'_n)$
$\langle 2\rangle 1.$ ASSUME: 2.1 $as' \in (\!| T'_{EL}(P_1.P_2) |\!)$
PROVE: $\exists \langle act_1, \ldots, act_n\rangle \in |P_1.P_2| : \exists as'_1 \in (\!| T'_{EL}(act_1) |\!) : \cdots : \exists as'_n \in (\!| T'_{EL}(act_n) |\!) : as' = as'_1 \frown \cdots as'_n$
$\langle 3\rangle 1.$ Choose $as'_1 \in (\!| T'_{EL}(P_1) |\!)$ and $as'_2 \in (\!| T'_{EL}(P_2) |\!)$ such that $as' = as'_1 \frown as'_2$
PROOF: By assumption 2.1, definition of T'_{EL} (Def. B.3), and definition of $(\!| \_ |\!)$ (Def. A.3).
$\langle 3\rangle 2.$ Choose $\langle act_{1.1}, \ldots, act_{1.j}\rangle \in |P_1|$ and $as'_{1.1} \in (\!| T'_{EL}(act_{1.1}) |\!), \ldots, as'_{1.j} \in (\!| T'_{EL}(act_{1.j}) |\!)$ such that $as'_1 = as'_{1.1} \frown \cdots \frown as'_{1.j}$
PROOF: By $\langle 3\rangle 1$ and assumption 1.2.
$\langle 3\rangle 3.$ Choose $\langle act_{2.1}, \ldots, act_{2.k}\rangle \in |P_2|$ and $as'_{2.1} \in (\!| T'_{EL}(act_{2.1}) |\!), \ldots, as'_{2.k} \in (\!| T'_{EL}(act_{2.k}) |\!)$ such that $as'_2 = as'_{2.1} \frown \cdots \frown as'_{2.k}$
PROOF: By $\langle 3\rangle 1$ and assumption 1.3.
$\langle 3\rangle 4.$ $\langle act_{1.1}, \ldots, act_{1.j}, act_{2.1}, \ldots, act_{2.k}\rangle \in |P_1.P_2|$
PROOF: By $\langle 3\rangle 2$, $\langle 3\rangle 3$ and definition of $|\_|$ (Def. D.1).
$\langle 3\rangle 5.$ Q.E.D.
PROOF: By $\langle 3\rangle 1$ - $\langle 3\rangle 4$.
$\langle 2\rangle 2.$ Q.E.D.
PROOF: By $\langle 2\rangle 1$ and logical implication.
$\langle 1\rangle 3.$ CASE: 1.1 $P = P_1 + P_2$
1.2 $as' \in (\!| T'_{EL}(P_1) |\!) \implies (\exists \langle act_1, \ldots, act_n\rangle \in |P_1| : \exists as'_1 \in (\!| T'_{EL}(act_1) |\!) : \cdots : \exists as'_n \in (\!| T'_{EL}(act_n) |\!) : as' = as'_1 \frown \cdots as'_n)$
1.3 $as' \in (\!| T'_{EL}(P_2) |\!) \implies (\exists \langle act_1, \ldots, act_n\rangle \in |P_2| : \exists as'_1 \in (\!| T'_{EL}(act_1) |\!) : \cdots : \exists as'_n \in (\!| T'_{EL}(act_n) |\!) : as' = as'_1 \frown \cdots as'_n)$
$\langle 2\rangle 1.$ ASSUME: 2.1 $as' \in (\!| T'_{EL}(P_1 + P_2) |\!)$
PROVE: $\exists \langle act_1, \ldots, act_n\rangle \in |P_1 + P_2| : \exists as'_1 \in (\!| T'_{EL}(act_1) |\!) : \cdots : \exists as'_n \in (\!| T'_{EL}(act_n) |\!) : as' = as'_1 \frown \cdots as'_n$
$\langle 3\rangle 1.$ Choose $i \in \{1, 2\}$ such that $as' \in (\!| T'_{EL}(P_i) |\!)$
PROOF: By assumption 2.1 and definition of T'_{EL} (Def. B.3), and definition of $(\!| \; |\!)$ (Def. A.3).
$\langle 3\rangle 2.$ Choose $\langle act_1, \ldots, act_n\rangle \in |P_i|$ and $as'_1 \in (\!| T'_{EL}(act_1) |\!), \cdots, as'_n \in (\!| T'_{EL}(act_n) |\!)$ such that $as' = as'_1 \frown \cdots as'_n$
PROOF: By $\langle 3\rangle 1$ and assumption 1.2 or assumption 1.3.
$\langle 3\rangle 3.$ $\langle act_1, \ldots, act_n\rangle \in |P_1 + P_2|$
PROOF: By $\langle 3\rangle 2$ and $|\_|$ (Def. D.1).
$\langle 3\rangle 4.$ Q.E.D.
PROOF: By $\langle 3\rangle 1$ - $\langle 3\rangle 3$.

$\langle 2\rangle 2$. Q.E.D.
Proof: By $\langle 2\rangle 1$ and logical implication.

$\langle 1\rangle 4$. Case: 1.1 $P = P_1^*$
1.2 $as' \in (\!| T'_{EL}(P_1) |\!) \implies (\exists \langle act_1, \ldots, act_n\rangle \in |P_1| : \exists as'_1 \in (\!| T'_{EL}(act_1) |\!) : \cdots : \exists as'_n \in (\!| T'_{EL}(act_n) |\!) : as' = as'_1 \frown \cdots as'_n)$

$\langle 2\rangle 1$. Assume: 2.1 $as' \in (\!| T'_{EL}(P_1^*) |\!)$
Prove: $\exists \langle act_1, \ldots, act_n\rangle \in |P_1^*| : \exists as'_1 \in (\!| T'_{EL}(act_1) |\!) : \cdots : \exists as'_n \in (\!| T'_{EL}(act_n) |\!) : as' = as'_1 \frown \cdots as'_n$

$\langle 3\rangle 1$. Choose $j \in \mathbb{N} \setminus \{0\}$ such that $as' \in (\!| T'_{EL}(P_1) |\!)^j$
Proof: By assumption 2.1 and definition of T'_{EL} (Def. B.3), and definition of $(\!| \; |\!)$ (Def. A.3).

$\langle 3\rangle 2$. Case: 3.1 $j = 1$

$\langle 4\rangle 1$. $as' \in (\!| T'_{EL}(P_1) |\!)$
Proof: By $\langle 3\rangle 1$, and assumption 3.1.

$\langle 4\rangle 2$. Choose $\langle act_1, \ldots, act_n\rangle \in |P_1|$ and $as'_1 \in (\!| T'_{EL}(act_1) |\!), \ldots, as'_n \in (\!| T'_{EL}(act_n) |\!)$ such that $as' = as'_1 \frown \cdots \frown as'_n$
Proof: By $\langle 4\rangle 1$ and assumption 1.2.

$\langle 4\rangle 3$. $\langle act_1, \ldots, act_n\rangle \in |P_1^*|$
Proof: By $\langle 4\rangle 2$ and definition of $|\_|$ (Def. D.1).

$\langle 4\rangle 4$. Q.E.D.
Proof: By $\langle 4\rangle 2$ and $\langle 4\rangle 3$.

$\langle 3\rangle 3$. Case: 3.1 $j = j' + 1$
3.2 $\forall i \le j' : as' \in (\!| T'_{EL}(P_1) |\!)^i \implies \exists \langle act_1, \ldots, act_n\rangle \in |P_1|^i : \exists as'_1 \in (\!| T'_{EL}(act_1) |\!) : \cdots : \exists as'_n \in (\!| T'_{EL}(act_n) |\!) : as' = as'_1 \frown \cdots as'_n$

$\langle 4\rangle 1$. Choose $aas'_1 \in (\!| T'_{EL}(P_1) |\!)^{j'}$ and $aas'_2 \in (\!| T'_{EL}(P_1) |\!)$ such that $as' = aas'_1 \frown aas'_2$
Proof: By $\langle 3\rangle 1$, assumption 3.1,definition of T'_{EL} (Def. B.3), and definition of $(\!| \; |\!)$ (Def. A.3).

$\langle 4\rangle 2$. Choose $\langle act_{1.1}, \ldots, act_{1.m}\rangle \in |P_1|^{j'}$ and $as'_{1.1} \in (\!| T'_{EL}(act_{1.1}) |\!), \ldots, as'_{1.m} \in (\!| T'_{EL}(act_{1.m}) |\!)$ such that $aas'_1 = as'_{1.1} \frown \cdots \frown as'_{1.m}$
Proof: By $\langle 4\rangle 1$ and assumption 3.2.

$\langle 4\rangle 3$. Choose $\langle act_{2.1}, \ldots, act_{2.n}\rangle \in |P_1|$ and $as'_{2.1} \in (\!| T'_{EL}(act_{2.1}) |\!), \ldots, as'_{2.n} \in (\!| T'_{EL}(act_{2.n}) |\!)$ such that $aas'_2 = as'_{2.1} \frown \cdots \frown as'_{2.n}$
Proof: By $\langle 4\rangle 1$ and assumption 1.2.

$\langle 4\rangle 4$. $\langle act_{1.1}, \ldots, act_{1.m}, act_{2.1}, \ldots, act_{2.n}\rangle \in |P_1^*|$
Proof: By $\langle 4\rangle 2$, $\langle 4\rangle 3$ and definition of $|\_|$ (Def. D.1).

$\langle 4\rangle 5$. Q.E.D.
Proof: By $\langle 4\rangle 2$ - $\langle 4\rangle 4$.

$\langle 3\rangle 4$. Q.E.D.
Proof: By $\langle 3\rangle 2$ and $\langle 3\rangle 3$.

$\langle 2\rangle 2$. Q.E.D.
Proof: By $\langle 2\rangle 1$ and logical implication.

$\langle 1\rangle 5$. Q.E.D.
Proof: By $\langle 1\rangle 1$ - $\langle 1\rangle 4$ and definition of **P** (Def. A.1).

Lemma T.B.1.2 If

- $var(EL) \cap (var(act_1) \cup \cdots \cup var(act_n)) = \emptyset$
- $as' \in \mathcal{A}$

- $as'_1 \in (\!| T'_{EL}(act_1) |\!), \dots, as'_n \in (\!| T'_{EL}(act_n) |\!)$
- $as' = as'_1 \frown \cdots \frown as'_n$

then

- $\exists as \in \mathcal{A} : \exists as_1 \in (\!| act_1 |\!) : \cdots : \exists as_n \in (\!| act_n |\!) : as = as_1 \frown \cdots \frown as_n \wedge (\forall i \in \{1, \dots, n\} : preState(as_i, var(EL)) = preState(as'_i, var(EL)) \wedge postState(as_i, var(EL)) = postState(as'_i, var(EL)))$

Proof of Lemma T.B.1.2 The proof is by induction on the length of the action sequence $(act_1, \dots, act_n)$. In the proof we make use of the following definition which high-lights the induction.

$$
\begin{array}{l}
Ind((act_1, \dots, act_n), EL) \overset{\text{def}}{=} \\
\quad \forall as' \in \mathcal{A} : \forall as'_1 \in (\!| T'_{EL}(act_1) |\!) : \cdots : \forall as'_n \in (\!| T'_{EL}(act_n) |\!) : \\
\quad\quad as' = as'_1 \frown \cdots \frown as'_n \\
\quad\quad\quad \Longrightarrow \exists as \in \mathcal{A} : \\
\quad\quad\quad\quad\quad \exists as_1 \in (\!| act_1 |\!) : \cdots \exists as_n \in (\!| act_n |\!) : \\
\quad\quad\quad\quad\quad\quad \wedge\, as = as_1 \frown \cdots \frown as_n \\
\quad\quad\quad\quad\quad\quad \wedge\, \forall i \in \{1, \dots, n\} : \\
\quad\quad\quad\quad\quad\quad\quad \wedge\, preState(as_i, var(EL)) = preState(as'_i, var(EL)) \\
\quad\quad\quad\quad\quad\quad\quad \wedge\, postState(as_i, var(EL)) = postState(as'_i, var(EL))
\end{array}
$$

Also, if s is a sequence, we denote by $pref(s, j)$, the prefix of s of length j. If j is greater than the length of s, then $pref(s, j) = s$.

