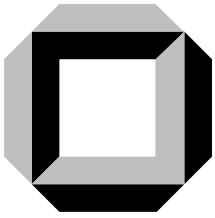
Extending Dynamic Logic for Reasoning about Evolving Algebras

Arno Schönegge

Interner Bericht Nr. 49/95

November 1995



Universität Karlsruhe

Fakultät für Informatik

Institut für Logik, Komplexität und Deduktionssysteme

Extending Dynamic Logic for Reasoning about Evolving Algebras

Arno Schönegge

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
D-76128 Karlsruhe, Germany
email: schoenegge@ira.uka.de
WWW: http://i11www.ira.uka.de/~schoeneg/

Abstract

The aim of this paper is to provide a logic for reasoning about evolving algebras [13, 14]. This is done by extending a variant of dynamic logic [10, 18] with additional program constructs: update of functions, extension of universes, and simultaneous execution. A calculus for this extended dynamic logic can be obtained from a sequent calculus for (not extended) dynamic logic only by adding further rules, but without modifications of original rules. This gives us reason to hope that the KIV system (Karlsruhe Interactive Verifier) [21] can be turned into a tool for reasoning about evolving algebras only by extending it, i.e. without (substantially) modifying existing code.

1 Introduction

Evolving algebras, introduced by Gurevich [13, 14], are abstract machines mainly applied for the specification of several programming languages (e.g. Prolog [2, 7], Occam [16], C [15]) and of real and virtual architectures (e.g. APE [3], PVM [5]).

The usefulness of reasoning about evolving algebras is beyond question. For instance, certain properties of a single evolving algebra can only be guaranteed by formal proof (e.g. determinism, absence of clashes, or so-called integrity constraints [13]). Another important thing is to prove relations between evolving algebras (e.g. equivalence). This is the technique of a proof of the WAM-compiler correctness [6], and some similar work (e.g. [4]). These are only a few examples in the range of applications of reasoning about evolving algebras. However, to our knowledge, up to now no (powerful) tool supporting the construction of formal proofs about evolving algebras exists.[1] This paper aims to make first steps towards such a tool. The basic idea is to get a deduction system for evolving algebras by appropriately modifying the KIV system [21].[2] This modification basically means a kind of extension of the logic underlying the KIV system, which is a variant of dynamic logic [17, 10, 18]. In this paper we provide syntax, semantics and a sequent calculus for such an extension of dynamic logic. So, our work can be thought of as a theoretical foundation for adapting the KIV system for reasoning about evolving algebras.

We start out from the definition of a variant of dynamic logic which is given in the next section. In section 3 this definition is extended by three additional constructs. The resulting extended dynamic logic (EDL) can be used to represent (statements about) evolving algebras. This is explained in Section 4. First steps towards a calculus for EDL are made in section 5. Finally, in the last section we draw conclusions and report on related work.

2 Basic Dynamic Logic

This section presents some basic definitions, especially the variant of dynamic logic we start from (which is quite close to a subset of the logic used in the KIV system).

[1] It should be mentioned that it is possible to reason about evolving algebras in some existing proof systems by 'coding' evolving algebras in the logic underlying the system. This approach was taken e.g. by Schellhorn [22] while doing the WAM case-study [6] in the KIV system.

[2] The KIV system (Karlsruhe Interactive Verifier) is an advanced tool for development of correct software. Especially, it supports interactive, evolutionary construction of (complicated) proofs.

Definition 2.1 (notions for sets of tuples)
Given a set A, $n \in \mathbb{N}$, the set of n-tuples of A is denoted by A^n. We use $a_1 \cdots a_n$ or $(a_1, \ldots, a_n)$ as notations for tuples; the empty tuple (i.e. $n = 0$) is written λ. Furthermore, we set:

- $A^+ := \bigcup_{n \in \mathbb{N} \setminus \{0\}} A^n$
- $A^* := \bigcup_{n \in \mathbb{N}} A^n$.

For a set of (so-called) indices I and sets A_i, $i \in I$, the family (or system) of the sets A_i is denoted by $(A_i)_{i \in I}$. For $i_1 \cdots i_n \in I^n$ we define:[3]

- $A_{i_1 \cdots i_n} := A_{i_1} \times \cdots \times A_{i_n}$.

Definition 2.2 (signatures)
A **signature** $SIG = (S, F)$ consists of a finite set S of **sorts** and a set F of **function symbols** where F is the disjoint union of (possibly infinite[4]) sets $F_{\underline{s},s}$ with $\underline{s} \in S^*$, $s \in S$. For $\underline{s} = s_1 \cdots s_n$, $F_{\underline{s},s}$ is the set of all n-ary function symbols from sorts $s_1 \cdots s_n$ to sort s.

A system of **variables** for a signature $SIG = (S, F)$ is a family $X = (X_s)_{s \in S}$ of (possibly infinite) pairwise disjoint sets X_s.

Definition 2.3 (terms)
Given $SIG = (S, F)$ and a system X of variables for SIG, the family $T_F(X) = \big(T_{F,s}(X)\big)_{s \in S}$ of **terms** over SIG and X is defined as the least family of sets such that

- $\lambda \in T_{F,\lambda}(X)$,
- $X_s \subseteq T_{F,s}(X)$ for every $s \in S$, and
- $f\underline{t} \in T_{F,s}(X)$ for every $\underline{s} \in S^*$, $s \in S$, $f \in F_{\underline{s},s}$, $\underline{t} \in T_{F,\underline{s}}(X)$.

Definition 2.4 (algebras, valuations)
For a signature $SIG = (S, F)$ a SIG-**algebra** $\mathcal{A}$ is a tuple, written $\mathcal{A} = \big((A_s)_{s \in S}, (f_{\mathcal{A}})_{f \in F}\big)$, where $(A_s)_{s \in S}$ is a family of non-empty carrier sets[5] (domain) and $(f_{\mathcal{A}})_{f \in F}$ is a family of interpretations for the function symbols in F. Given $f \in F_{\underline{s},s}$ with $\underline{s} \in S^*, s \in S$, then $f_{\mathcal{A}}$ is a *total* function from $A_{\underline{s}}$ into A_s. The set of all SIG-algebras is denoted by $Alg(SIG)$.

For a system X of variables for SIG and an $\mathcal{A} \in Alg(SIG)$ an $\mathcal{A}$-**valuation** $v = (v_s)_{s \in S}$ is a family of total functions $v_s : X_s \to A_s$. $Val(X, \mathcal{A})$ is the set of all such $\mathcal{A}$-valuations. For $s \in S$, $x \in X_s$, and $a \in A_s$, we write $v[x \leftarrow a]$ for the modification of v which assigns a to x and is otherwise the same as v.

[3] Notice that $A_\lambda = \{\lambda\}$.

[4] We allow infinite sets here, because proving with the calculus we introduce in section 5 requires that one has new function symbols and new variables in reserve.

[5] Note, that the carrier sets for different sorts are not required to be disjoint.

Definition 2.5 (disjoint signatures, sum of algebras)
Two signatures $SIG = (S, F)$, $SIG' = (S', F')$ are said to be **disjoint**, written $SIG \cap SIG' = \emptyset$, if $S \cap S' = \emptyset$ and $F \cap F' = \emptyset$. In this case $SIG \cup SIG' := (S \cup S', F \cup F')$ is again a signature, and from $\mathcal{A} \in Alg(SIG)$, $\mathcal{B} \in Alg(SIG')$ a $(SIG \cup SIG')$-algebra

$$\mathcal{A} + \mathcal{B} \quad := \quad \left(\left((A+B)_s \right)_{s \in S \cup S'}, \left(f_{\mathcal{A}+\mathcal{B}} \right)_{f \in F \cup F'} \right)$$

can be constructed by (cf. in [8]: amalgamated sum of algebras)

$$(A+B)_s \quad := \quad \begin{cases} A_s & \text{, if } s \in S \\ B_s & \text{, if } s \in S' \end{cases}$$

$$f_{(\mathcal{A}+\mathcal{B})} := \begin{cases} f_{\mathcal{A}} & \text{, if } f \in F \\ f_{\mathcal{B}} & \text{, if } f \in F'. \end{cases}$$

Definition 2.6 (semantics of terms)
Let $SIG = (S, F)$ be a signature, X a system of variables for SIG, $\mathcal{A} \in Alg(SIG)$, and $v \in Val(X, \mathcal{A})$. The **value** $t_{v,\mathcal{A}}$ of a term $t \in \bigcup_{s \in S} T_{F,s}(X)$ in $\mathcal{A}$ under v is given by:

- $x_{v,\mathcal{A}} := v_s(x)$ for $x \in X_s$, $s \in S$;
- $(f\underline{t})_{v,\mathcal{A}} := f_{\mathcal{A}}(\underline{t}_{v,\mathcal{A}})$ for $\underline{s} \in S^*$, $s \in S$, $f \in F_{\underline{s},s}$, $\underline{t} = t_1 \cdots t_n \in T_{F,\underline{s}}(X)$

where $(t_1 \cdots t_n)_{v,\mathcal{A}} := t_{1\,v,\mathcal{A}} \cdots t_{n\,v,\mathcal{A}}$.

Definition 2.7 (atomic formulas)
Let $SIG = (S, F)$ and X be a system of variables for SIG. The set $AT(SIG, X)$ of **atomic formulas** over SIG and X is the least set satisfying:

- **false** $\in AT(SIG, X)$
- for $s \in S$ and $t_1, t_2 \in T_{F,s}(X)$ is $(t_1 = t_2) \in AT(SIG, X)$.

Definition 2.8 (boolean expressions)
Let $SIG = (S, F)$ and X be a system of variables for SIG. The set $BXP(SIG, X)$ of **boolean expressions** over SIG and X is the least set satisfying:

- $AT(SIG, X) \subseteq BXP(SIG, X)$
- for $\varphi, \psi \in BXP(SIG, X)$ is $(\varphi \to \psi) \in BXP(SIG, X)$.

In the following definition so-called *counters* are introduced. This is a special built-in data structure used for inductive arguments about while loops (see definition 2.10 and appendix A) (cf. [18]).

Definition 2.9 (extension by counters)
For the **counter signature** $CSIG = (\{ctr\}, F_{\lambda,ctr} \cup F_{ctr,ctr})$ with $F_{\lambda,ctr} := \{czero\}$ and $F_{ctr,ctr} := \{csucc\}$ we fix a standard algebra $\mathcal{A}_{ctr}$ with the carrier $A_{ctr} = \mathbb{N}$, and which gives $czero$ and $csucc$ their usual meanings, i.e. zero and successor-function on natural numbers.

We assume all signatures SIG considered in the following to be disjoint from $CSIG$. So the **standard extension** $SIG_+ := SIG \cup CSIG$ of SIG is well-defined. Correspondingly, we fix a countable, infinite set $X_{ctr} := \{\kappa, \kappa_0, \kappa_1, \ldots\}$ of variables for sort ctr, and assume all other sets of variables considered in the following, to be disjoint from X_{ctr}. So we can define the standard extension $X_+ := (X_s)_{s \in S \cup \{ctr\}}$ of a system $X = (X_s)_{s \in S}$ of variables for SIG. The standard extension of an $\mathcal{A} \in Alg(SIG)$ is the SIG_+-algebra $\mathcal{A}_+ := \mathcal{A} + \mathcal{A}_{ctr}$.

Definition 2.10 (programs)
Given a signature $SIG = (S, F)$, X a system of variables for SIG, and $SIG_+ = (S_+, F_+)$ and X_+ their standard extensions by counters. The set $PROG(SIG, X)$ of **programs** over SIG and X is the least set satisfying:[6]

- (skip)
  skip $\in PROG(SIG, X)$
- (assignment)
  for $s \in S$, $x \in X_s$, and $t \in T_{F,s}(X)$ is
  $(x := t) \in PROG(SIG, X)$
- (nondeterministic choice)
  for $\alpha, \beta \in PROG(SIG, X)$ is
  $(\alpha \cup \beta) \in PROG(SIG, X)$
- (composition)
  for $\alpha, \beta \in PROG(SIG, X)$ is
  $(\alpha; \beta) \in PROG(SIG, X)$
- (conditional)
  for $\alpha, \beta \in PROG(SIG, X)$ and $\epsilon \in BXP(SIG, X)$ is
  if ϵ **then** α **else** $\beta \in PROG(SIG, X)$
- (while)
  for $\alpha \in PROG(SIG, X)$ and $\epsilon \in BXP(SIG, X)$ is
  while ϵ **do** $\alpha \in PROG(SIG, X)$

[6] Notice that there are no counters involved in assignments or conditions ϵ.

- (bounded loop)
  for $\alpha \in PROG(SIG, X)$ and $t \in T_{F_+,ctr}(X_+)$ is
  loop α **times** $t \in PROG(SIG, X)$.

Definition 2.11 (dynamic logic formulas)
Let $SIG = (S, F)$, X a system of variables for SIG, and $SIG_+ = (S_+, F_+)$ and X_+ their standard extensions by counters. The set $DL(SIG, X)$ of **dynamic logic formulas** over SIG and X is the least set satisfying:

- $AT(SIG_+, X_+) \subseteq DL(SIG, X)$
- for $\varphi, \psi \in DL(SIG, X)$ is
  $(\varphi \to \psi) \in DL(SIG, X)$
- for $\varphi \in DL(SIG, X)$, $s \in S_+$, and $x \in X_s$ is
  $\forall x.\varphi \in DL(SIG, X)$
- for $\alpha \in PROG(SIG, X)$ and $\varphi \in DL(SIG, X)$ is
  $[\alpha]\varphi \ \in DL(SIG, X)$.

Definition 2.12 (abbreviations)
We use the following abbreviations:

$$
\begin{array}{lcl}
\neg\,\varphi & :\equiv & (\varphi \to \mathbf{false}) \\
t_1 \neq t_2 & :\equiv & \neg\, t_1 = t_2 \\
\mathbf{true} & :\equiv & \neg\, \mathbf{false} \\
(\varphi \vee \psi) & :\equiv & (\neg\, \varphi \to \psi) \\
(\varphi \wedge \psi) & :\equiv & \neg\, (\varphi \to \neg\, \psi) \\
(\varphi \leftrightarrow \psi) & :\equiv & ((\varphi \to \psi) \wedge (\psi \to \varphi)) \\
\exists x.\varphi & :\equiv & \neg\, \forall x.\neg\, \varphi \\
\langle\alpha\rangle\varphi & :\equiv & \neg\, [\alpha] \neg\, \varphi \\
\mathbf{abort} & :\equiv & \mathbf{while\ true\ do\ skip} \\
\mathbf{if}\ \epsilon\ \mathbf{then}\ \alpha & :\equiv & \mathbf{if}\ \epsilon\ \mathbf{then}\ \alpha\ \mathbf{else\ skip}.
\end{array}
$$

Definition 2.13 (semantics of programs and formulas)
Let $SIG = (S, F)$ a signature with a system X of variables, and $\mathcal{A} \in Alg(SIG)$. Let further $SIG_+ = (S_+, F_+)$, X_+, and $\mathcal{A}_+$ be the corresponding standard extensions and $v, v' \in Val(X_+, \mathcal{A}_+)$. For $\varphi \in DL(SIG, X)$ we write $\mathcal{A}, v \models \varphi$ if φ **is true in** $\mathcal{A}$ **under** v, and $\mathcal{A}, v \not\models \varphi$ otherwise. For $\alpha \in PROG(SIG, X)$ we write $v\, [\![\alpha]\!]_{\mathcal{A}}\, v'$ if v' is a valuation that can be reached from v by executing α interpreted under $\mathcal{A}$. The relation $[\![\alpha]\!]_{\mathcal{A}} \subseteq Val(X_+, \mathcal{A}_+) \times Val(X_+, \mathcal{A}_+)$ describes the input-output behavior of α under $\mathcal{A}$. These notions are defined simultaneously[7] as follows:

[7] Since the semantics of programs and formulas depend on each other, we have to define it simultaneously.

- $\mathcal{A}, v \not\models \mathbf{false}$
- $\mathcal{A}, v \models t_1 = t_2$ iff $t_{1\,v,\mathcal{A}_+} = t_{2\,v,\mathcal{A}_+}$
- $\mathcal{A}, v \models \varphi \to \psi$ iff ($\mathcal{A}, v \models \varphi$ implies $\mathcal{A}, v \models \psi$)
- $\mathcal{A}, v \models \forall x.\varphi$ (where $x \in X_s$ for some $s \in S_+$) iff
  $\mathcal{A}, v[x \leftarrow a] \models \varphi$ for all $a \in A_s$
- $\mathcal{A}, v \models [\alpha]\varphi$ iff $\mathcal{A}, v' \models \varphi$ for all $v' \in Val(X_+, \mathcal{A}_+)$ with $v \,[\![\alpha]\!]_{\mathcal{A}}\, v'$
- $v \,[\![\mathbf{skip}]\!]_{\mathcal{A}}\, v'$ iff $v = v'$
- $v \,[\![x := t]\!]_{\mathcal{A}}\, v'$ iff $v' = v[x \leftarrow t_{v,\mathcal{A}}]$
- $v \,[\![(\alpha \cup \beta)]\!]_{\mathcal{A}}\, v'$ iff ($v \,[\![\alpha]\!]_{\mathcal{A}}\, v'$ or $v \,[\![\beta]\!]_{\mathcal{A}}\, v'$)
- $v \,[\![(\alpha; \beta)]\!]_{\mathcal{A}}\, v'$ iff
  there is a $v'' \in Val(X_+, \mathcal{A}_+)$ such that $v \,[\![\alpha]\!]_{\mathcal{A}}\, v''$ and $v'' \,[\![\beta]\!]_{\mathcal{A}}\, v'$
- $v \,[\![\mathbf{if}\ \epsilon\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]\!]_{\mathcal{A}}\, v'$ iff
  either $\mathcal{A}, v \models \epsilon$ and $v \,[\![\alpha]\!]_{\mathcal{A}}\, v'$
  or else $\mathcal{A}, v \not\models \epsilon$ and $v \,[\![\beta]\!]_{\mathcal{A}}\, v'$
- $v \,[\![\mathbf{while}\ \epsilon\ \mathbf{do}\ \alpha]\!]_{\mathcal{A}}\, v'$ iff
  $\mathcal{A}, v' \not\models \epsilon$ and there are some $v_0, \ldots, v_n \in Val(X_+, \mathcal{A}_+)$ such that $v_0 = v$, $v_n = v'$, and $\mathcal{A}, v_i \models \epsilon$ and $v_i \,[\![\alpha]\!]_{\mathcal{A}}\, v_{i+1}$ for $i \in \{0, \ldots, n \Leftrightarrow 1\}$
- $v \,[\![\mathbf{loop}\ \alpha\ \mathbf{times}\ t]\!]_{\mathcal{A}}\, v'$ iff
  with $n := t_{v,\mathcal{A}_+}$ there are some $v_0, \ldots, v_n \in Val(X_+, \mathcal{A}_+)$ such that $v_0 = v$, $v_n = v'$, and $v_i \,[\![\alpha]\!]_{\mathcal{A}}\, v_{i+1}$ for $i \in \{0, \ldots, n \Leftrightarrow 1\}$.

3 Extending Dynamic Logic

In this section we define syntax and semantics of EDL (Extended Dynamic Logic), which extends dynamic logic by three additional program constructs: extension of domains (universes), function update, and simultaneous execution.

Definition 3.1 (syntax of EDL)
Given a signature $SIG = (S, F)$ and X a system of variables for SIG. The set $EPROG(SIG, X)$ of **extended programs** over SIG and X and the set $EDL(SIG, X)$ of **extended dynamic logic formulas** over SIG and X are defined just as $PROG(SIG, X)$ and $DL(SIG, X)$ in definitions 2.10 and 2.11, except that there are the following additional program constructs:

- (extension of domains)[8]
  for $s \in S$, $x \in X_s$, and $\alpha \in EPROG(SIG, X)$ is
  extend s **by** x **with** $\alpha \in EPROG(SIG, X)$

- (function update)
  for $\underline{s} \in S^*$, $s \in S$, $f \in F_{\underline{s},s}$, $\underline{t} \in T_{F,\underline{s}}(X)$, and $t \in T_{F,s}(X)$ is
  $(f\underline{t} := t) \in EPROG(SIG, X)$

- (simultaneous execution)
  for $\alpha, \beta \in EPROG(SIG, X)$ is
  $(\alpha, \beta) \in EPROG(SIG, X)$.