The proof is given by the following.

ASSUME: 1. $var(EL) \cap (var(act_1) \cup \cdots \cup var(act_n)) = \emptyset$ for $EL \in \mathbf{EL}$ and $act_1, \dots, act_n \in \mathbf{Act}$ and $n \geq 1$

PROVE: $Ind((act_1, \dots, act_n), EL)$

$\langle 1 \rangle 1.$ CASE: 1.1 $n = 1$

  $\langle 2 \rangle 1.$ ASSUME: 2.1 $as' \in \mathcal{A}$

            2.2 $as'_1 \in (\!| T'_{EL}(act_1) |\!)$

            2.3 $as' = as'_1$

    PROVE: $\exists as \in \mathcal{A} : \exists as_1 \in (\!| act_1 |\!) : as = as_1 \wedge preState(as_1, var(EL)) = preState(as'_1, var(EL)) \wedge postState(as_1, var(EL)) = postState(as'_1, var(EL)))$

    $\langle 3 \rangle 1.$ CASE: 3.1 $act_1 = (\epsilon, bx_\epsilon, \epsilon, a_\epsilon)$ for $bx_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$ and $a_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

      $\langle 4 \rangle 1.$ Choose $as \in (\!| act_1 |\!)$ such that $as = as'$

        $\langle 5 \rangle 1.$ $(\!| T'_{EL}(act_1) |\!) = (\!| tact_{EL}(act_1) |\!) = (\!| act_1 |\!)$

          PROOF: By assumption 3.1, definition of T'_{EL} (Def. B.3), and definition of $tact_{EL}$ (Eq. (15)), definition of te_{EL} (Eq.14), and definition of $\mathbf{E}$ (App. A.1.1).

        $\langle 5 \rangle 2.$ Q.E.D.

          PROOF: By assumption 2.2, assumption 2.3, and $\langle 5 \rangle 1$.

      $\langle 4 \rangle 2.$ $as \in \mathcal{A}$

        PROOF: By $\langle 4 \rangle 1$ and assumption 2.1.

      $\langle 4 \rangle 3.$ $preState(as, var(EL)) = preState(as', var(EL))$ and $postState(as, var(EL)) = postState(as', var(EL))$

        PROOF: By $\langle 4 \rangle 1$.

      $\langle 4 \rangle 4.$ Q.E.D.

PROOF: By ⟨4⟩1, ⟨4⟩2, and ⟨4⟩3.

⟨3⟩2. CASE: 3.1 $act_1 = (ie, bx_\epsilon, \epsilon, a_\epsilon)$ for $ie \in \mathbf{IE}$, $bx_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

⟨4⟩1. CASE: 4.1 $ie \in \mathcal{D}om(te_{EL})$

⟨5⟩1. Choose $aas'_1 \in (\!| (\epsilon, bx_\epsilon, \epsilon, \epsilon) |\!)$, $aas'_2 \in (\!| te_{EL}(ie) |\!)$, and $aas'_3 \in (\!| (\epsilon, \epsilon, \epsilon, a_\epsilon) |\!)$ such that $as'_1 = aas'_1 \frown aas'_2 \frown aas'_3$

PROOF: By assumption 2.2, assumption 3.1, assumption 4.1, and definition of T'_{EL} (Def. B.3), and $(\!| \_ |\!)$ (Def. A.3).

⟨5⟩2. Choose $as \in (\!| (ie, bx_\epsilon, \epsilon, a_\epsilon) |\!)$ such that $preState(as, var(EL)) = preState(aas'_1, var(EL))$ and $postState(as, var(EL)) = postState(aas'_3, var(EL))$

PROOF: By assumption 1 and ⟨5⟩1, definition of $(\!| \_ |\!)$ (Def .A.3), definition of te_{EL} (Eq. 14), and definition of $var(EL)$ (Section B.1).

⟨5⟩3. $as \in \mathcal{A}$

PROOF: By assumptions 2.1, 2.2, and 2.3, and ⟨5⟩1, ⟨5⟩2, and definition of $\mathcal{A}$ (Eq. 10).

⟨5⟩4. Q.E.D.

PROOF: By ⟨5⟩2, assumption 3.1, and ⟨5⟩3.

⟨4⟩2. CASE: 4.1 $ie \notin \mathcal{D}om(te_{EL})$

⟨5⟩1. Choose $as \in (\!| act_1 |\!)$ such that $as = as'$

⟨6⟩1. $(\!| T'_{EL}(act_1) |\!) = (\!| tact_{EL}(act_1) |\!) = (\!| act_1 |\!)$

PROOF: By assumption 3.1, assumption 4.1, definition of T'_{EL} (Def. B.3), definition of $tact_{EL}$ (Eq. (15)), definition of te_{EL} (Eq. 14), and definition of $\mathbf{E}$ (App. A.1.1).

⟨6⟩2. Q.E.D.

PROOF: By assumption 2.2, assumption 2.3, and ⟨6⟩1.

⟨5⟩2. $as \in \mathcal{A}$

PROOF: By ⟨5⟩1 and assumption 2.1.

⟨5⟩3. $preState(as, var(EL)) = preState(as', var(EL))$ and $postState(as, var(EL)) = postState(as')var(EL)$

PROOF: By ⟨5⟩2.

⟨5⟩4. Q.E.D.

PROOF: By ⟨5⟩1, ⟨5⟩2, and ⟨5⟩3.

⟨4⟩3. Q.E.D.

PROOF: By ⟨4⟩1 and ⟨4⟩2.

⟨3⟩3. CASE: 3.1 $act_1 = (\epsilon, bx_\epsilon, oe, a_\epsilon)$ for $oe \in \mathbf{OE}$, $bx_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

PROOF: Proof is the same as for ⟨3⟩2.

⟨3⟩4. Q.E.D.

PROOF: By ⟨3⟩1 - ⟨3⟩3 and definition of **Act** (Eq. (7)) and (Eq. (8)).

⟨2⟩2. Q.E.D.

PROOF: By ⟨2⟩1 and logical implication.

⟨1⟩2. CASE: 1.1 $n = n' + 1$
1.2 $\forall j \in \mathbb{N} \setminus \{0\} : j \leq n' \implies Ind(pref((act_1, \ldots, act_n), j), EL)$

⟨2⟩1. ASSUME: 2.1 $as' \in \mathcal{A}$
2.2 $as'_1 \in (\!| T'_{EL}(act_1) |\!) \wedge \cdots \wedge as'_n \in (\!| T'_{EL}(act_n) |\!)$
2.3 $as' = as'_1 \frown \cdots \frown as'_n$

PROVE: $\exists as \in \mathcal{A} : \exists as_1 \in (\!| act_1 |\!) : \cdots : \exists as_n \in (\!| act_n |\!) : as = as_1 \frown \cdots \frown as_n \wedge (\forall i \in \{1, \ldots, n\} : preState(as_i, var(EL)) = preState(as'_i, var(EL)) \wedge postState(as_i, var(EL)) = postState(as'_i, var(EL)))$

⟨3⟩1. Choose $aas' \in \mathcal{A}$ such that $as' = aas' \frown as'_n$

PROOF: By assumption 2.1, assumption 2.3, and definition of $\mathcal{A}$ (Eq. (10)).

$\langle 3\rangle 2$. Choose $aas \in \mathcal{A}$ and $as_1 \in (\!|\, act_1 \,|\!), \ldots, as_{n-1} \in (\!|\, act_{n-1} \,|\!)$ such that $aas = as_1 \frown \cdots \frown as_{n-1}$ and $\forall i \in \{1, \ldots, n-1\} : preState(as_i, var(EL)) = preState(as'_i, var(EL)) \wedge postState(as_i, var(EL)) = postState(as'_i, var(EL))$

PROOF: By $\langle 3\rangle 1$ and assumptions 1.1, 1.2, 2.2, and 2.3.

$\langle 3\rangle 3$. Choose $as_n \in (\!|\, act_n \,|\!)$ such that $aas \frown as_n \in \mathcal{A}$, $preState(as_n, var(EL)) = postState(aas, var(EL))$, $preState(as_n, var(EL)) = preState(as'_n, var(EL))$, and $postState(as_n, var(EL)) = postState(as'_n, var(EL))$

$\langle 4\rangle 1$. Choose $as_n \in (\!|\, act_n \,|\!)$ such that $preState(as_n, var(EL)) = postState(aas, var(EL))$

PROOF: By $\langle 3\rangle 2$ and definition of $(\!|\, \_ \,|\!)$ (Def. A.3).

$\langle 4\rangle 2$. $preState(as_n, var(EL)) = preState(as'_n, var(EL))$

$\langle 5\rangle 1$. $postState(aas, var(EL)) = postState(as'_{n-1}, var(EL))$

PROOF: By $\langle 3\rangle 2$ and definition of $postState(\_, \_)$ (Def. D.2).

$\langle 5\rangle 2$. $postState(as'_{n-1}, var(EL)) = preState(as'_n, EL)$

PROOF: By assumptions 2.1 - 2.3 and definition of $\mathcal{A}$ (Eq. (10)).

$\langle 5\rangle 3$. Q.E.D.

PROOF: By $\langle 4\rangle 1$, $\langle 5\rangle 1$, and $\langle 5\rangle 2$.

$\langle 4\rangle 3$. $postState(as_n, var(EL)) = postState(as'_n, var(EL))$

$\langle 5\rangle 1$. CASE: 3.1 $act_n = (\epsilon, bx_\epsilon, \epsilon, a_\epsilon)$ for $bx_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$ and $a_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

$\langle 6\rangle 1$. $as'_n \in (\!|\, act_n \,|\!) = (\!|\, T'_{EL}(act_n) \,|\!)$

PROOF: By assumption 2.2, assumption 3.1, and definition of T'_{EL} (Def. B.3).

$\langle 6\rangle 2$. Q.E.D.

PROOF: By assumption 1, $\langle 4\rangle 1$, $\langle 4\rangle 2$, $\langle 6\rangle 1$, and definition of $(\!|\, \_ \,|\!)$ (Def. A.3).