Agreement (algebras with error elements). The most intuitive semantics of the **extend**-construct is defined using *partial* algebras. However, on the other hand, allowing partial algebras would cause substantial changes in the semantics (and calculus) of basic dynamic logic. Thus we decided[9] to 'simulate' partiality using explicit error elements, i.e. we use total algebras (as defined in 2.4) but demand each domain A_s to contain a special element $UNDEF_s$.[10]

Definition 3.2 (extension of domains)
Let $SIG = (S, F)$ and $\mathcal{A} = \left((A_s)_{s\in S}, (f_{\mathcal{A}})_{f\in F}\right)$, $\mathcal{A}' = \left((A'_s)_{s\in S}, (f_{\mathcal{A}'})_{f\in F}\right)$ two SIG-algebras (containing error elements $UNDEF_s$ for each sort $s \in S$). Let further $s_0 \in S$ and $a \in A'_{s_0}$. If

$$A'_s = \begin{cases} A_s \cup \{a\} & \text{, if } s = s_0 \\ A_s & \text{, otherwise} \end{cases}$$

and for $f \in F_{\underline{s},s}$ ($\underline{s} \in S^*$, $s \in S$)

$$f_{\mathcal{A}'}(\underline{a}) = \begin{cases} f_{\mathcal{A}}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}} \\ UNDEF_s & \text{, otherwise} \end{cases}$$

then $\mathcal{A}'$ is called the **extension of** $\mathcal{A}$ **by** a **at** s_0, written $\mathcal{A}' = \mathcal{A} +_{s_0} \{a\}$.

---

[8] Because the sort information is already attached to the variable symbols the explicit reference to the sort s is a kind of redundancy (which may increase readability). Another syntax for the **extend**-construct, which emphasizes the relationship to local variable bindings is used in [19] and looks like **let** $x =$ **new**(s) **in** α **endlet**.

We use **extend** s **by** $x_1, \ldots, x_n$ **with** α as an abbreviation for
extend s **by** x_1 **with** ... **extend** s **by** x_n **with** α.

[9] Essentially, this decision was enforced by our overall aim, which is to use a modification of the KIV system for reasoning about evolving algebras. Using error elements instead of proper partiality minimizes the modifications of the KIV system (and is even the technique used in basic introductions to evolving algebras [13, 14]).

[10] Notice that we do not explicitly forbid non-strict functions in algebras. However, if strictness is desired it can be proved as a property (preservation of strictness).

Definition 3.3 (update of functions)
Let $SIG = (S, F)$ a signature, $\mathcal{A} = \left((A_s)_{s\in S}, (f_{\mathcal{A}})_{f\in F}\right)$ a SIG-algebra, $f_0 \in F_{\underline{s},s}$ ($\underline{s} \in S^*$, $s \in S$), $\underline{a}_0 \in A_{\underline{s}}$, and $a_0 \in A_s$. Then we call the SIG-algebra $\mathcal{A}' := \left((A_s)_{s\in S}, (f_{\mathcal{A}'})_{f\in F}\right)$ where

$$f_{\mathcal{A}'}(\underline{a}) := \begin{cases} a_0 & \text{, if } f \equiv f_0 \text{ and } \underline{a} = \underline{a}_0 \\ f_{\mathcal{A}}(\underline{a}) & \text{, otherwise} \end{cases}$$

the **update of $\mathcal{A}$ by setting $f_0(\underline{a}_0)$ to a_0**, and denote it by $\mathcal{A}[f_0(\underline{a}_0) \leftarrow a_0]$.

The crucial point of the program constructs extension of domains and function update is that executing them affects not only the (e)valuation of variables, but also the evaluation of function symbols (i.e. the evaluation of terms) and the universes (i.e. the evaluation of quantifiers). Therefore, we define states as pairs of algebras and valuations (and not merely as valuations as it is common practice in dynamic logic).

Definition 3.4 (states, state changes)
Let SIG a signature and X a system of variables for SIG. Then the set $STATE(SIG, X)$ of **states** over SIG and X is defined by

$$STATE(SIG, X) \quad := \quad \{(\mathcal{A}_+, v) | \mathcal{A} \in Alg(SIG), v \in Val(X_+, \mathcal{A}_+)\}.$$

Definition 3.5 (operations on states)
Let $SIG = (S, F)$ a signature, X a system of variables for SIG, and $(\mathcal{A}, v) \in STATE(SIG, X)$.

$$\begin{array}{lcll} (\mathcal{A}, v) +_s \{a\} & := & (\mathcal{A} +_s \{a\}, v) & \\ (\mathcal{A}, v)[x \leftarrow a] & := & (\mathcal{A}, v[x \leftarrow a]) & \text{for } x \in X_s,\ a \in A_s \\ (\mathcal{A}, v)[f(\underline{a}) \leftarrow a] & := & (\mathcal{A}[f(\underline{a}) \leftarrow a], v) & \text{for } f \in F_{\underline{s},s},\ \underline{a} \in A_{\underline{s}},\ a \in A_s. \end{array}$$

In order to declare the semantics of simultaneous execution we define a *join* operator on states:[11] the join of two states st', st'' modulo st, written $st' \oplus_{st} st''$, is the state that arises when the effects of the state change $st \rightsquigarrow st'$ and the state change $st \rightsquigarrow st''$ are combined, provided this is consistent. Informally, consistency of two state changes means that there are no *clashes*, i.e. that

- there are no extensions of the same domain by the same value in both state changes, i.e. new elements in st' differ from new elements in st'',

[11] This definition is partly adopted from Rix Groenboom and Gerard Renardel de Lavalette de [12].

- there are no conflicting function updates, i.e. if in both state changes a value of a function value is changed, then these updates of the function value are the same, and
- there are no conflicting assignments to variables, i.e. if in both state changes a value of a variable is changed, then these assignments to the variable are the same.

Definition 3.6 (consistency of state changes)
Let $SIG = (S, F)$ a signature, X a system of variables for SIG, and $st = (\mathcal{A}, v)$, $st' = (\mathcal{A}', v')$, and $st'' = (\mathcal{A}'', v'')$ three states from $STATE(SIG, X)$. If

- $A'_s \cap A''_s = A_s$ for all $s \in S$,
- $f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}''}(\underline{a})$ for all $f \in F_{\underline{s},s}$, $\underline{a} \in A_{\underline{s}}$, and
- $v'_s(x) = v_s(x)$ or $v''_s(x) = v_s(x)$ or $v'_s(x) = v''_s(x)$ for all $x \in X_s, s \in S$

we say that the **state changes $st \rightsquigarrow st'$ and $st \rightsquigarrow st''$ are consistent**.

Definition 3.7 (join of states)
Let $SIG = (S, F)$ a signature, X a system of variables for SIG, and $st = (\mathcal{A}, v)$, $st' = (\mathcal{A}', v')$, and $st'' = (\mathcal{A}'', v'')$ three states from $STATE(SIG, X)$. If the state changes $st \rightsquigarrow st'$ and $st \rightsquigarrow st''$ are not consistent then the **join of st' and st'' modulo st**, written $st' \oplus_{st} st''$, is undefined; otherwise it is a state from $STATE(SIG, X)$, defined by:

$$\begin{aligned} st' \oplus_{st} st'' &:= (\mathcal{A}^{\oplus}, v^{\oplus}) \\ \mathcal{A}^{\oplus} &:= \left((A^{\oplus}_s)_{s \in S}, (f_{\mathcal{A}^{\oplus}})_{f \in F}\right) \end{aligned}$$

with $(s \in S, f \in F_{\underline{s},s}, x \in X_s)$

$$A^{\oplus}_s := A'_s \cup A''_s$$

$$f_{\mathcal{A}^{\oplus}}(\underline{a}) := \begin{cases} UNDEF_s & \text{, if } \underline{a} \in A^{\oplus}_{\underline{s}} \setminus (A'_{\underline{s}} \cup A''_{\underline{s}}) \\ f_{\mathcal{A}'}(\underline{a}) & \text{, if } \underline{a} \in A'_{\underline{s}} \setminus A_{\underline{s}} \\ f_{\mathcal{A}''}(\underline{a}) & \text{, if } \underline{a} \in A''_{\underline{s}} \setminus A_{\underline{s}} \\ f_{\mathcal{A}'}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}} \text{ and } f_{\mathcal{A}'}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a}) \\ f_{\mathcal{A}''}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}} \text{ and } f_{\mathcal{A}''}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a}) \\ f_{\mathcal{A}}(\underline{a}) & \text{, otherwise} \end{cases}$$

$$v^{\oplus}_s(x) := \begin{cases} v'_s(x) & \text{, if } v'_s(x) \neq v_s(x) \\ v''_s(x) & \text{, if } v''_s(x) \neq v_s(x) \\ v_s(x) & \text{, otherwise.} \end{cases}$$

The reader may check that the join is in fact well-defined provided the state changes are consistent.

Fact 3.8 (basic properties of join)
Let $SIG = (S, F)$ a signature, X a system of variables for SIG, and $st = (\mathcal{A}, v)$, $st' = (\mathcal{A}', v')$, $st'' = (\mathcal{A}'', v'')$, and $st''' = (\mathcal{A}''', v''')$ states from $STATE(SIG, X)$.

(a) commutativity:
If the state changes $st \rightsquigarrow st'$ and $st \rightsquigarrow st''$ are consistent, then

$$st' \oplus_{st} st'' \quad = \quad st'' \oplus_{st} st'.$$

(b) associativity:
The state changes $st \rightsquigarrow st'$ and $st \rightsquigarrow st''$, and the state changes $st \rightsquigarrow (st' \oplus_{st} st'')$ and $st \rightsquigarrow st'''$ are consistent iff the state changes $st \rightsquigarrow st''$ and $st \rightsquigarrow st'''$, and the state changes $st \rightsquigarrow st'$ and $st \rightsquigarrow (st'' \oplus_{st} st''')$ are consistent. In this case it holds

$$(st' \oplus_{st} st'') \oplus_{st} st''' \quad = \quad st' \oplus_{st} (st'' \oplus_{st} st''').$$

(c) neutral element:
If the state changes $st \rightsquigarrow st$ and $st \rightsquigarrow st'$ are consistent, then

$$st \oplus_{st} st' \quad = \quad st'.$$

Proof. The proof for this fact, especially for (b) (which is in no way obvious) is quite technical and long. So we have postponed it to appendix B.2. ■

We are now prepared to define the semantics of programs and formulas of EDL. One major difference between definition 2.13 is that the input-output behavior $[\![\alpha]\!]$ of a program α is not a binary relation on valuations only, but on states.

Definition 3.9 (semantics of programs and formulas)
Let $SIG = (S, F)$ a signature with a system X of variables and $st, st' \in STATE(SIG, X)$, $st = (\mathcal{A}, v)$. For $\varphi \in EDL(SIG, X)$ we write $st \models \varphi$ if φ **is true in** st, and $st \not\models \varphi$ otherwise. For $\alpha \in EPROG(SIG, X)$ we write $st\,[\![\alpha]\!]\,st'$ if the state st' can be reached from state st by executing α. The relation $[\![\alpha]\!] \subseteq STATE(SIG, X) \times STATE(SIG, X)$ describes the input-output behavior of α. These notions are defined simultaneously as follows:

- $st \not\models$ **false**

- $st \models t_1 = t_2$ iff[12] $st(t_1) = st(t_2)$
- $st \models \varphi \to \psi$ iff ($st \models \varphi$ implies $st \models \psi$)
- $st \models \forall x.\varphi$ (where $x \in X_s$ for some $s \in S$) iff
  $st[x \leftarrow a] \models \varphi$ for all $a \in A_s$
- $st \models [\alpha]\varphi$ iff $st' \models \varphi$ for all $st' \in STATE(SIG, X)$ with $st \,[\![\alpha]\!]\, st'$
- $st \,[\![\mathbf{skip}]\!]\, st'$ iff $st = st'$
- $st \,[\![x := t]\!]\, st'$ iff $st' = st[x \leftarrow st(t)]$
- $st \,[\![f\underline{t} := t]\!]\, st'$ iff $st' = st[f(st(\underline{t})) \leftarrow st(t)]$
- $st \,[\![(\alpha \cup \beta)]\!]\, st'$ iff ($st \,[\![\alpha]\!]\, st'$ or $st \,[\![\beta]\!]\, st'$)
- $st \,[\![(\alpha; \beta)]\!]\, st'$ iff
  there is some $st'' \in STATE(SIG, X)$
  such that $st \,[\![\alpha]\!]\, st''$ and $st'' \,[\![\beta]\!]\, st'$
- $st \,[\![(\alpha, \beta)]\!]\, st'$ iff[13]
  there are some $st_\alpha, st_\beta \in STATE(SIG, X)$ such that
  $st \,[\![\alpha]\!]\, st_\alpha$, $st \,[\![\beta]\!]\, st_\beta$, and the state changes
  $st \rightsquigarrow st_\alpha$ and $st \rightsquigarrow st_\beta$ are consistent with $st' = st_\alpha \oplus_{st} st_\beta$
- $st \,[\![\mathbf{if}\ \epsilon\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]\!]\, st'$ iff
  either $st \models \epsilon$ and $st \,[\![\alpha]\!]\, st'$
  or else $st \not\models \epsilon$ and $st \,[\![\beta]\!]\, st'$
- $st \,[\![\mathbf{extend}\ s\ \mathbf{by}\ x\ \mathbf{with}\ \alpha]\!]\, st'$ (where $x \in X_s$) iff
  there is some $st'' = (\mathcal{A}'', v'') \in STATE(SIG, X)$ and an[14] $a \in A''_s \setminus A_s$
  such that $\Big((st +_s \{a\})[x \leftarrow a]\Big)[\![\alpha]\!] st''$ and[15] $st' = st''[x \leftarrow st(x)]$
- $st \,[\![\mathbf{while}\ \epsilon\ \mathbf{do}\ \alpha]\!]\, st'$ iff
  $st' \not\models \epsilon$ and there are some $st_0, \ldots, st_n \in STATE(SIG, X)$ such that
  $st_0 = st$, $st_n = st'$, and $st_i \models \epsilon$ and $st_i \,[\![\alpha]\!]\, st_{i+1}$ for $i \in \{0, \ldots, n \Leftrightarrow 1\}$

[12] For $st = (\mathcal{A}, v)$ we abbreviate $st(t) := t_{v,\mathcal{A}}$.

[13] Especially combined with composition or while loops this is not simultaneous execution (as one intuitively might think of), because we look at the executions of α and β as separate black boxes, i.e. the intermediate states passed through while executing α and the intermediate states passed through while executing β are regarded as completely independent from each other.

[14] Notice, that there is no operation for discarding elements from domains.

[15] After the execution of the **extend** construct x is bound to its original value.

- $st\,[\![\textbf{loop } \alpha \textbf{ times } t]\!]_{\mathcal{A}}\,st'$ iff
  with $n := st(t)$ there are some $st_0, \ldots, st_n \in STATE(SIG, X)$ such that $st_0 = st$, $st_n = st'$, and $st_i\,[\![\alpha]\!]\,st_{i+1}$ for $i \in \{0, \ldots, n \Leftrightarrow 1\}$.

The following theorem illustrates that the 'simultaneous execution' construct behaves as simultaneous execution where the programs to be executed can be viewed as black boxes, i.e. there is no communication despite shared variables.

Theorem 3.10 (properties of simultaneous execution semantics)
Let SIG a signature, X a system of variables for SIG, and $\alpha, \alpha_1, \alpha_2, \alpha_3, \beta \in EPROG(SIG, X)$.

(a) $[\![(\alpha, \beta)]\!] = [\![(\beta, \alpha)]\!]$

(b) $[\![((\alpha_1, \alpha_2), \alpha_3)]\!] = [\![(\alpha_1, (\alpha_2, \alpha_3))]\!]$

(c) $[\![(\textbf{skip}, \alpha)]\!] = [\![\alpha]\!]$

(d) $[\![((\alpha_1 \cup \alpha_2), \beta)]\!] = [\![((\alpha_1, \beta) \cup (\alpha_2, \beta))]\!]$

(e) $[\![(\textbf{if } \epsilon \textbf{ then } \alpha_1 \textbf{ else } \alpha_2, \beta)]\!] = [\![\textbf{if } \epsilon \textbf{ then } (\alpha_1, \beta) \textbf{ else } (\alpha_2, \beta)]\!]$

(f) $[\![(\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha, \beta)]\!] = [\![\textbf{extend } s \textbf{ by } x' \textbf{ with } (\alpha_x^{x'}, \beta)]\!]$
where[16] *$x' \in X_s$ is a variable not occurring in α or β.*

Proof. (a), (b), and (c) follow from fact 3.8. (d) and (e) can be shown simply by unfolding the definitions. The proof of (f) is more complicated and requires some auxiliary lemmata. It is given in full detail in appendix B.3. ■

4 Representing Evolving Algebras

This section is intended to describe how (statements about) evolving algebras can be formalized in (a subset of) EDL. We start with a notion of evolving algebra rules (which is quite general: especially one is allowed to arbitrarily nest indeterministic choice and simultaneous execution). The following definition essentially coincides with the definition of extended programs (cf. 3.1), but without **while** and **loop**.

Definition 4.1 (evolving algebra rules)
Let $SIG = (S, F)$ a signature and X a system of variables for SIG. The set $EAR(SIG, X)$ of **evolving algebra rules** over SIG and X is the least set satisfying:

[16] $\alpha_x^{x'}$ denotes the result of syntactically replacing x by x' in α.

- (skip)[17]
  skip $\in EAR(SIG, X)$

- (assignment)
  for $s \in S$, $x \in X_s$, and $t \in T_{F,s}(X)$ is
  $(x := t) \in EAR(SIG, X)$

- (function update)
  for $\underline{s} \in S^*$, $s \in S$, $f \in F_{\underline{s},s}$, $\underline{t} \in T_{F,\underline{s}}(X)$, and $t \in T_{F,s}(X)$ is
  $(f\underline{t} := t) \in EAR(SIG, X)$

- (nondeterministic choice)
  for $\alpha, \beta \in EAR(SIG, X)$ is
  $(\alpha \cup \beta) \in EAR(SIG, X)$

- (simultaneous execution)
  for $\alpha, \beta \in EAR(SIG, X)$ is
  $(\alpha, \beta) \in EAR(SIG, X)$

- (conditional)
  for $\alpha, \beta \in EAR(SIG, X)$ and $\epsilon \in BXP(SIG, X)$ is
  if ϵ **then** α **else** $\beta \in EAR(SIG, X)$

- (extension of domains)
  for $s \in S$, $x \in X_s$, and $\alpha \in EAR(SIG, X)$ is
  extend s **by** x **with** $\alpha \in EAR(SIG, X)$.

The semantics of evolving algebra rules is defined just as in definition 3.9.

Formal reasoning about evolving algebras requires a formal representation, i.e. we have to define syntax and semantics for evolving algebras.

Definition 4.2 (evolving algebras)
An **evolving algebra** $EA = (SIG, X, I, F, \alpha)$ consists of a signature SIG, a system X of variables for SIG, a formula $I \in EDL(SIG, X)$, a boolean expression $F \in BXP(SIG, X)$, and an evolving algebra rule $\alpha \in EAR(SIG, X)$. The formula I describes the initial states, F is the stopping condition.[18]

The semantics of an evolving algebra is the set of its runs.

[17] The **skip** construct is needed to simulate **if** ϵ **then** α by **if** ϵ **then** α **else skip**.

[18] The stopping condition F is restricted to be a boolean expression (instead of an arbitrary formula from $EDL(SIG, X)$) because this allows $\neg F$ to be used as condition of a **while** loop or a conditional — as done in the examples on page 17.