$\langle 5\rangle 2$. CASE: 3.1 $act_n = (ie, bx_\epsilon, \epsilon, a_\epsilon)$ for $ie \in \mathbf{IE}$, $bx_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

$\langle 6\rangle 1$. CASE: 4.1 $ie \in \mathcal{D}om(te_{EL})$

$\langle 7\rangle 1$. Choose $aas'_1 \in (\!|\, (\epsilon, bx_\epsilon, \epsilon, \epsilon) \,|\!)$, $aas'_2 \in (\!|\, te_{EL}(ie) \,|\!)$, and $aas'_3 \in (\!|\, (\epsilon, \epsilon, \epsilon, a_\epsilon) \,|\!)$ such that $as'_n = aas'_1 \frown aas'_2 \frown aas'_3$

PROOF: By assumption 2.2, assumption 3.1, assumption 4.1, and definition of T'_{EL} (Def. B.3).

$\langle 7\rangle 2$. $postState(as_n, var(EL)) = postState(aas'_3, var(EL))$

PROOF: By assumption 1, assumption 3.1, $\langle 4\rangle 1$, $\langle 4\rangle 2$, $\langle 7\rangle 1$, and definition of $(\!|\, \_ \,|\!)$ (Def. A.3).

$\langle 7\rangle 3$. Q.E.D.

PROOF: By $\langle 7\rangle 1$ and $\langle 7\rangle 2$.

$\langle 6\rangle 2$. CASE: 4.1 $ie \notin \mathcal{D}om(te_{EL})$

$\langle 7\rangle 1$. $as'_n \in (\!|\, act_n \,|\!) = (\!|\, T'_{EL}(act_n) \,|\!)$

PROOF: By assumption 2.2, assumption 3.1, and definition of T'_{EL} (Def. B.3).

$\langle 7\rangle 2$. Q.E.D.

PROOF: By assumption 1, $\langle 4\rangle 1$, $\langle 4\rangle 2$, $\langle 7\rangle 1$, and definition of $(\!|\, \_ \,|\!)$ (Def. A.3).

$\langle 6\rangle 3$. Q.E.D.

PROOF: By $\langle 6\rangle 1$ and $\langle 6\rangle 2$.

$\langle 5\rangle 3$. CASE: 3.1 $act_n = (\epsilon, bx_\epsilon, oe, a_\epsilon)$ for $oe \in \mathbf{OE}$, $bx_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

Proof: Proof is the same as for $\langle 5\rangle 2$.

$\langle 5\rangle 4$. Q.E.D.

Proof: By $\langle 5\rangle 1$ - $\langle 5\rangle 3$ and definition of **Act** (Eq. (7)) and (Eq. (8)).

$\langle 4\rangle 4$. $aas \frown as_n \in \mathcal{A}$

Proof: By assumption 2.1 - 2.3, $\langle 3\rangle 2$, $\langle 4\rangle 1$, $\langle 4\rangle 2$, assumption 1, definition of T'_{EL} (Def. B.3), and definition of $\mathcal{A}$ (Def. 10).

$\langle 4\rangle 5$. Q.E.D.

Proof: By $\langle 4\rangle 1$ - $\langle 4\rangle 4$.

$\langle 3\rangle 4$. Q.E.D.

Proof: By $\langle 3\rangle 1$, $\langle 3\rangle 2$ and $\langle 3\rangle 3$.

$\langle 2\rangle 2$. Q.E.D.

Proof: By $\langle 2\rangle 1$ and logical implication.

$\langle 1\rangle 3$. Q.E.D.

Proof: By $\langle 1\rangle 1$ and $\langle 1\rangle 2$.

Lemma T.B.1.3 If

- $EL_1 = rnm(P, EL)$ and $EL_2 = rnm(Q, EL)$ for some $P, Q \in \mathbf{P}$ and $EL, EL_1, EL_2 \in \mathbf{EL}$
- $\langle act_{1.1}, \ldots, act_{1.j}\rangle \in pr(|P|)$
- $as'_{1.1} \in (\!| T'_{EL_1}(act_{1.1}) |\!) \wedge \cdots \wedge as'_{1.j} \in (\!| T'_{EL_1}(act_{1.j}) |\!)$
- $as'_1 \in \mathcal{A}$ and $as'_1 = as'_{1.1} \frown \cdots \frown as'_{1.j}$
- $as_{1.1} \in (\!| act_{1.1} |\!) \wedge \cdots \wedge as_{1.j} \in (\!| act_{1.j} |\!)$
- $\forall i \in \{1, \ldots, j\} : (preState(as_{1.i}, var(EL_1)) = preState(as'_{1.i}, var(EL_1)))\wedge$ $(postState(as_{1.i}, var(EL_1)) = postState(as'_{1.i}, var(EL_1)))$
- $as_1 \in pr((\!| P |\!)) \cap \mathcal{A}$ and $as_1 = as_{1.1} \frown \cdots \frown as_{1.j}$
- $as_2 \in pr((\!| Q |\!)) \cap \mathcal{A}$ and $io(as_2) = io(as_1)$
- $\langle act_{2.1}, \ldots, act_{2.k}\rangle \in pr(|Q|)$
- $as_{2.1} \in (\!| act_{2.1} |\!), \ldots, as_{2.k} \in (\!| act_{2.k} |\!)$
- $as_2 = as_{2.1} \frown \cdots \frown as_{2.k}$

then

- $\exists as'_2 \in (\!| T'_{EL_2}(act_{2.1}) |\!) \frown \cdots \frown (\!| T'_{EL_2}(act_{2.k}) |\!) : as'_2 \in \mathcal{A} \wedge io(as'_2) = io(as'_1)$

Proof of Lemma T.B.1.3 The proof is by induction on the sum j and k. In the proof, we make use of the following definition which high-lights the induction.

$$
\begin{aligned}
&Ind(P,Q,EL_1,EL_2,n) \overset{\text{def}}{=}\\
&\quad \forall j,k \in \mathbb{N}\setminus\{0\}:\\
&\quad\quad \forall\langle act_{1.1},\ldots,act_{1.j}\rangle \in pr(|P|):\\
&\quad\quad\quad \forall as'_{1.1} \in (\!|\, T'_{EL_1}(act_{1.1}) \,|\!) : \cdots \forall as'_{1.j} \in (\!|\, T'_{EL_1}(act_{1.j}) \,|\!):\\
&\quad\quad\quad\quad \forall as_{1.1} \in (\!|\, act_{1.1} \,|\!) : \cdots \forall as_{1.j} \in (\!|\, act_{1.j} \,|\!):\\
&\quad\quad\quad\quad\quad \forall\langle act_{2.1},\ldots,act_{2.k}\rangle \in pr(|Q|):\\
&\quad\quad\quad\quad\quad\quad \forall as_{2.1} \in (\!|\, act_{2.1} \,|\!) : \cdots : \forall as_{2.k} \in (\!|\, act_{2.k} \,|\!):\\
&\quad\quad\quad\quad\quad\quad\quad \forall as'_1 \in \mathcal{A} : \forall as_1 \in pr((\!|\, P \,|\!)) \cap \mathcal{A} : \forall as_2 \in pr((\!|\, Q \,|\!)) \cap \mathcal{A}:\\
&\quad\quad\quad\quad\quad\quad\quad\quad \wedge\, j+k=n\\
&\quad\quad\quad\quad\quad\quad\quad\quad \wedge\, as'_1 = as'_{1.1}\frown\cdots\frown as'_{1.j}\\
&\quad\quad\quad\quad\quad\quad\quad\quad \wedge\, as_1 = as_{1.1}\frown\cdots\frown as_{1.j}\\
&\quad\quad\quad\quad\quad\quad\quad\quad \wedge\, as_2 = as_{2.1}\frown\cdots\frown as_{2.k}\\
&\quad\quad\quad\quad\quad\quad\quad\quad \wedge\, io(as_1) = io(as_2)\\
&\quad\quad\quad\quad\quad\quad\quad\quad \wedge\, \forall i \in \{1,\ldots,j\}:\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge\, preState(as_{1.i}, var(EL_1)) = preState(as'_{1.i}, var(EL_1))\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge\, postState(as_{1.i}, var(EL_1)) = postState(as'_{1.i}, var(EL_1))\\
&\quad\quad\quad\quad\quad\quad\quad\quad \implies \exists as'_2 \in (\!|\, T'_{EL_2}(act_{2.1}) \,|\!)\frown\cdots\frown(\!|\, T'_{EL_2}(act_{2.k}) \,|\!):\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge\, as'_2 \in \mathcal{A}\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge\, io(as'_2) = io(as'_1)
\end{aligned}
$$

ASSUME: 1. $EL_1 = rnm(P, EL)$ and $EL_2 = rnm(Q, EL)$ for some $P, Q \in \mathbf{P}$ and $EL, EL_1, EL_2 \in \mathbf{EL}$
2. $n \geq 2$ for some $n \in \mathbb{N}$

PROVE: $Ind(P,Q,EL_1,EL_2,n)$

$\langle 1\rangle 1.$ CASE: 1.1 $n = 2$

$\langle 2\rangle 1.$ ASSUME: 2.1 $j = 1$ and $k = 1$ for $j,k \in \mathbb{N}\setminus\{0\}$
2.2 $\langle act_1\rangle \in pr(|P|)$
2.3 $as'_1 \in (\!|\, T'_{EL_1}(act_1) \,|\!)$
2.4 $as'_1 \in \mathcal{A}$
2.5 $as_1 \in (\!|\, act_1 \,|\!)$
2.6 $(preState(as_1, var(EL_1)) = preState(as'_1, var(EL_1))) \wedge (postState(as_1, var(EL_1)) = postState(as'_1, var(EL_1)))$
2.7 $as_1 \in pr((\!|\, P \,|\!)) \cap \mathcal{A}$
2.8 $as_2 \in pr((\!|\, Q \,|\!)) \cap \mathcal{A}$ and $io(as_2) = io(as_1)$
2.9 $\langle act_2\rangle \in pr(|Q|)$
2.10 $as_2 \in (\!|\, act_2 \,|\!)$

PROVE: $\exists as'_2 \in (\!|\, T'_{EL_2}(act_2) \,|\!) : as'_2 \in \mathcal{A} \wedge io(as'_2) = io(as'_1)$

$\langle 3\rangle 1.$ CASE: 3.1 $act_1 = (\epsilon, bx_\epsilon, \epsilon, a_\epsilon)$ for $bx_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$ and $a_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

$\langle 4\rangle 1.$ CASE: 4.1 $act_2 = (\epsilon, bx'_\epsilon, \epsilon, a'_\epsilon)$ for $bx'_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$ and $a'_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

$\langle 5\rangle 1.$ Choose $as'_2 \in (\!|\, T'_{EL_2}(act_2) \,|\!)$ such that $as'_2 = as_2$

$\langle 6\rangle 1.$ $(\!|\, T'_{EL_2}(act_2) \,|\!) = (\!|\, tact_{EL_2}(act_2) \,|\!) = (\!|\, act_2 \,|\!)$

PROOF: By definition of T'_{EL_2} (Def. B.3), definition of $tact_{EL}$ (Eq. (15)), and assumption 4.1.

$\langle 6\rangle 2.$ Q.E.D.