Definition 4.3 (runs)
Let $EA = (SIG, X, I, F, \alpha)$ be an evolving algebra. A **terminating run** of EA is a finite sequence $st_0, \ldots, st_n$ of states over SIG and X such that $st_0 \models I$, $st_n \models F$, and $st_k \not\models F$ and $st_k [\![\alpha]\!] st_{k+1}$ for all $k \in \{0, \ldots, n \Leftrightarrow 1\}$. An infinite sequence $st_0, st_1, \ldots$ of states over SIG and X is called a **non-terminating run** of EA if $st_0 \models I$, and $st_k \not\models F$ and $st_k [\![\alpha]\!] st_{k+1}$ for all $k \in \{0, \ldots\}$.

There are several differences to other common definitions of evolving algebras:

- many-sorted signature:
  Instead of modeling sorts by means of predicate symbols (ranging over a so-called super-universe) as e.g. in [13, 14], we prefer to use a many-sorted signature. Thus, we loose a little bit of expressiveness (e.g. allowing objects which are elements of more than one universe), but avoid keeping track of sorting information while constructing proofs.

- initial states:
  Mostly no syntactic representation for initial states is given. In the definition above initial states are restricted to those algebras which can be (uniquely up to isomorphism) described with a finite set of formulas from $EDL(SIG, X)$. However we believe that this class is sufficient to cope with most evolving algebras in practice.[19] While in most definitions an evolving algebra has exactly one initial state, we permit a set of initial states, namely all $st \in STATE(SIG, X)$ with $st \models I$.

- final states:
  In some publications (e.g. [6]) the set of final (terminal) states is implicitly defined to be the states which are reachable from the initial state(s) by applying rules, but in which no further rule is applicable. In other definitions final states are not defined at all. As e.g. in [19] we prefer to make final states explicit, namely as a boolean expression describing a stopping condition. Below we give some illustration for the use of these stopping conditions.

- further rule constructors:
  It is often convenient to allow some further constructors in evolving algebra rules besides the basic ones from definition 4.1. For instance a **let** construct is frequently used. Integrating it in EDL will make no serious problems. Sometimes even sequential execution is useful in

[19] At least all *computable* algebras can be uniquely (up to isomorphism) described by a finite set of EDL formulas (cf. [1, 23]).

evolving algebra rules (e.g. in [6]). EDL already provides such a construct for sequential execution of programs (composition).[20] There are a lot of further constructs that may be desirable in certain applications (cf. [14]: **choose**, **duplicate**, ...). We do not discuss them here.

- one rule only:
  Usually an evolving algebra is defined to contain a finite *set* of rules $\{\alpha_1, \ldots, \alpha_n\}$. In some publications a computation step is defined as firing one, indeterministically chosen rule, in other publications (e.g. [13]) a computation step is defined as firing all rules simultaneously. We avoid excluding one of these approaches, but prefer to combine all rules in a *single* one, which an advocate of the first definition would write as $\alpha := (\alpha_1 \cup \cdots \cup \alpha_n)$, and an advocate of the second definition would write as $\alpha := (\alpha_1, \ldots, \alpha_n)$.

- clash handling:
  In [13] executing conflicting updates is defined to indeterministically choose one of the greatest subsets of non-conflicting updates. Some more recent publications (e.g. [14]) define a clash to behave just like **skip**. However, it is commonly accepted that a detection of an inconsistency should manifest itself in some way. In our opinion it should even not be possible to continue a computation if a clash has occurred. Therefore we define conflicting updates to have no next state, i.e. a clash manifests itself in (local) non-termination.

In the following we discuss how to cope with some of the possible evolving algebra semantics (proposed elsewhere) using the notion of evolving algebras as defined in 4.2. This is done on an example with three rules.

$$\begin{array}{l} \textbf{if } \epsilon_1 \textbf{ then } \alpha_1 \\ \textbf{if } \epsilon_2 \textbf{ then } \alpha_2 \\ \textbf{if } \epsilon_3 \textbf{ then } \alpha_3 \end{array}$$

If a semantics (of evolving algebra computation) is taken, where in each step any of the rules is chosen indeterministically, then we set in $EA = (SIG, X, I, F, \alpha)$:

$$\begin{array}{lcl} F & := & \textbf{false} \\ \alpha & := & (\ (\textbf{if } \epsilon_1 \textbf{ then } \alpha_1) \\ & & \cup\ (\textbf{if } \epsilon_2 \textbf{ then } \alpha_2) \\ & & \cup\ (\textbf{if } \epsilon_3 \textbf{ then } \alpha_3)\ ). \end{array}$$

[20] However, the calculus presented in this paper (cf. section 5) is not sufficient for sequential execution involved in simultaneous execution.

If the semantics is to indeterministically apply rules until no rule is applicable, we set:

$$\begin{array}{lcl} F & := & \neg\,\epsilon_1 \wedge \neg\,\epsilon_2 \wedge \neg\,\epsilon_3 \\ \alpha & := & (\ \ (\textbf{if}\ \epsilon_1\ \textbf{then}\ \alpha_1) \\ & & \cup\ (\textbf{if}\ \epsilon_2\ \textbf{then}\ \alpha_2) \\ & & \cup\ (\textbf{if}\ \epsilon_3\ \textbf{then}\ \alpha_3)\ ). \end{array}$$

Otherwise, if the semantics is to indeterministically choose one of the applicable rules (cf. e.g. [6]), then we set:

$$\begin{array}{lcl} F & := & \neg\,\epsilon_1 \wedge \neg\,\epsilon_2 \wedge \neg\,\epsilon_3 \\ \alpha & := & \textbf{if}\quad\ \epsilon_1 \wedge \epsilon_2 \wedge \epsilon_3\ \textbf{then}\ (\alpha_1 \cup \alpha_2 \cup \alpha_3) \\ & & \textbf{else if}\ \epsilon_1 \wedge \epsilon_2\quad\ \textbf{then}\ (\alpha_1 \cup \alpha_2) \\ & & \textbf{else if}\ \epsilon_1 \wedge \epsilon_3\quad\ \textbf{then}\ (\alpha_1 \cup \alpha_3) \\ & & \textbf{else if}\ \epsilon_2 \wedge \epsilon_3\quad\ \textbf{then}\ (\alpha_2 \cup \alpha_3) \\ & & \textbf{else if}\ \epsilon_1\quad\ \textbf{then}\ \alpha_1 \\ & & \textbf{else if}\ \epsilon_2\quad\ \textbf{then}\ \alpha_2 \\ & & \textbf{else}\qquad\qquad\ \alpha_3. \end{array}$$

The expressiveness of EDL is best demonstrated by formalizing some statements about evolving algebras. Let $EA = (SIG, X, I, F, \alpha)$ and $EA' = (SIG, X, I, F', \alpha')$.

- EA has a terminating run:

$$I \rightarrow \langle \textbf{while}\ \neg\, F\ \textbf{do}\ \alpha \rangle \textbf{true}$$

- every terminating run of EA stops in a situation where φ holds:

$$I \rightarrow [\textbf{while}\ \neg\, F\ \textbf{do}\ \alpha]\varphi$$

- if there is some terminating run stopping in a situation where φ holds, then every terminating run stops in a situation where φ holds (global determinism):

$$I \rightarrow \Big(\langle \textbf{while}\ \neg\, F\ \textbf{do}\ \alpha \rangle \varphi \rightarrow [\textbf{while}\ \neg\, F\ \textbf{do}\ \alpha]\varphi\Big)$$

- in any reachable state of EA the formula φ holds, i.e. φ is an invariance property:

$$I \rightarrow \forall \kappa.[\textbf{loop if}\ \neg\, F\ \textbf{then}\ \alpha\ \textbf{times}\ \kappa]\varphi$$

The following two examples are special cases of the one above, i.e. φ is more specialized.

- every reachable state interprets the function symbol $f \in F_{s_1 \dots s_n, s}$ as a strict function:[21]

$$I \to \forall \kappa.[\textbf{loop if } \neg F \textbf{ then } \alpha \textbf{ times } \kappa] \\ \forall x_1 \dots \forall x_n.\Big(x_1 = \bot_{s_1} \vee \dots \vee x_n = \bot_{s_n} \to f(x_1, \dots, x_n) = \bot_s\Big)$$

- for any reachable state it holds that if φ is true in some next state, then φ is true in all possible next states (local determinism):

$$I \to \forall \kappa.[\textbf{loop if } \neg F \textbf{ then } \alpha \textbf{ times } \kappa] \\ \Big(\langle \textbf{if } \neg F \textbf{ then } \alpha \rangle \varphi \to [\textbf{if } \neg F \textbf{ then } \alpha] \varphi\Big)$$

- whenever there is a terminating run of EA stopping in a situation where φ holds, then there is also a terminating run of EA' stopping in a situation where ψ holds:

$$I \to \Big(\langle \textbf{while } \neg F \textbf{ do } \alpha \rangle \varphi \to \langle \textbf{while } \neg F' \textbf{ do } \alpha' \rangle \psi\Big).$$

5 Towards a Calculus for EDL

In this section first steps towards a sequent calculus for EDL are made. It turns out that the rules for not-extended dynamic logic remain valid. We propose rules for reasoning about the additional constructs (update of functions, extension of universes and simultaneous execution). These rules are based on the idea of symbolic execution.

Definition 5.1 (sequents)
Let $SIG = (S, F)$ and X a system of variables for SIG. The set of **extended dynamic logic sequents** over SIG and X is defined by

$$SEQ(SIG, X) \;:=\; \Big\{ \; , \; \Rightarrow \Delta \;\Big|\; , \; , \Delta \in EDL(SIG, X)^* \Big\}.$$

For a sequent $, \Rightarrow \Delta$ we call $,$ its **antecedent** and Δ its **succedent**.

Definition 5.2 (semantics of sequents)
Let $SIG = (S, F)$ and X a system of variables for SIG. A sequent $seq = ((\varphi_1, \dots, \varphi_n) \Rightarrow (\psi_1, \dots, \psi_n)) \in SEQ(SIG, X)$ is said to be **true**, written $\models seq$, if[22] $st \models (\varphi_1 \wedge \dots \wedge \varphi_n) \to (\psi_1 \vee \dots \vee \psi_n)$ for all $st \in STATE(SIG, X)$.

[21] Here we assume the signature SIG to provide constant symbols $\bot_s$ naming the error elements $UNDEF_s$.