PROOF: By $\langle 6\rangle 1$ and assumption 2.10.

⟨5⟩2. $as'_2 \in \mathcal{A}$
PROOF: By ⟨5⟩1 and assumption 2.8.
⟨5⟩3. $io(as'_2) = io(as'_1)$
⟨6⟩1. $io(as'_2) = \langle\rangle$
PROOF: By ⟨5⟩1, assumption 2.10, assumption 4.1, definition of $(\!|\ |\!)$ (Def. A.3) and definition of io (Eq. (11)).
⟨6⟩2. $io(as'_1) = \langle\rangle$
PROOF: By assumption 2.3, assumption 3.1, definition of T'_{EL_1} (Def. B.3), definition of $(\!|\ |\!)$ (Def. A.3) and definition of io (Eq. (11)).
⟨6⟩3. Q.E.D.
PROOF: By ⟨6⟩1 and ⟨6⟩2.
⟨5⟩4. Q.E.D.
PROOF: By ⟨5⟩1 - ⟨5⟩3.
⟨4⟩2. CASE: 4.1 $act_2 = (ie', bx'_\epsilon, \epsilon, a'_\epsilon)$ for $ie' \in \mathbf{IE} \cup \{\epsilon\}$, $bx'_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a_{\epsilon'} \in \mathbf{Assign} \cup \{\epsilon\}$
PROOF: Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that $io(as_2) \neq io(as_1)$ by definition of $(\!|\_|\!)$ (Def. A.3) and definition of io (Eq. (11)). Hence, assumption 4.1 cannot hold.
⟨4⟩3. CASE: 4.1 $act_2 = (\epsilon, bx'_\epsilon, oe', a'_\epsilon)$ for $oe' \in \mathbf{OE} \cup \{\epsilon\}$, $bx_{\epsilon'} \in \mathbf{BExp} \cup \{\epsilon\}$, and $a_{\epsilon'} \in \mathbf{Assign} \cup \{\epsilon\}$
PROOF: Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that $io(as_2) \neq io(as_1)$ by definition of $(\!|\_|\!)$ (Def. A.3) and definition of io (Eq. (11)). Hence, assumption 4.1 cannot hold.
⟨4⟩4. Q.E.D.
PROOF: By ⟨4⟩1 - ⟨4⟩3 and definition of **Act** (Eq. 7) and (Eq. 8).
⟨3⟩2. CASE: 3.1 $act_1 = (ie, bx_\epsilon, \epsilon, a_\epsilon)$ for $ie \in \mathbf{IE} \cup \{\epsilon\}$, $bx_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$
⟨4⟩1. CASE: 4.1 $act_2 = (\epsilon, bx'_\epsilon, \epsilon, a'_\epsilon)$
PROOF: Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that $io(as_2) \neq io(as_1)$ by definition of $(\!|\_|\!)$ (Def. A.3) and definition of io (Eq. (11)). Hence, assumption 4.1 cannot hold.
⟨4⟩2. CASE: 4.1 $act_2 = (ie', bx'_\epsilon, \epsilon, a'_\epsilon)$ for $ie' \in \mathbf{IE} \cup \{\epsilon\}$, $bx'_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a'_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$
⟨5⟩1. CASE: 5.1 $ie \in \mathcal{D}om(te_{EL_1})$ and $ie' \in \mathcal{D}om(te_{EL_2})$
⟨6⟩1. $\exists as'_2 \in (\!|\,(\epsilon, bx'_\epsilon, \epsilon, \epsilon).te_{EL_2}(ie').(\epsilon, \epsilon, \epsilon, a'_\epsilon)\,|\!) : as'_2 \in \mathcal{A} \wedge io(as'_1) = io(as'_2)$
⟨7⟩1. $\exists as'_2 \in (\!|\,(\epsilon, bx'_\epsilon, \epsilon, \epsilon)\,|\!) \frown (\!|\,te_{EL_2}(ie')\,|\!) \frown (\!|\,(\epsilon, \epsilon, \epsilon, a'_\epsilon)\,|\!) : as'_2 \in \mathcal{A} \wedge io(as'_1) = io(as'_2)$
⟨8⟩1. Choose $as'_{1.1} \in (\!|\,(\epsilon, bx_\epsilon, \epsilon, \epsilon)\,|\!)$, $as'_{1.2} \in (\!|\,te_{EL_1}(ie)\,|\!)$, and $as'_{1.3} \in (\!|\,(\epsilon, \epsilon, \epsilon, a_\epsilon)\,|\!)$ such that $as'_1 = as'_{1.1} \frown as'_{1.2} \frown as'_{1.3}$
⟨9⟩1. $(\!|\,T'_{EL_1}(act_1)\,|\!) = (\!|\,tact_{EL_1}(act_1)\,|\!) = (\!|\,(\epsilon, bx_\epsilon, \epsilon, \epsilon).te_{EL_1}(ie).(\epsilon, \epsilon, \epsilon, a_\epsilon)\,|\!)$
PROOF: By definition of T'_{EL_1} (Def. B.3), assumption 3.1, assumption 5.1, and definition of $tact_{EL}$ (Eq. (15)).
⟨9⟩2. $(\!|\,(\epsilon, bx_\epsilon, \epsilon, \epsilon).te_{EL_1}(ie).(\epsilon, \epsilon, \epsilon, a_\epsilon)\,|\!) = (\!|\,(\epsilon, bx_\epsilon, \epsilon, \epsilon)\,|\!) \frown (\!|\,te_{EL_1}(ie)\,|\!) \frown (\!|\,(\epsilon, \epsilon, \epsilon, a_\epsilon)\,|\!)$
PROOF: By definition of $(\!|\_|\!)$ (Def. A.3).
⟨9⟩3. Q.E.D.
PROOF: By ⟨9⟩1, ⟨9⟩2, and assumption 2.3.
⟨8⟩2. Choose variable renaming functions $vr_1, vr_2 \in \mathbf{Var} \rightarrow \mathbf{Var}$ such that $vr_1(EL) = EL_2$ and $vr_2(EL) = EL_2$ when

vr_1 and vr_2 are lifted to event libraries.

PROOF: By assumption 1.

$\langle 8\rangle 3.$ Choose $as'_{2.1} \in (\!|\, (\epsilon, bx'_\epsilon, \epsilon, \epsilon) \,|\!)$ such that $preState(as'_{2.1}, var(EL_2)) = preState(as_2, var(EL_2))$ and $preState(as'_{2.1})(vr_2(x)) = preState(as'_{1.1})(vr_1(x))$ for all $x \in var(EL)$

PROOF: By assumptions 1, 2.9, and 4.1, and $\langle 8\rangle 1$, $\langle 8\rangle 2$, definition of $(\!|\, \_ \,|\!)$ (Def. A.3), $preState(\_, \_)$, and $preState(\_, \_)$ (Def. D.2).

$\langle 8\rangle 4.$ $as'_{2.1} \in \mathcal{A}$

PROOF: By $\langle 8\rangle 3$ and assumption 2.8, and definition of $\mathcal{A}$ (Eq. (10)).

$\langle 8\rangle 5.$ $io(as'_{1.1}) = io(as'_{2.1}) = \langle\rangle$

PROOF: By $\langle 8\rangle 1$, $\langle 8\rangle 3$ and definition of $(\!|\, \_ \,|\!)$ (Def. A.3) and io (Eq. (11)).

$\langle 8\rangle 6.$ Choose $as'_{2.2} \in (\!|\, te_{EL_2}(ie') \,|\!)$ such that $preState(as'_{2.2}) = postState(as'_{2.1})$

PROOF: By $\langle 8\rangle 3$, assumption 5.1, definition of $(\!|\, \_ \,|\!)$ (Def. A.3), and definition of $preState(\_, \_)$ and $postState(\_, \_)$ (Def. D.2).

$\langle 8\rangle 7.$ $as'_{2.2} \in \mathcal{A}$

PROOF: By assumption 2.4, assumption 2.6, $\langle 8\rangle 1$, $\langle 8\rangle 3$, $\langle 8\rangle 6$, and definition of $\mathcal{A}$ (Eq. (10)).

$\langle 8\rangle 8.$ $io(as'_{2.2}) = io(as'_{1.2})$

PROOF: By assumption 2.6, assumption 2.8, $\langle 8\rangle 1$, $\langle 8\rangle 3$, $\langle 8\rangle 6$, and definition of io (Eq. (11)).

$\langle 8\rangle 9.$ Choose $as'_{2.3} \in (\!|\, (\epsilon, \epsilon, \epsilon, a'_\epsilon) \,|\!)$ such that $preState(as'_{2.3}) = postState(as'_{2.2})$

PROOF: By definition of $(\!|\, \_ \,|\!)$ (Def. A.3), and definition of $preState()$ and $postState()$ (Def. D.2).

$\langle 8\rangle 10.$ $as'_{2.3} \in \mathcal{A}$

PROOF: By $\langle 8\rangle 9$ and definition of $\mathcal{A}$ (Eq. (10)).

$\langle 8\rangle 11.$ $io(as'_{2.3}) = io(as'_{1.3})$

PROOF: By $\langle 8\rangle 1$, $\langle 8\rangle 9$, definition of $(\!|\, \_ \,|\!)$ (Def. A.3), and definition of io (Eq. (11)).

$\langle 8\rangle 12.$ Choose $as'_2 \in \mathcal{A}$ such that $as'_2 = as'_{2.1} \frown as'_{2.2} \frown as'_{2.3}$

PROOF: By $\langle 8\rangle 4$, $\langle 8\rangle 6$, $\langle 8\rangle 7$, $\langle 8\rangle 9$, $\langle 8\rangle 10$, and definition of $\mathcal{A}$ (Eq. (10)).

$\langle 8\rangle 13.$ $io(as'_1) = io(as'_2)$

PROOF: By $\langle 8\rangle 1$, $\langle 8\rangle 5$, $\langle 8\rangle 8$, $\langle 8\rangle 11$, $\langle 8\rangle 12$, and definition of io (Eq. (11)).

$\langle 8\rangle 14.$ Q.E.D.

PROOF: By $\langle 8\rangle 3$, $\langle 8\rangle 6$, $\langle 8\rangle 9$, $\langle 8\rangle 12$ and $\langle 8\rangle 13$.

$\langle 7\rangle 2.$ Q.E.D.

PROOF: By $\langle 7\rangle 1$ and definition $(\!|\, \_ \,|\!)$ (Def. A.3).

$\langle 6\rangle 2.$ Q.E.D.

PROOF: By assumption 5.1, $\langle 6\rangle 1$, and definition of T'_{EL_2} (Def. B.3).

$\langle 5\rangle 2.$ CASE: 5.1 $ie \notin \mathcal{D}om(te_{EL_1})$ and $ie' \notin \mathcal{D}om(te_{EL_2})$

$\langle 6\rangle 1.$ Choose $as'_2 \in (\!|\, T'_{EL_2}(act_2) \,|\!)$ such that $as'_2 = as_2$

$\langle 7\rangle 1.$ $(\!|\, T'_{EL_2}(act_2) \,|\!) = (\!|\, tact_{EL_2}(act_2) \,|\!) = (\!|\, act_2 \,|\!)$

PROOF: By assumption 4.1, assumption 5.1, and definition of T'_{EL_2} (Def. B.3), definition $(\!|\, \_ \,|\!)$ (Def. A.3), and definition of

$tact_{EL}$ (Eq. (15)).

⟨7⟩2. Q.E.D.

PROOF: By ⟨7⟩1 and assumption 2.10.

⟨6⟩2. $as'_2 \in \mathcal{A}$

PROOF: By ⟨6⟩1 and assumption 2.8.

⟨6⟩3. $as_1 = as'_1$

⟨7⟩1. $(\!| T'_{EL_1}(act_1) |\!) = (\!| tact_{EL_1}(act_1) |\!) = (\!| act_1 |\!)$

PROOF: By assumption 3.1, assumption 5.1, and definition of T'_{EL_1} (Def. B.3), definition $(\!| \_ |\!)$ (Def. A.3), and definition of $tact_{EL_1}$ (Eq. (15)).

⟨7⟩2. Q.E.D.

PROOF: By ⟨7⟩1, assumptions 2.3 and 2.5.

⟨6⟩4. $io(as'_2) = io(as'_1)$

PROOF: By ⟨6⟩1, ⟨6⟩2, and assumption 2.8.

⟨6⟩5. Q.E.D.

PROOF: By ⟨6⟩1, ⟨6⟩2, and ⟨6⟩4.

⟨5⟩3. Q.E.D.

PROOF: By ⟨5⟩1 and ⟨5⟩2.

⟨4⟩3. CASE: 4.1 $act_2 = (\epsilon, bx'_\epsilon, oe', a'_\epsilon)$ for $oe' \in \mathbf{OE} \cup \{\epsilon\}$, $bx'_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a'_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

PROOF: Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that $io(as_2) \neq io(as_1)$ by definition of $(\!| \_ |\!)$ (Def. A.3) and definition of io (Eq. (11)). Hence, assumption 4.1 cannot hold.

⟨4⟩4. Q.E.D.

PROOF: By ⟨4⟩1, ⟨4⟩2, ⟨4⟩3 and definition of **Act** (Eq. 7) and (Eq. 8).

⟨3⟩3. CASE: 3.1 $act_1 = (\epsilon, bx_\epsilon, oe, a_\epsilon)$ for $oe \in \mathbf{OE} \cup \{\epsilon\}$, $bx_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

PROOF: Proof is the same as for ⟨3⟩2.

⟨3⟩4. Q.E.D.

PROOF: By ⟨3⟩1, ⟨3⟩2, ⟨3⟩3 and definition of **Act** (Eq. 7) and (Eq. 8).

⟨2⟩2. Q.E.D.

PROOF: By ⟨2⟩1 and logical implication ($\implies$).

⟨1⟩2. CASE: 1.1 $n = n' + 1$ for $n' \in \mathbb{N}$

1.2 $\forall i \in \mathbb{N} : i \geq 2 \wedge i \leq n' \implies Ind(P, Q, EL_1, EL_2, i)$

⟨2⟩1. ASSUME: 2.1 $j + k = n$ for $j, k \in \mathbb{N} \setminus \{0\}$

2.2 $\langle act_{1.1}, \ldots, act_{1.j} \rangle \in pr(|P|)$

2.3 $as'_{1.1} \in (\!| T'_{EL_1}(act_{1.1}) |\!) \wedge \cdots \wedge as'_{1.j} \in (\!| T'_{EL_1}(act_{1.j}) |\!)$

2.4 $as'_1 \in \mathcal{A}$ and $as'_1 = as'_{1.1} \frown \cdots \frown as'_{1.j}$

2.5 $as_{1.1} \in (\!| act_{1.1} |\!) \wedge \cdots \wedge as_{1.j} \in (\!| act_{1.j} |\!)$

2.6 $\forall i \in \{1, \ldots, j\} : (preState(as_{1.i}, var(EL_1)) = preState(as'_{1.i}, var(EL_1))) \wedge$ $(postState(as_{1.i}, var(EL_1)) = postState(as'_{1.i}, var(EL_1)))$

2.7 $as_1 \in pr((\!| P |\!)) \cap \mathcal{A}$ and $as_1 = as_{1.1} \frown \cdots \frown as_{1.j}$

2.8 $as_2 \in pr((\!| Q |\!)) \cap \mathcal{A}$ such that $io(as_2) = io(as_1)$

2.9 $\langle act_{2.1}, \ldots, act_{2.k} \rangle \in |Q|$

2.10 $as_{2.1} \in (\!| act_{2.1} |\!) \wedge \cdots \wedge as_{2.k} \in (\!| act_{2.k} |\!)$

2.11 $as_2 = as_{2.1} \frown \cdots \frown as_{2.k}$

PROVE: $\exists as'_2 \in (\!| T'_{EL_2}(act_{2.1}) |\!) \frown \cdots \frown (\!| T'_{EL_2}(act_{2.k}) |\!) : as'_2 \in \mathcal{A} \wedge io(as'_2) = io(as'_1)$

⟨3⟩1. CASE: 3.1 $act_{1.j} = (\epsilon, bx_\epsilon, \epsilon, a_\epsilon)$ for $bx_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$ and $a_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

$\langle 4\rangle 1.$ Choose $aas'_1 \in \mathcal{A}$ such that $as'_1 = aas'_1 \frown as'_{1.j}$
PROOF: By assumption 2.4 and definition of $\mathcal{A}$ (Eq. (10)).

$\langle 4\rangle 2.$ Choose $aas_1 \in pr(\llparenthesis P \rrparenthesis) \cap \mathcal{A}$ such that $as_1 = aas_1 \frown as_{1.j}$
PROOF: By assumption 2.7, definition of $\mathcal{A}$ (Eq. (10)), and definition of $pr(\_)$ (Def. D.3).

$\langle 4\rangle 3.$ $io(aas_1) = io(as_2)$
PROOF: By $\langle 4\rangle 2$, assumptions 2.5, 2.8 and 3.1, and definition of $\llparenthesis \_ \rrparenthesis$ (Def. A.3) and definition of io (Eq. (11)).

$\langle 4\rangle 4.$ Q.E.D.
PROOF: By assumptions 1.1, 1.2, 2.1, 2.2, 2.3, 2.5, 2.6, and 2.8 - 2.11, and $\langle 4\rangle 1$, $\langle 4\rangle 2$, and $\langle 4\rangle 3$ since this implies that the antecedent of $Ind(P, Q, EL_1, EL_2, n')$ is satisfied for $j-1$ and k.

$\langle 3\rangle 2.$ CASE: 3.1 $act_{1.j} = (ie, bx_\epsilon, \epsilon, a_\epsilon)$ for $ie \in \mathbf{IE}$, $bx_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

$\langle 4\rangle 1.$ CASE: 4.1 $act_{2.k} = (\epsilon, bx'_\epsilon, \epsilon, a'_\epsilon)$ for $bx'_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$ and $a'_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

$\langle 5\rangle 1.$ Choose $aas_2 \in pr(\llparenthesis Q \rrparenthesis) \cap \mathcal{A}$ such that $as_2 = aas_2 \frown as_{2.k}$
PROOF: By assumptions 2.8 and 2.11, and definition of $\mathcal{A}$ (Eq. (10)), and definition of $pr(\_)$ (Def. D.3).

$\langle 5\rangle 2.$ $io(as_1) = io(aas_2)$
PROOF: By $\langle 5\rangle 1$, assumptions 4.1 and 2.8, and definition of $\llparenthesis \_ \rrparenthesis$ (Def. A.3) and definition of io (Eq. (11)).

$\langle 5\rangle 3.$ Choose $aas'_2 \in \llparenthesis T'_{EL_2}(act_{2.1}) \rrparenthesis \frown \cdots \frown \llparenthesis T'_{EL_2}(act_{2.k-1}) \rrparenthesis : aas'_2 \in \mathcal{A} \wedge io(aas'_2) = io(as'_1)$
PROOF: By assumptions 1.1, 1.2, 2.1 - 2.7, 2.9 - 2.11, and $\langle 5\rangle 1$ and $\langle 5\rangle 2$ since this implies that the antecedent of $Ind(P, Q, EL_1, EL_2, n')$ is satisfied for j and $k-1$.

$\langle 5\rangle 4.$ Choose $as'_{2.k} \in \llparenthesis T'_{EL_2}(act_{2.k}) \rrparenthesis \cap \mathcal{A}$ such that $io(as'_{2.k}) = \langle\rangle$ and $preState(as'_{2.k}) = postState(aas'_2)$
PROOF: By $\langle 5\rangle 3$, assumption 4.1, assumption 1, assumption 2.8 - 2.11, definition of T'_{EL_2} (Def. B.3), and definition of $\llparenthesis \_ \rrparenthesis$ (Def. A.3).

$\langle 5\rangle 5.$ $aas'_2 \frown as'_{2.k} \in \mathcal{A}$ and $io(aas'_2 \frown as'_{2.k}) = io(as'_1)$
PROOF: By $\langle 5\rangle 3$, $\langle 5\rangle 4$, definition of $\mathcal{A}$ (Eq. (10)), and definition of io (Eq. (11)).

$\langle 5\rangle 6.$ Q.E.D.
PROOF: By $\langle 5\rangle 3$, $\langle 5\rangle 4$, and $\langle 5\rangle 5$.

$\langle 4\rangle 2.$ CASE: 4.1 $act_{2.k} = (ie', bx'_\epsilon, \epsilon, a'_\epsilon)$ for $ie' \in \mathbf{IE}$, $bx'_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a'_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

$\langle 5\rangle 1.$ Choose $aas'_1 \in \mathcal{A}$ such that $as'_1 = aas'_1 \frown as'_{1.j}$
PROOF: By assumption 2.4 and definition of $\mathcal{A}$ (Eq. (10)).

$\langle 5\rangle 2.$ Choose $aas_1 \in pr(\llparenthesis P \rrparenthesis) \cap \mathcal{A}$ such that $as_1 = aas_1 \frown as_{1.j}$
PROOF: By assumption 2.7, definition of $\mathcal{A}$ (Eq. (10)), and definition of $pr()$ (Def. D.3).

$\langle 5\rangle 3.$ Choose $aas_2 \in pr(\llparenthesis Q \rrparenthesis) \cap \mathcal{A}$ such that $as_2 = aas_2 \frown as_{2.k}$
PROOF: By assumptions 2.8 and 2.11, and definition of $\mathcal{A}$ (Eq. (10)), and definition of $pr()$ (Def. D.3).

$\langle 5\rangle 4.$ $io(aas_1) = io(aas_2)$
PROOF: By $\langle 5\rangle 2$, $\langle 5\rangle 3$, assumptions 2.8, 3.1, and 4.1, and definition of io (Eq. (11)).

$\langle 5\rangle 5.$ Choose $aas'_2 \in \llparenthesis T'_{EL_2}(act_{2.1}) \rrparenthesis \frown \cdots \frown \llparenthesis T'_{EL_2}(act_{2.k-1}) \rrparenthesis \cap \mathcal{A}$ such that $io(aas'_1) = io(aas'_2)$

PROOF: By assumptions 1.1, 1.2, 2.1, 2.2, 2.3, 2.5, 2.6, 2.9, 2.10, 2.11, and $\langle 5\rangle 1$ - $\langle 5\rangle 4$ since this implies that the antecedent of $Ind(P, Q, EL_1, EL_2, n'-1)$ is satisfied for $j-1$ and $k-1$.