[22] As usual, the empty conjunction is defined to be **true**, the empty disjunction is defined to be **false**.

Definition 5.3 (inference rules, theorems)
Let $SIG = (S, F)$ and X a system of variables for SIG. An **(inference) rule** r over SIG and X is a tuple $r = (pr_1 \cdots pr_n, concl)$ with $pr_1, \ldots, pr_n, concl \in SEQ(SIG, X)$. The sequents pr_i are called the **premises** of r and $concl$ is called the **conclusion** of r. We use the following notation for rules:

$$\frac{pr_1 \quad \cdots \quad pr_n}{concl} \text{ (r)}$$

A rule $r = (pr_1 \cdots pr_n, concl)$ over SIG and X is called **correct** if whenever $\models pr_i$ for all $i \in \{1, \ldots, n\}$ then also $\models concl$.

Let R be a set of rules over SIG and X. A sequent $seq \in SEQ(SIG, X)$ is called **theorem** of R, written $\vdash_R seq$, if it is member of the least set $THM_R \subseteq SEQ(SIG, X)$ satisfying

- for $(pr_1 \cdots pr_n, concl) \in R$ and $pr_1, \ldots, pr_n \in THM_R$ is $concl \in THM_R$.

R is called **correct** if $\vdash_R seq$ implies $\models seq$ for all $seq \in SEQ(SIG, X)$.

Fact 5.4 *A set of rules is correct if all its rules are correct.*

Thus we can concentrate on the correctness of single inference rules. It turns out that (most of) the rules known from (not extended) dynamic logic are correct for EDL too. Appendix A gives a proposal for a sequent calculus, which is based on the idea of symbolic execution. Propositional rules, equality rules, quantifier rules, and rules for basic program constructs are adopted from (not extended) dynamic logic, and their correctness is quite obvious. So, it remains to introduce inference rules for the additional program constructs. From theorem 3.10 we can derive the following rules:[23]

$$\frac{[(\beta, \alpha)]\varphi, , \ \Rightarrow \Delta}{[(\alpha, \beta)]\varphi, , \ \Rightarrow \Delta} \text{ (sim\_com\_l)} \qquad \frac{, \ \Rightarrow [(\beta, \alpha)]\varphi, \Delta}{, \ \Rightarrow [(\alpha, \beta)]\varphi, \Delta} \text{ (sim\_com\_r)}$$

$$\frac{[(\alpha_1, (\alpha_2, \alpha_3))]\varphi, , \ \Rightarrow \Delta}{[((\alpha_1, \alpha_2), \alpha_3)]\varphi, , \ \Rightarrow \Delta} \text{ (sim\_ass\_l)} \qquad \frac{, \ \Rightarrow [(\alpha_1, (\alpha_2, \alpha_3))]\varphi, \Delta}{, \ \Rightarrow [((\alpha_1, \alpha_2), \alpha_3)]\varphi, \Delta} \text{ (sim\_ass\_r)}$$

$$\frac{[\alpha]\varphi, , \ \Rightarrow \Delta}{[(\mathbf{skip}, \alpha)]\varphi, , \ \Rightarrow \Delta} \text{ (sim\_skip\_l)} \qquad \frac{, \ \Rightarrow [\alpha]\varphi, \Delta}{, \ \Rightarrow [(\mathbf{skip}, \alpha)]\varphi, \Delta} \text{ (sim\_skip\_r)}$$

$$\frac{[((\alpha_1, \beta) \cup (\alpha_2, \beta))]\varphi, , \ \Rightarrow \Delta}{[((\alpha_1 \cup \alpha_2), \beta)]\varphi, , \ \Rightarrow \Delta} \text{ (sim\_choice\_l)}$$

[23] The schematic notation of rules is described at the beginning of appendix A.

$$\frac{,\ \Rightarrow [((\alpha_1,\beta)\cup(\alpha_2,\beta))]\varphi,\Delta}{,\ \Rightarrow [((\alpha_1\cup\alpha_2),\beta)]\varphi,\Delta}\ (\text{sim\_choice\_r})$$

$$\frac{[\textbf{if } \epsilon \textbf{ then } (\alpha_1,\beta) \textbf{ else } (\alpha_2,\beta)]\varphi,,\ \Rightarrow \Delta}{[(\textbf{if } \epsilon \textbf{ then } \alpha_1 \textbf{ else } \alpha_2,\beta)]\varphi,,\ \Rightarrow \Delta}\ (\text{sim\_if\_l})$$

$$\frac{,\ \Rightarrow [\textbf{if } \epsilon \textbf{ then } (\alpha_1,\beta) \textbf{ else } (\alpha_2,\beta)]\varphi,\Delta}{,\ \Rightarrow [(\textbf{if } \epsilon \textbf{ then } \alpha_1 \textbf{ else } \alpha_2,\beta)]\varphi,\Delta}\ (\text{sim\_if\_r})$$

$$\frac{[\textbf{extend } s \textbf{ by } x' \textbf{ with } (\alpha_x^{x'},\beta)]\varphi,,\ \Rightarrow \Delta}{[(\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha,\beta)]\varphi,,\ \Rightarrow \Delta}\ (\text{sim\_extend\_l})$$

$$\frac{,\ \Rightarrow [\textbf{extend } s \textbf{ by } x' \textbf{ with } (\alpha_x^{x'},\beta)]\varphi,\Delta}{,\ \Rightarrow [(\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha,\beta)]\varphi,\Delta}\ (\text{sim\_extend\_r})$$

where $x' \in X_s$ is a variable not occurring in α or β.

We do not give rules for programs where compositions or (while) loops are involved in simultaneous execution for two reasons: firstly, such situation does not appear in evolving algebras (cf. definition 4.1)[24], and secondly such rules would be quite nasty (including renaming of a lot of signature symbols and subtle formulas for capturing clashes of function updates and domain extensions).

For the rules dealing with simultaneous execution of assignments and function updates some preparatory is needed. Firstly, a notational one: we write simply $(f_1\underline{t}_1 := t_1, \ldots, f_n\underline{t}_n := t_n)$ to denote a simultaneous execution of assignments and function updates. That is (due to theorem 3.10(b)) we omit the brackets and we do not differ (syntactically) between assignments and function updates, i.e. the f_i in $f_i\underline{t}_i := t_i$ may be variable symbols, which is possible by using $x\lambda := t$ (where λ is the empty term tuple) as a notation for an assignment $x := t$. Thus, syntactically variables are treated just as constants (0-ary function symbols), which enables a compact and convenient notation.

Secondly, we define a formula[25]

$$affects\big(f\underline{t} := t, g\underline{x}\big) \ :\equiv\ \begin{cases} \underline{x} = \underline{t} & \text{, if } f \equiv g \\ \textbf{false} & \text{, otherwise} \end{cases}$$

[24] However, allowing constructs for sequential execution in evolving algebra rules is sometimes convenient, as e.g. in [6].

[25] We use $(t_1,\ldots,t_n) = (t'_1,\ldots,t'_n)$ as an abbreviation for the conjunction of equations $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n$.

which is true if the function update (or assignment) $f\underline{t} := t$ *affects* the value of the term $g\underline{x}$, and a formula

$$clash\Big(f_1\underline{t}_1 := t_1, f_2\underline{t}_2 := t_2\Big) \;:\equiv\; \begin{cases} \underline{t}_1 = \underline{t}_2 \wedge t_1 \neq t_2 & \text{, if } f_1 \equiv f_2 \\ \textbf{false} & \text{, otherwise} \end{cases}$$

which is true if the function updates (or assignments) $f_1\underline{t}_1 := t_1$ and $f_2\underline{t}_2 := t_2$ clash.

Our proposal for a rule for symbolic execution of simultaneous function updates and assignments is as follows:[26]

$$\frac{\begin{array}{l} f'_1\underline{t}_1 = t_1, \ldots, f'_n\underline{t}_n = t_n, \\ \quad \bigwedge_{1\leq i\leq n} \forall \underline{x}_i.\Big(\bigwedge_{1\leq j\leq n} \neg\, affects\Big(f_j\underline{t}_j := t_j, f_i\underline{x}_i\Big) \\ \qquad\qquad \rightarrow f'_i\underline{x}_i = f_i\underline{x}_i\Big), \\ \quad , \;\Rightarrow \varphi^{f'_1,\ldots,f'_n}_{f_1,\ldots,f_n}, \Delta \end{array}}{, \;\Rightarrow [(f_1\underline{t}_1 := t_1, \ldots, f_n\underline{t}_n := t_n)]\varphi, \Delta} \text{ (sim\_update\_r)}$$

where the $f'_1, \ldots, f'_n$ are new[27] function symbols (or new variables in the case of assignments) with the same sorting as $f_1, \ldots, f_n$, and with $f'_i \equiv f'_j$ iff $f_i \equiv f_j$. For $f_i \in F_{\underline{s},s}$, $\underline{x}_i \in X_{\underline{s}}$ is a vector of distinct new variables.[28]

This rule needs some explanation. The new function symbols (or variables) f'_i play the part of the functions resulting from updating the functions f_i. With this in mind the premise of the rule (sim_update_r) can be read as:

If the new functions behave as imposed by the updates

$$f'_1\underline{t}_1 = t_1, \ldots, f'_n\underline{t}_n = t_n,$$

and behave on all locations not affected by any update just as the original functions

$$\bigwedge_{1\leq i\leq n} \forall \underline{x}_i.\Big(\bigwedge_{1\leq j\leq n} \neg\, affects\Big(f_j\underline{t}_j := t_j, f_i\underline{x}_i\Big) \rightarrow f'_i\underline{x}_i = f_i\underline{x}_i\Big),$$

and if all other assumptions hold

,

then follows that

$\Rightarrow$

[26] $\varphi^{f'_1,\ldots,f'_n}_{f_1,\ldots,f_n}$ is the formula resulting from syntactically replacing the function symbols (or variables) $f_1, \ldots, f_n$ by the function symbols (or variables) $f'_1, \ldots, f'_n$ in φ.

[27] Here and in all what follows, *new* means "not occurring in the conclusion of the rule".

[28] Given $(x_1, \ldots, x_n) \in X_{\underline{s}}$ then $\forall(x_1, \ldots, x_n).\varphi$ is an abbreviation for $\forall x_1.(\ldots \forall x_n.\varphi)$. For the empty tuple of variables λ we define $\forall\lambda.\varphi :\equiv \textbf{true}$.

the formula φ where the updated functions are used holds

$$\varphi^{f_1',\ldots,f_n'}_{f_1,\ldots,f_n},$$

or one of the other conjectures holds

$$\Delta.$$

Notice that

$$f_1'\underline{t}_1 = t_1, \ldots, f_n'\underline{t}_n = t_n \quad \rightarrow \quad \bigwedge_{1\leq i\leq n}\ \bigwedge_{i<j\leq n} \neg\, clash\Big(f_i\underline{t}_i := t_i, f_j\underline{t}_j := t_j\Big).$$

There is no need for a premise dealing with the case a clash occurs: remember that a clash is defined to behave like non-termination, i.e. $[(f_1\underline{t}_1 := t_1, \ldots, f_n\underline{t}_n := t_n)]\varphi$ is true in this case.

The following rule for symbolic execution of a simultaneous update in the antecedent is very similar to the above.

$$\frac{\begin{array}{l} f_1'\underline{t}_1 = t_1, \ldots, f_n'\underline{t}_n = t_n, \\ \quad \bigwedge_{1\leq i\leq n} \forall \underline{x}_i.\Big(\bigwedge_{1\leq j\leq n} \neg\, affects\Big(f_j\underline{t}_j := t_j, f_i\underline{x}_i\Big) \\ \qquad\qquad \rightarrow f_i'\underline{x}_i = f_i\underline{x}_i\Big), \\ \quad \varphi^{f_1',\ldots,f_n'}_{f_1,\ldots,f_n}, , \ \Rightarrow \Delta \end{array}}{\begin{array}{l} \bigvee_{1\leq i\leq n}\ \bigvee_{i<j\leq n} clash\Big(f_i\underline{t}_i := t_i, f_j\underline{t}_j := t_j\Big), , \ \Rightarrow \Delta \end{array}}\ (\text{sim\_update\_l})$$

The first premise differs from the one in (sim_update_r) only in the position of the occurrence of $\varphi^{f_1',\ldots,f_n'}_{f_1,\ldots,f_n}$. The second premise deals with the case a clash occurs. Notice that in the case of a single assignment the rules (sim_update_l, sim_update_r) degenerate to the ordinary assignment rules (assign_l, assign_r) (cf. appendix A).

Finally, here is our proposal for rules for the **extend** construct.

$$\frac{new(x', freevars, fcts), [\alpha^{x'}_x]\varphi, , \ ^{|x'} \Rightarrow \Delta^{|x'}}{[\textbf{extend}\ s\ \textbf{by}\ x\ \textbf{with}\ \alpha]\varphi, , \ \Rightarrow \Delta}\ (\text{extend\_l})$$

$$\frac{new(x', freevars, fcts), , \ ^{|x'} \Rightarrow [\alpha^{x'}_x]\varphi, \Delta^{|x'}}{, \ \Rightarrow [\textbf{extend}\ s\ \textbf{by}\ x\ \textbf{with}\ \alpha]\varphi, \Delta}\ (\text{extend\_r})$$

where $x' \in X_s$ is a new variable of sort s, $freevars$ denotes the set of all variables occurring free in the conclusion and $fcts$ denotes the set of all function symbols occurring in the conclusion. $, \ ^{|x'}, \Delta^{|x'}$ are constructed from

, , Δ by recursively replacing all quantified subformulas $\forall y.\psi$ with $y \in X_s$ by $\forall y.(y = x' \vee \psi^{|x'})$. Given any system of finite sets of variables $\tilde{X} \subseteq X$ and any finite set of function symbols $\tilde{F} \subseteq F$, then the formula

$$new(x', \tilde{X}, \tilde{F}) \quad :\equiv \quad \bigwedge_{y \in \tilde{X}_s} \big(y \neq x'\big) \wedge \bigwedge_{\underline{s} \in S^*} \bigwedge_{f \in \tilde{F}_{\underline{s},s}} \forall \underline{y}.\big(f\underline{y} \neq x'\big)$$

is true in a state if the value of x' differs from the values of all variables in $\tilde{X}_s$ and from all values of functions in $\cup_{\underline{s} \in S^*} \tilde{F}_{\underline{s},s}$. The correctness of (extend_l) and (extend_r) is not obvious. So this necessitates a proof. We start with the following lemma.

Lemma 5.5 (extension)
Let $SIG = (S, F)$ a signature, X a system of variables for SIG, $s \in S$, $\alpha \in EPROG(SIG, X)$, $\varphi \in EDL(SIG, X)$, and $x, x' \in X_s$ with $x' \not\equiv x$ and x' does not occur in α or φ. Furthermore let $st = (\mathcal{A}, v) \in STATE(SIG, X)$.

(a) *there is an $a \notin A_s$ with $(st +_s \{a\})[x \leftarrow a] \models \varphi$ iff $(st +_s \{a\})[x \leftarrow a] \models \varphi$ for all $a \notin A_s$*

(b) $(st +_s \{a\})[x' \leftarrow a] \models new(x', \tilde{X}, \tilde{F})$ *for all $a \notin A_s$ and finite $\tilde{X} \subseteq X$, $\tilde{F} \subseteq F$*

(c) $st \models \varphi$ *iff for all $a \notin A_s$ is $(st +_s \{a\})[x' \leftarrow a] \models \varphi^{|x'}$ where $\varphi^{|x'}$ is constructed from φ by recursively replacing all quantified subformulas $\forall y.\psi$ with $y \in X_s$ by $\forall y.(y = x' \vee \psi^{|x'})$*

(d) $st \models [$**extend** s **by** x **with** $\alpha]\varphi$ *iff* $(st +_s \{a\})[x' \leftarrow a] \models [\alpha^{x'}_x]\varphi$ *for all $a \notin A_s$.*

Proof. (a) holds because for $a, b \notin A_s$ the states $(st +_s \{a\})[x \leftarrow a]$ and $(st +_s \{b\})[x \leftarrow b]$ are isomorphic. (b) is obvious. (c) can be shown by structural induction on the formula φ. The proof of (d) is a little bit tricky and needs more effort. It is given in appendix B.4. ■

Theorem 5.6 (correctness of extend rules)
The rules (extend_l) *and* (extend_r) *are correct.*

Proof. Let $SIG = (S, F)$ a signature, X a system of variables for SIG. To prove correctness of rule (extend_l) we have to show that for all $s \in S$, $\alpha \in EPROG(SIG, X)$, $\varphi \in EDL(SIG, X)$, and $x, x' \in X_s$ with $x' \not\equiv x$ and x' does not occur in α or φ it holds that

$$\models \quad new(x', freevars, fcts), [\alpha^{x'}_x]\varphi, , \, ^{|x'} \Rightarrow \Delta^{|x'} \qquad \text{(A)}$$

implies

$$\models \quad [\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha]\varphi, , \Rightarrow \Delta. \qquad \text{(B)}$$

Assuming (A) we get that for all $st = (\mathcal{A}, v) \in STATE(SIG, X)$ and all $a \notin A_s$ holds

$$(st +_s \{a\})[x' \leftarrow a] \quad \models \quad new(x', freevars, fcts), [\alpha_x^{x'}]\varphi, , {}^{|x'} \Rightarrow \Delta^{|x'}.$$

Together with lemma 5.5(a) follows that for all st at least one of the following cases happens:[29]

- $(st +_s \{a\})[x' \leftarrow a] \not\models new(x', freevars, fcts)$ for all $a \notin A_s$
- $(st +_s \{a\})[x' \leftarrow a] \not\models [\alpha_x^{x'}]\varphi$ for all $a \notin A_s$
- $(st +_s \{a\})[x' \leftarrow a] \not\models , {}^{|x'}$ for all $a \notin A_s$
- $(st +_s \{a\})[x' \leftarrow a] \models \Delta^{|x'}$ for all $a \notin A_s$.

By applying lemma 5.5(b)–(d) one gets that for all st at least one of the following cases happens:

- $st \not\models$ [**extend** s **by** x **with** $\alpha]\varphi$
- $st \not\models ,$
- $st \models \Delta$

which means that

$$st \quad \models \quad [\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha]\varphi, , \Rightarrow \Delta$$

for all states st, i.e. (B) holds.

The correctness proof for (extend_r) works very similar. ■

We conclude this section with some remarks.

Remark (completeness)
The rules presented here are not sufficient for a complete calculus. There are several reasons for this incompleteness:

- Just as in ordinary dynamic logic there is no (effective) complete axiomatization for the **while** construct.
- We have not given any rules for symbolic execution of programs where compositions or (while) loops are involved in simultaneous execution.

[29] For simplicity but without loss of generality we assume , and Δ to be single formulas (instead of tuples of formulas).

- The rules dealing with domain extension (extend_l, extend_r) are not equivalence preserving. Presumably, this can be achieved by adding formulas $\forall x_1, \ldots, x_n . f(x_1, \ldots, x', \ldots, x_n) = UNDEF_s$ to the antecedent of the premise. These formulas impose the value of functions at the new created element x' to be undefined.[30]

However, we believe (and hope to demonstrate it in the foreseeable future) that these kinds of incompleteness do not seriously restrict reasoning about evolving algebras in practice.

Remark (economy of rules)
Possibly, the proposed rules lead to non-economic proofs. Especially, introducing new function symbols or variables (cf. sim_update_l, sim_update_r) and doubling of parts of the program (cf. sim_choice_l, sim_choice_r, sim_if_l, sim_if_r) (which might cause doubling of parts of proofs), should be avoided whenever possible. In the next future we will work on such improvements of the calculus.

Remark (separating static and dynamic part)
The function symbols from the signature of an evolving algebra can be distinguished into so-called *dynamic* functions, which might be updated during a run of the evolving algebra, and so-called *static* functions, for which no update exists in the rules of the evolving algebra. (The same distinction is possible for variables.) In practice it is reasonable to make this separation explicit in the signature, a technique which is used e.g. in [19]. Besides methodological merits one also gains advantages in (interactive) reasoning about evolving algebras: All the information (axioms and derived lemmata) about the static part can be kept globally, so that the current (sub-)goals in a proof have only to keep information concerning the dynamic part. So the goals become more readable and more tractable.