$\langle 5\rangle 6$. Choose $as'_{2.k} \in (\!| T'_{EL_2}(act_{2.k}) |\!) \cap \mathcal{A}$ such that $io(as'_{2.k}) = io(as'_{1.j})$ and $postState(aas'_2) = preState(as'_{2.k})$

$\langle 6\rangle 1$. CASE: 5.1 $ie \in \mathcal{D}om(te_{EL_1})$ and $ie' \in \mathcal{D}om(te_{EL_2})$

$\langle 7\rangle 1$. $\exists as'_{2.k} \in (\!| (\epsilon, bx'_\epsilon, \epsilon, \epsilon).te_{EL_2}(ie').(\epsilon, \epsilon, \epsilon, a'_\epsilon) |\!) \cap \mathcal{A} \wedge io(as'_{2.k}) = io(as'_{1.j}) \wedge postState(aas'_2) = preState(as'_{2.k})$

$\langle 8\rangle 1$. $\exists as'_{2.k} \in (\!| (\epsilon, bx'_\epsilon, \epsilon, \epsilon) |\!) \frown (\!| te_{EL_2}(ie') |\!) \frown (\!| (\epsilon, \epsilon, \epsilon, a'_\epsilon) |\!) \cap \mathcal{A} \wedge io(as'_{2.k}) = io(as'_{1.j}) \wedge postState(aas'_2) = preState(as'_{2.k})$

$\langle 9\rangle 1$. Choose $aas'_{1.1} \in (\!| (\epsilon, bx_\epsilon, \epsilon, \epsilon) |\!)$, $aas'_{1.2} \in (\!| te_{EL_1}(ie) |\!)$, and $aas'_{1.3} \in (\!| (\epsilon, \epsilon, \epsilon, a_\epsilon) |\!)$ such that $as'_{1.j} = aas'_{1.1} \frown aas'_{1.2} \frown aas'_{1.3}$

$\langle 10\rangle 1$. $(\!| T'_{EL_1}(act_{1.j}) |\!) = (\!| tact_{EL_1}(act_{1.j}) |\!) = (\!| (\epsilon, bx_\epsilon, \epsilon, \epsilon).te_{EL_1}(ie).(\epsilon, \epsilon, \epsilon, a_\epsilon) |\!)$

PROOF: By definition of T'_{EL_1} (Def. B.3), assumption 3.1, assumption 5.1, and definition of $tact_{EL}$ (Eq. (15)).

$\langle 10\rangle 2$. $(\!| (\epsilon, bx_\epsilon, \epsilon, \epsilon).te_{EL_1}(ie).(\epsilon, \epsilon, \epsilon, a_\epsilon) |\!) = (\!| (\epsilon, bx_\epsilon, \epsilon, \epsilon) |\!) \frown (\!| te_{EL_1}(ie) |\!) \frown (\!| (\epsilon, \epsilon, \epsilon, a_\epsilon) |\!)$

PROOF: By definition of $(\!| \_ |\!)$ (Def. A.3).

$\langle 10\rangle 3$. Q.E.D.

PROOF: By $\langle 10\rangle 1$, $\langle 10\rangle 2$, and assumption 2.3.

$\langle 9\rangle 2$. Choose variable renaming functions $vr_1, vr_2 \in \mathbf{Var} \rightarrow \mathbf{Var}$ such that $vr_1(EL) = EL_2$ and $vr_2(EL) = EL_2$ when vr_1 and vr_2 are lifted to event libraries.

PROOF: By assumption 1.

$\langle 9\rangle 3$. Choose $aas'_{2.1} \in (\!| (\epsilon, bx'_\epsilon, \epsilon, \epsilon) |\!)$ such that $preState(aas'_{2.1}) = postState(aas'_2)$ and $preState(aas'_{2.1})(vr_2(x)) = preState(aas'_{1.1})(vr_1(x))$ for all $x \in var(EL)$

PROOF: By assumptions 1, 2.9, and 4.1, and $\langle 5\rangle 5$, $\langle 9\rangle 1$, definition of $(\!| \_ |\!)$ (Def. A.3), and definition of $preState()$ (Def. D.2).

$\langle 9\rangle 4$. $aas'_{2.1} \in \mathcal{A}$

PROOF: By $\langle 9\rangle 3$, assumption 2.8, assumption 2.10, assumption 4.1, and definition of $\mathcal{A}$ (Eq. (10)).

$\langle 9\rangle 5$. $io(aas'_{1.1}) = io(aas'_{2.1})$

PROOF: By $\langle 9\rangle 1$, $\langle 9\rangle 3$ and definition of $(\!| \_ |\!)$ (Def. A.3) and io (Eq. (11)).

$\langle 9\rangle 6$. Choose $aas'_{2.2} \in (\!| te_{EL_2}(ie') |\!)$ such that $preState(aas'_{2.2}) = postState(aas'_{2.1})$

PROOF: By $\langle 9\rangle 3$, assumption 5.1, definition of $(\!| \_ |\!)$ (Def. A.3), and definition of $preState()$ and $postState()$ (Def. D.2).

$\langle 9\rangle 7$. $aas'_{2.2} \in \mathcal{A}$

PROOF: By assumption 2.4, assumption 2.6, $\langle 9\rangle 1$, $\langle 9\rangle 3$, $\langle 9\rangle 6$, and definition of $\mathcal{A}$ (Eq. (10)).

$\langle 9\rangle 8$. $io(aas'_{2.2}) = io(aas'_{1.2})$

PROOF: By assumption 2.6, assumption 2.8, $\langle 9\rangle 1$, $\langle 9\rangle 3$, $\langle 9\rangle 6$, and definition of io (Eq. (11)).

$\langle 9\rangle 9$. Choose $aas'_{2.3} \in (\!| (\epsilon, \epsilon, \epsilon, a'_\epsilon) |\!)$ such that $preState(aas'_{2.3}) = postState(aas'_{2.2})$

PROOF: By definition of $(\!|\_|\!)$ (Def. A.3), and definition of $preState()$ and $postState()$ (Def. D.2).

$\langle 9 \rangle 10.$ $aas'_{2.3} \in \mathcal{A}$

PROOF: By $\langle 9 \rangle 9$ and definition of $\mathcal{A}$ (Eq. (10)).

$\langle 9 \rangle 11.$ $io(aas'_{2.3}) = io(aas'_{1.3})$

PROOF: By $\langle 9 \rangle 1$, $\langle 9 \rangle 9$, definition of $(\!|\_|\!)$ (Def. A.3), and definition of io (Eq. (11)).

$\langle 9 \rangle 12.$ Choose $aas'_2 \in \mathcal{A}$ such that $aas'_2 = aas'_{2.1} \frown aas'_{2.2} \frown aas'_{2.3}$

PROOF: By $\langle 9 \rangle 4$, $\langle 9 \rangle 6$, $\langle 9 \rangle 7$, $\langle 9 \rangle 9$, $\langle 9 \rangle 10$, and definition of $\mathcal{A}$ (Eq. (10)).

$\langle 9 \rangle 13.$ $io(as'_{1.j}) = io(aas'_2)$

PROOF: By $\langle 9 \rangle 1$, $\langle 9 \rangle 5$, $\langle 9 \rangle 8$, $\langle 9 \rangle 11$, $\langle 9 \rangle 13$, and definition of io (Eq. (11)).

$\langle 9 \rangle 14.$ Q.E.D.

PROOF: By $\langle 9 \rangle 12$, $\langle 9 \rangle 13$.

$\langle 8 \rangle 2.$ Q.E.D.

PROOF: By $\langle 8 \rangle 1$ and definition $(\!|\_|\!)$ (Def. A.3).

$\langle 7 \rangle 2.$ Q.E.D.

PROOF: By assumption 5.1, $\langle 7 \rangle 1$, and definition of T'_{EL_2} (Def. B.3).

$\langle 6 \rangle 2.$ CASE: $ie \notin \mathcal{D}om(te_{EL_1})$ and $ie' \notin \mathcal{D}om(te_{EL_2})$

$\langle 7 \rangle 1.$ Choose $as'_{2.k} \in (\!| T'_{EL_2}(act_{2.k}) |\!)$ such that $as'_{2.k} = as_{2.k}$

$\langle 8 \rangle 1.$ $(\!| T'_{EL_2}(act_{2.k}) |\!) = (\!| tact_{EL_2}(act_{2.k}) |\!) = (\!| act_{2.k} |\!)$

PROOF: By assumption 4.1, assumption 5.1, and definition of T'_{EL_2} (Def. B.3), and definition of $tact_{EL_2}$ (Eq. (15)).

$\langle 8 \rangle 2.$ Q.E.D.

PROOF: By $\langle 8 \rangle 1$ and assumption 2.10.

$\langle 7 \rangle 2.$ $as'_{2.k} \in \mathcal{A}$

PROOF: By $\langle 7 \rangle 1$ and assumption 2.8.

$\langle 7 \rangle 3.$ $as_{1.j} = as'_{1.j}$

$\langle 8 \rangle 1.$ $(\!| T'_{EL_1}(act_{1.j}) |\!) = (\!| tact_{EL_1}(act_{1.j}) |\!) = (\!| act_{1.j} |\!)$

PROOF: By assumption 3.1, assumption 5.1, and definition of T'_{EL_1} (Def. B.3), and definition of $tact_{EL_1}$ (Eq. (15)).

$\langle 8 \rangle 2.$ Q.E.D.

PROOF: By $\langle 8 \rangle 1$, assumptions 2.3, 2.5, and 2.6.

$\langle 7 \rangle 4.$ $io(as'_{2.k}) = io(as'_{1.j})$

PROOF: By $\langle 7 \rangle 1$, $\langle 7 \rangle 3$, and assumption 2.8.

$\langle 7 \rangle 5.$ Q.E.D.

PROOF: By $\langle 7 \rangle 1$, $\langle 7 \rangle 2$, and $\langle 7 \rangle 4$.

$\langle 6 \rangle 3.$ CASE: $ie \notin \mathcal{D}om(te_{EL_1})$ and $ie' \in \mathcal{D}om(te_{EL_2})$

PROOF: Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that $io(as_2) \neq io(as_1)$ by assumption 1, definition of rnm (App.B.1), definition of io (Eq. (11)) and definition of te_{EL} (Eq. 14). Hence, the case assumption cannot hold.

$\langle 6 \rangle 4.$ CASE: $ie \in \mathcal{D}om(te_{EL_1})$ and $ie' \notin \mathcal{D}om(te_{EL_2})$

PROOF: Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that $io(as_2) \neq io(as_1)$ by assumption 1, definition of rnm (App.B.1), definition of io (Eq. (11)) and definition of te_{EL} (Eq. 14). Hence, the case assumption cannot hold.

$\langle 6 \rangle 5.$ Q.E.D.

PROOF: By $\langle 6\rangle 1$ and $\langle 6\rangle 2$, $\langle 6\rangle 3$, and $\langle 6\rangle 4$.

$\langle 5\rangle 7$. Q.E.D.

PROOF: By $\langle 5\rangle 5$ and $\langle 5\rangle 6$.

$\langle 4\rangle 3$. CASE: 4.1 $act_{2.k} = (\epsilon, bx'_\epsilon, oe', a'_\epsilon)$ for $oe' \in \mathbf{OE}$, $bx'_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a'_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

PROOF: Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that $io(as_2) \neq io(as_1)$ by definition of $(\!|\_|\!)$ (Def. A.3) and definition of io (Eq. (11)). Hence, assumption 4.1 cannot hold.

$\langle 4\rangle 4$. Q.E.D.

PROOF: By $\langle 4\rangle 1$, $\langle 4\rangle 2$, $\langle 4\rangle 3$ and definition of **Act** (Eq. (7)) and (Eq. (8)).

$\langle 3\rangle 3$. CASE: 3.1 $act_{1.j} = (\epsilon, bx_\epsilon, oe, a_\epsilon)$ for $oe \in \mathbf{OE}$, $bx_\epsilon \in \mathbf{BExp} \cup \{\epsilon\}$, and $a_\epsilon \in \mathbf{Assign} \cup \{\epsilon\}$

PROOF: Proof is the same as for $\langle 3\rangle 2$.

$\langle 3\rangle 4$. Q.E.D.

PROOF: By $\langle 3\rangle 1$, $\langle 3\rangle 2$, $\langle 3\rangle 3$ and definition of **Act** (Eq. (7)) and (Eq. (8)).

$\langle 2\rangle 2$. Q.E.D.

PROOF: By $\langle 2\rangle 1$ and logical implication ($\Longrightarrow$).

$\langle 1\rangle 3$. Q.E.D.

PROOF: By $\langle 1\rangle 2$ and $\langle 1\rangle 2$.

Lemma B.1 If EL is an event library, and P_1 and P_2 be basic state machines whose variables are disjoint from those in EL, then $[\![T'_{EL}]\!]$ is homomorphic w.r.t. union, i.e.,

$$[\![T'_{EL}]\!]([\![P_1]\!] \cup [\![P_2]\!]) = [\![T'_{EL}]\!]([\![P_1]\!]) \cup [\![T'_{EL}]\!]([\![P_2]\!])$$

Proof of Lemma B.1.1

$$\begin{array}{rcll}
[\![T'_{EL}(P_1) + T'_{EL}(P_2)]\!] & = & [\![T'_{EL}(P_1) + T'_{EL}(P_2)]\!] & \\
[\![T'_{EL}(P_1 + P_2)]\!] & = & [\![T'_{EL}(P_1) + T'_{EL}(P_2)]\!] & \text{Def. B.3} \\
[\![T'_{EL}(P_1 + P_2)]\!] & = & [\![T'_{EL}(P_1)]\!] \cup [\![T'_{EL}(P_2)]\!] & \text{Lemma B.1.2} \\
[\![T'_{EL}]\!]([\![P_1 + P_2]\!]) & = & [\![T'_{EL}]\!]([\![P_1]\!]) \cup [\![T'_{EL}]\!]([\![P_2]\!]) & \text{Def. B.4} \\
[\![T'_{EL}]\!]([\![P_1]\!] \cup [\![P_2]\!]) & = & [\![T'_{EL}]\!]([\![P_1]\!]) \cup [\![T'_{EL}]\!]([\![P_2]\!]) & \text{Lemma B.1.2}
\end{array}$$

Lemma B.1.2 $[\![P + Q]\!] = [\![P]\!] \cup [\![Q]\!]$

Proof of Lemma B.1.2

$$\begin{array}{rcll}
[\![P + Q]\!] & = & [\![P + Q]\!] & \\
[\![P + Q]\!] & = & io((\!| P + Q |\!) \cap \mathcal{A}) & \text{Def. A.4} \\
[\![P + Q]\!] & = & io(((\!| P |\!) \cup (\!| Q |\!)) \cap \mathcal{A}) & \text{Def. A.3} \\
[\![P + Q]\!] & = & io(((\!| P |\!) \cap \mathcal{A}) \cup ((\!| Q |\!) \cap \mathcal{A})) & \\
[\![P + Q]\!] & = & io(((\!| P |\!) \cap \mathcal{A})) \cup io(((\!| Q |\!) \cap \mathcal{A})) & \text{By (11)} \\
[\![P + Q]\!] & = & [\![P]\!] \cup [\![Q]\!] & \text{Def. A.4}
\end{array}$$

Lemma B.2 If EL is an event library, and P and Q be basic state machines whose variables are disjoint from those in EL, then $[\![T'_{EL}]\!]()$ is homomorphic w.r.t. concatenation, i.e.,

$$[\![T'_{EL}]\!]([\![P]\!]) \frown [\![T'_{EL}]\!]([\![Q]\!]) \quad = \quad [\![T'_{EL}]\!]([\![P]\!] \frown [\![Q]\!])$$

Proof of Lemma B.2
ASSUME: 1. $var(EL) \cap (var(Q) \cup var(P)) = \emptyset$
PROVE: $[\![T'_{EL}]\!]([\![P]\!]) \frown [\![T'_{EL}]\!]([\![Q]\!]) = [\![T'_{EL}]\!]([\![P]\!] \frown [\![Q]\!])$

$\langle 1 \rangle 1.$ Choose $Q' \in \mathbf{Q}$ such that $[\![Q]\!] = [\![Q']\!]$, and $[\![P.Q']\!] = [\![P]\!] \frown [\![Q']\!]$
PROOF: Assume that all variables in Q are assigned to a value before they are used in a guard. Then execution of Q will be the same even if we choose an arbitrary initial data state. Therefore we must have that $[\![P.Q]\!] = [\![P]\!] \frown [\![Q]\!]$. However, if Q has unassigned variables in guards, then executing P before executing Q might affect the execution of Q. Therefore it may be the case that $[\![P.Q]\!] \neq [\![P]\!] \frown [\![Q]\!]$. However, we can always rename the variables of Q to obtain the state machine Q' such that $[\![Q]\!] = [\![Q']\!]$ and $var(P) \cap var(Q') \neq \emptyset$. In this case, executing P before Q' will not affect the execution of Q'. Therefore $[\![P.Q']\!] = [\![P]\!] \frown [\![Q']\!]$.
$\langle 1 \rangle 2.$ $[\![T'_{EL}]\!]([\![P]\!]) \frown [\![T'_{EL}]\!]([\![Q']\!]) = [\![T'_{EL}]\!]([\![P]\!] \frown [\![Q']\!])$
PROOF:

$$\begin{array}{rcll}
[\![T'_{EL}(P).T'_{EL}(Q')]\!] & = & [\![T'_{EL}(P).T'_{EL}(Q')]\!] & \\
[\![T'_{EL}(P.Q')]\!] & = & [\![T'_{EL}(P).T'_{EL}(Q')]\!] & \text{Def. B.3} \\
[\![T'_{EL}(P.Q')]\!] & = & [\![T'_{EL}(P)]\!] \frown [\![T'_{EL}(Q')]\!] & \langle 1 \rangle 1 \\
[\![T'_{EL}]\!]([\![P.Q']\!]) & = & [\![T'_{EL}]\!]([\![P]\!]) \frown [\![T'_{EL}]\!]([\![Q']\!]) & \text{Def. B.4} \\
[\![T'_{EL}]\!]([\![P]\!] \frown [\![Q']\!]) & = & [\![T'_{EL}]\!]([\![P]\!]) \frown [\![T'_{EL}]\!]([\![Q']\!]) & \langle 1 \rangle 1 \text{ and ass. } 1
\end{array}$$

$\langle 1 \rangle 3.$ Q.E.D.
PROOF: By $\langle 1 \rangle 2$ and Theorem B.1

D.3 Composite transformations

Theorem B.2 The relation $[\![T^C_{EL}]\!]$ is a function when restricted to the image of $\widehat{T}_{EL}$, i.e.,

$$[\![P]\!] = [\![Q]\!] \implies [\![T^C_{EL}(P)]\!] \cap Im_{EL} = [\![T^C_{EL}(Q)]\!] \cap Im_{EL}$$

Proof of Theorem B.2
ASSUME: 1. $[\![P_1 \parallel \cdots \parallel P_n]\!] = [\![Q_1 \parallel \cdots \parallel Q_n]\!]$
PROVE: $[\![T^C_{EL}(P_1 \parallel \cdots \parallel P_n)]\!] \cap Im_{EL} = [\![T^C_{EL}(Q_1 \parallel \cdots \parallel Q_n)]\!] \cap Im_{EL}$

$\langle 1 \rangle 1.$ ASSUME: 1.1 $s' \in [\![T^C_{EL}(P_1 \parallel \cdots \parallel P_n)]\!] \cap Im_{EL}$
PROVE: $s' \in [\![T^C_{EL}(Q_1 \parallel \cdots \parallel Q_n)]\!] \cap Im_{EL}$
$\langle 2 \rangle 1.$ Choose $EL_1 \in \mathbf{EL}$ such that $EL_1 = rnm(P_1 \parallel \cdots \parallel P_n, EL)$
PROOF: By definition of rnm (see App. B.1).
$\langle 2 \rangle 2.$ Choose $s'_1 \in [\![T'_{EL_1}]\!]([\![P_1]\!]), \ldots, s'_n \in [\![T'_{EL_1}]\!]([\![P_n]\!])$ such that $s' \in \parallel (s'_1, \ldots, s'_n) \cap Im_{EL_1}$
PROOF: By assumption 1.1 and definition of $[\![\_]\!]$ (Def. A.5), and definition of $T^C_{EL'}$ (Def. B.5).
$\langle 2 \rangle 3.$ Choose $s_1 \in [\![P_1]\!], \ldots, s_n \in [\![P_n]\!]$ such that $s'_1 \in [\![T'_{EL_1}]\!](\{s_1\}), \ldots, s'_n \in [\![T'_{EL_1}]\!](\{s_n\})$
PROOF: By $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, Theorem B.1 (which ensures that $[\![T'_{EL_1}]\!](\_)$ is a function) and Lemma B.1(which ensures that $[\![T'_{EL_1}]\!](\_)$ is defined for singleton sets).
$\langle 2 \rangle 4.$ Choose $s \in [\![P_1 \parallel \cdots \parallel P_n]\!]$ such that $s \in \parallel (s_1, \ldots, s_n)$
PROOF: By $\langle 2 \rangle 2$, $\langle 2 \rangle 3$, definition of $[\![\_]\!]$ (Def. A.5) and definition of Im_{EL_1} (Eq. (16)).

⟨2⟩5. $s \in [\![Q_1 \parallel \cdots \parallel Q_n]\!]$
PROOF: By ⟨2⟩4 and assumption 1.

⟨2⟩6. $t_1 \in [\![Q_1]\!], \ldots, t_n \in [\![Q_n]\!]$ such that $s \in \parallel (t_1, \ldots, t_n)$
PROOF: By ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, definition of $[\![\ ]\!]$ (Def. A.5), and definition of composite state machine expressions Def. A.2.

⟨2⟩7. $t_i = s_i$ for all $i \in \{1, \ldots, n\}$
PROOF: By definition of a composite state machine, each basic state machine it consists of must have different names. Therefore $[\![Q_i]\!] \cap [\![Q_j]\!] \neq \emptyset$ for all $i, j \in \{1, \ldots, n\}$ such that $i \neq j$. By definition of parallel composition, there is one and only one trace t_i in each basic state machine Q_i (for $i \in \{1, \ldots, n\}$) such that $s \in (t_1, \ldots, t_n)$. The same holds for the basic state machines P_i, therefore we have that s_i must be equal to t_i.

⟨2⟩8. Choose $EL_2 \in \mathbf{EL}$ such that $EL_2 = rnm(Q_1 \parallel \cdots \parallel Q_n, EL)$
PROOF: By definition of rnm (see App. B.1).

⟨2⟩9. $s' \in \parallel ([\![T'_{EL}]\!](\{s_1\}), \ldots, [\![T'_{EL}]\!](\{s_n\})) \cap Im_{EL}$
PROOF: By ⟨2⟩2, ⟨2⟩3, ⟨2⟩8, Theorem B.1 (which ensures that $[\![T_{EL}]\!](\_)$ is a function).

⟨2⟩10. Q.E.D.
PROOF: By ⟨2⟩2, ⟨2⟩3, ⟨2⟩6, ⟨2⟩7, ⟨2⟩9, and definition of $[\![\ ]\!]$ (Def. A.5).

⟨1⟩2. Q.E.D.
PROOF: By ⟨1⟩1 and symmetry of $=$.

Lemma B.3 The semantics of the event transformation for composite state machines induced by an event library EL is homomorphic w.r.t. the union operator when restricted to its image, i.e.,

$$\widehat{T}_{EL}([\![P]\!] \cup [\![Q]\!]) = \widehat{T}_{EL}([\![P]\!]) \cup \widehat{T}_{EL}([\![Q]\!])$$

when $var(EL) \cap var(P) \cap var(Q) = \emptyset$

Proof of Lemma B.3 Assume $var(EL) \cap var(P) \cap var(Q) = \emptyset$, then we have

$$\begin{array}{rcll} \bigcup_{s \in [\![P]\!] \cup [\![Q]\!]} t^C_{EL}(s) & = & (\bigcup_{s \in [\![P]\!]} t^C_{EL}(s)) \cup (\bigcup_{s \in [\![Q]\!]} t^C_{EL}(s)) & \\ \widehat{T}_{EL}([\![P]\!] \cup [\![Q]\!]) & = & \widehat{T}_{EL}([\![P]\!]) \cup \widehat{T}_{EL}([\![Q]\!]) & \text{By Lemma B.3.1} \end{array}$$

Lemma B.3.1 Let the function $t^C_{EL} \in \mathbf{E}^* \rightarrow \mathbb{P}(\mathbf{E}^*)$ be defined by

$$t^C_{EL}(s) \stackrel{\mathsf{def}}{=} \parallel ([\![T'_{EL}]\!](\{s|_{\mathbf{E}_{nm_1}}\}), \cdots, [\![T'_{EL}]\!](\{s|_{\mathbf{E}_{nm_n}}\})$$

for all $s \in \mathcal{T}$ and $\mathbf{Nm} = \{nm_1, \ldots, nm_n\}$.

Then the semantics of the transformation $\widehat{T}_{EL}(P)$ is entirely characterized by t^C_{EL} if $var(EL) \cap var(P) = \emptyset$, i.e.,

$$\widehat{T}_{EL}([\![P]\!]) = \bigcup_{s \in [\![P]\!]} t^C_{EL}(s)$$

Proof of Lemma B.3.1
PROVE: $\widehat{T}_{EL}([\![P]\!]) = \bigcup_{s \in [\![P]\!]} t^C_{EL}(s)$

LET: $P \stackrel{\mathsf{def}}{=} P_1 \parallel \cdots \parallel P_n$

⟨1⟩1. ASSUME: 1.1 $var(EL) \cap var(P) = \emptyset$
1.2 $s' \in \widehat{T}_{EL}(\llbracket P \rrbracket)$
PROVE: $s' \in t^C_{EL}(s)$ for some $s \in \llbracket P \rrbracket$

⟨2⟩1. Choose $s_1 \in \llbracket P_1 \rrbracket, \ldots, s_n \in \llbracket P_n \rrbracket$ such that $s' \in \parallel (\llbracket T'_{EL} \rrbracket(\{s_1\}), \ldots, \llbracket T'_{EL} \rrbracket(\{s_n\})) \cap Im_{EL}$
PROOF: By assumption 1.1, assumption 1.2, definition of $\llbracket\ \rrbracket$ (Def. A.5), and definition of $\widehat{T}_{EL}()$ (Eq. (17)).

⟨2⟩2. Choose $s \in \llbracket P \rrbracket$ such that $s \in \parallel (s_1, \ldots, s_n)$
PROOF: By ⟨2⟩1, definition of P, definition of $\llbracket \_ \rrbracket$ (Def. A.5) and definition of Im_{EL} (Eq. (16)).

⟨2⟩3. $s_1 = s|_{\mathbf{E}_{nm_1}}, \ldots, s_n = s|_{\mathbf{E}_{nm_n}}$ for some $nm_1, \ldots, nm_n \in \mathbf{Nm}$
PROOF: By ⟨2⟩1, definition of P and definition of $\mathbf{P}^C$ (Def. A.2).

⟨2⟩4. Q.E.D.
PROOF: By ⟨2⟩1, ⟨2⟩2, ⟨2⟩3 and definition of t^C_{EL}.

⟨1⟩2. ASSUME: 1.1 $var(EL) \cap var(P) = \emptyset$
1.2 $s' \in t^C_{EL'}(s)$ for arbitrary $s \in \llbracket P \rrbracket$
PROVE: $s' \in \widehat{T}_{EL}(\llbracket P \rrbracket)$

⟨2⟩1. Choose $s_1 \in \llbracket P_1 \rrbracket, \ldots, s_n \in \llbracket P_n \rrbracket$ and $nm_1, \ldots, nm_n \in \mathbf{Nm}$ such that $s_1 = s|_{\mathbf{E}_{nm_1}}, \ldots, s_n = s|_{\mathbf{E}_{nm_n}}$
PROOF: By assumption 1.1, assumption 1.2, definition of $\llbracket \_ \rrbracket$ (Def. A.5), and definition of $\mathbf{E}_{nm}$ (Eq. (9)).

⟨2⟩2. $s' \in \parallel (\llbracket T'_{EL} \rrbracket(\{s_1\}), \ldots, \llbracket T'_{EL} \rrbracket(\{s_n\}))$
PROOF: By assumption 1.1, assumption 1.2, ⟨2⟩1, and definition of t^C_{EL}.

⟨2⟩3. Q.E.D.
PROOF: By ⟨2⟩1, ⟨2⟩2, definition of $\llbracket\ \rrbracket$ (Def. A.5), and definition of $\llbracket T^C_{EL} \rrbracket()$ (Def. B.5).

⟨1⟩3. Q.E.D.
PROOF: By ⟨1⟩1 and ⟨1⟩2.

D.4 Adherence preservation under event transformations

Theorem B.3 Let $\mathfrak{r}$ be a restriction, and $\mathfrak{h}$ be a high-level relation, and $\llbracket T^C_{EL} \rrbracket$ be the transformation induced by event library EL. Then T^C_{EL} preserves $\text{BSP}_{\mathfrak{rh}}$ for specification P if the following conditions are satisfied for all $s, t, u \in \llbracket P \rrbracket, s' \in \widehat{T}_{EL}(\{s\}), t' \in \widehat{T}_{EL}(\{t\})$

$$s' \xrightarrow{\mathfrak{r}} t' \implies s \xrightarrow{\mathfrak{r}} t \tag{26}$$

$$s \xrightarrow{\mathfrak{r}} t \wedge s \xrightarrow{\mathfrak{h}} u \sim_{\mathfrak{l}} t \implies \exists u' \in \widehat{T}_{EL}(\{u\}) : s' \xrightarrow{\mathfrak{h}} u' \sim_{\mathfrak{l}} t' \tag{27}$$

Proof of Theorem B.3
ASSUME: 1. $\forall s, t \in \llbracket P \rrbracket, s' \in \widehat{T}_{EL}(\{s\}), t' \in \widehat{T}_{EL}(\{t\}) : s' \xrightarrow{\mathfrak{r}} t' \implies s \xrightarrow{\mathfrak{r}} t$
2. $\forall s, t, u \in \llbracket P \rrbracket, s' \in \widehat{T}_{EL}(\{s\}), t' \in \widehat{T}_{EL}(\{t\}) :$
$s \xrightarrow{\mathfrak{r}} t \wedge s \xrightarrow{\mathfrak{h}} u \sim_{\mathfrak{l}} t \implies \exists u' \in \widehat{T}_{EL}(\{u\}) : s' \xrightarrow{\mathfrak{h}} u' \sim_{\mathfrak{l}} t'$
PROVE: $\text{BSP}_{\mathfrak{rh}}(\llbracket P \rrbracket) \implies \text{BSP}_{\mathfrak{rh}}(\widehat{T}_{EL}(\llbracket P \rrbracket))$

⟨1⟩1. ASSUME: 1.1 $\text{BSP}_{\mathfrak{rh}}(\llbracket P \rrbracket)$
1.2 $s' \xrightarrow{\mathfrak{r}} t'$ for $s', t' \in \widehat{T}_{EL}(\llbracket P \rrbracket)$
PROVE: $\exists u' \in \widehat{T}_{EL}(\llbracket P \rrbracket) : s' \xrightarrow{\mathfrak{h}} u' \sim_{\mathfrak{l}} t'$

⟨2⟩1. Choose $s, t \in [\![P]\!]$ such that $s \xrightarrow{\mathfrak{r}} t$, $s' \in \widehat{T}_{EL}(\{s\})$, and $t' \in \widehat{T}_{EL}(\{t\})$
PROOF: By assumption 1, assumption 1.2, and Lemma. B.3.

⟨2⟩2. Choose $u \in [\![P]\!]$ such that $s \xrightarrow{\mathfrak{h}} u \sim_{\mathfrak{l}} t$
PROOF: By assumption 1.1, ⟨2⟩1 and definition of $\mathrm{BSP}_{\mathfrak{r}\mathfrak{h}}$ (Eq. (18)).

⟨2⟩3. Choose $u' \in \widehat{T}_{EL}(\{u\})$ such that $s' \xrightarrow{\mathfrak{h}} u' \sim_{\mathfrak{l}} t'$
PROOF: By ⟨2⟩1, ⟨2⟩2, and assumption 2.

⟨2⟩4. $u' \in \widehat{T}_{EL}([\![P]\!])$
PROOF: By ⟨2⟩2, ⟨2⟩3, and Lemma. B.3.

⟨2⟩5. Q.E.D.
PROOF: By ⟨2⟩3 and ⟨2⟩4.

⟨1⟩2. Q.E.D.
PROOF: By ⟨1⟩1 and definition of $\mathrm{BSP}_{\mathfrak{r}\mathfrak{h}}$ (Eq. (18)).