6 Conclusion and Related Work

We have defined syntax and semantics of EDL, an extension of dynamic logic by update of functions, extension of universes and simultaneous execution. This extension allows to directly represent (statements about) evolving algebras. We have indicated that a calculus for EDL can be obtained from a sequent calculus for (not extended) dynamic logic only by adding further rules, but without modifications of original rules. This gives us reason to hope that the KIV system, which supports interactive, evolutionary construction

[30] We have omitted these formulas, since we believe that in most practical reasoning (about software) there is no need for inference steps like: "if a certain value is undefined, then ...".

of (complicated) dynamic logic proofs, can be turned into a powerful tool for reasoning about evolving algebras only by extending it, i.e. without (substantially) modifying existing code. This tool will be useful for proving certain properties of single evolving algebras or relations between (two or more) evolving algebras, and even for proving relations between PASCAL-like programs and evolving algebras, and relations between algebraic first-order specifications and evolving algebras.

The work most close to ours, and with the same aim — which is to make first steps towards (a system for) formal reasoning about evolving algebras (and related specification formalisms, e.g. COLD [9]) — was done by Groenboom and Renardel de Lavalette [11, 12]. In [11] Groenboom and Renardel de Lavalette present MLCM (Modal Logic of Creation and Modification), which is (as EDL) a derivation from traditional dynamic logic. On the basis of MLCM they developed a Formal Language for Evolving Algebras (FLEA$^{\square}$) [12]. Though closely related there are substantial differences to our approach. The semantics of FLEA$^{\square}$ is defined using a so-called super-universum (cf. [14]) and special pre-defined functions (e.g. `Reserve`). Furthermore a repetition construct is available only in MLCM but not in FLEA$^{\square}$. A minor difference is that in the semantics of the **extend** construct presented here the variable is bound locally and not globally. The axiomatization for MLCM and FLEA$^{\square}$ considerably differs from the sequent calculus proposed in this paper (which is based on the idea of symbolic execution). While the aim of Groenboom and Renardel de Lavalette is to get an axiomatization which is as complete as possible, we have presented a calculus designed for practical use (in the KIV system).

Another proposal for reasoning about evolving algebras, restricted to invariance properties of single evolving algebras, can be found in Poetzsch-Heffter's work on deriving partial correctness logics from evolving algebras [20]. In order to prove that an evolving algebra has an invariance property INV, one has to show that INV holds in the initial state, in formulas

$$START \rightarrow INV,$$

(the formula $START$ describes the relevant properties of the initial state) and that INV is invariant during computation, in formulas

$$INV \rightarrow \mathbf{wb}[INV].$$

Here **wb** is a weakest backward transformer in the style of the weakest precondition transformer of Hoare logic, i.e. the (first-order) formula $\mathbf{wb}[INV]$ expresses that INV holds in all states that can be reached by firing one of the rules of the evolving algebra.

Another work, which has strongly stimulated our interest in the topic, is that of Schellhorn and Ahrendt [22]. Their aim is to reconstruct the WAM

compiler correctness proofs outlined in [6] in a rigorous way supported by deduction systems. In doing so, the KIV system has already been successfully applied for reasoning about (a representation of a certain class of) evolving algebras. For this purpose the KIV system was not adapted in any way, but evolving algebras are simulated by formalizing dynamic functions as association lists, i.e. as explicit data. (A similar technique is used in [20]).

Acknowledgments

I am indebted to Thomas Fuchß, Gerhard Schellhorn, and Wolfgang Ahrendt for many valuable discussions on the topic of this paper. Thomas Fuchß, Martin Giese, Elmar Habermalz and Roland Preiß have made helpful comments on an earlier draft.

References

[1] J.A. Bergstra and J.V. Tucker. Algebraic specifications of computable and semicomputable data types. *Theoret. Comput. Sci.*, 50:137–181, 1987.

[2] E. Börger. A logical operational semantics for full Prolog. In E. Börger, H. Kleine-Büning, and M.M. Richter, editors, *Proceedings of the CSL'89*, volume 440 of *Lecture Notes in Computer Science*, pages 36–64. Springer Verlag, 1990.

[3] E. Börger, G. Del Castillo, P. Galvan, and D. Rosenzweig. Towards a mathematical specification of the APE100 architecture: The APESE model. In B. Pehrson and I. Simon, editors, *Proceedings of the IFIP 13th World Computer Congress, volume I: Technology/Foundations*, pages 396–401. Elsevier Science Publishers B. V., 1994.

[4] E. Börger, I. Durdanovic, and D. Rosenzweig. Occam: Specification and compiler correctness. In U. Montanari and E.-R. Olderog, editors, *Proceedings of the IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET 94)*. North Holland, 1994.

[5] E. Börger and U. Glässer. A formal specification of the PVM architecture. In B. Pehrson and I. Simon, editors, *Proceedings of the IFIP 13th World Computer Congress, volume I: Technology/Foundations*. Elsevier Science Publishers B. V., 1994.

[6] E. Börger and D. Rosenzweig. The WAM — definition and compiler correctness. Technical Report TR–14/92, Universita degli Studi di Pisa, Dipartimento di Informatika, 1992.

[7] E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 1994.

[8] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[9] L.M.G. Feijs and H.B.M. Jonkers. *Formal Specification and Design*, volume 35 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1994.

[10] R. Goldblatt. *Axiomatizing the Logic of Computer Programming*, volume 130 of *Lecture Notes in Computer Science*. Springer Verlag, 1982.

[11] R. Groenboom and G.R. Renardel de Lavalette. Reasoning about dynamic features in specification languages: A modal view on creation and modification. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the Workshop in Semantics of Specification Languages, Utrecht, 1993*. Springer Verlag, 1994.

[12] R. Groenboom and G.R. Renardel de Lavalette. A formalization of evolving algebras. In *Proceedings of Accolade 1995*, forthcoming.

[13] Y. Gurevich. Evolving algebras. A tutorial introduction. In *Bulletin of the European Association for Theoretical Computer Science*, volume 43, pages 264–284, 1991.

[14] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.

[15] Y. Gurevich and J. Huggins. The semantics of the C programming language. In *Proceedings of the CSL*, volume 702 of *Lecture Notes in Computer Science*, pages 273–309. Springer Verlag, 1993.

[16] Y. Gurevich and L. Moss. Algebraic operational semantics and Occam. In E. Börger, H. Kleine-Büning, and M.M. Richter, editors, *Proceedings of the CSL'89*, volume 440 of *Lecture Notes in Computer Science*, pages 176–192. Springer Verlag, 1990.

[17] D. Harel. *First Order Dynamic Logic*. Lecture Notes in Computer Science. Springer Verlag, 1979.

[18] M. Heisel, W. Reif, and W. Stephan. A dynamic logic for program verification. In *Logical Foundations of Computer Science*, volume 363 of *Lecture Notes in Computer Science*, pages 134–145. Springer Verlag, 1989.

[19] A. Kappel. Algebraische operationale Semantik und ihre Anwendung auf Prolog. Master's thesis, Universität Dortmund, Lehrstuhl Informatik V, 1990.

[20] A. Poetzsch-Heffter. Deriving partial correctness logics from evolving algebras. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress 1994, Volume I: Technology/Foundations*. Elsevier Amsterdam, 1994.

[21] W. Reif. The KIV-system: Systematic construction of verified software. In *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[22] G. Schellhorn. `<unknown title>` (report on a case study with KIV which deals with proving WAM-compiler correctness), 1996 forthcoming.

[23] M. Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675–788. Elsevier Science Publishers B. V., 1990.

A Inference Rules for EDL

Notations

We assume a given signature $SIG = (S, F)$ with a system X of variables, and use the following meta-variables:

$t, t_1, t_2, t_1', t_2', \ldots$	for terms from $T_F(X)$
$x, x', x_1, x_2, \ldots$	for variables from X
$f, f_1, f_2, \ldots$	for function symbols from F
s	for sorts from S
φ, ψ	for formulas from $EDL(SIG, X)$
ϵ	for boolean expressions from $BXP(SIG, X)$
$, , \Delta$	for tuples of formulas from $EDL(SIG, X)$
$\alpha, \beta, \alpha_1, \alpha_2, \alpha_3$	for programs from $EPROG(SIG, X)$.

Thus each item below represents an infinite *set* of rules, with the range of meta-variables as just defined.

A relaxed notation for sequences is used. E.g. denotes

$$\varphi, , \ \Rightarrow \Delta, \psi$$

the sequence with antecedent $(\varphi, ,)$ which results from attaching the formula φ in front of the tuple $, ,$ and with succedent (Δ, ψ) which results from attaching the formula ψ at the end of the tuple Δ.

Propositional Rules

$$\frac{}{\varphi, , \ \Rightarrow \varphi, \Delta}\ (\text{ax}) \qquad \frac{}{\mathbf{false}, , \ \Rightarrow \Delta}\ (\text{false\_l})$$

$$\frac{, \ \Rightarrow \varphi, \Delta \quad \psi, , \ \Rightarrow \Delta}{\varphi \rightarrow \psi, , \ \Rightarrow \Delta}\ (\text{imp\_l}) \qquad \frac{\varphi, , \ \Rightarrow \psi, \Delta}{, \ \Rightarrow \varphi \rightarrow \psi, \Delta}\ (\text{imp\_r})$$

$$\frac{, \ \Rightarrow \varphi, \Delta \quad \varphi, , \ \Rightarrow \Delta}{, \ \Rightarrow \Delta}\ (\text{cut})$$

Structure Rules

$$\frac{, \ \Rightarrow \Delta}{\varphi, , \ \Rightarrow \Delta}\ (\text{weakening\_l}) \qquad \frac{, \ \Rightarrow \Delta}{, \ \Rightarrow \varphi, \Delta}\ (\text{weakening\_r})$$

$$\frac{, , \varphi \Rightarrow \Delta}{\varphi, , \ \Rightarrow \Delta}\ (\text{rotate\_l}) \qquad \frac{, \ \Rightarrow \Delta, \varphi}{, \ \Rightarrow \varphi, \Delta}\ (\text{rotate\_r})$$

Equality Rules[31]

$$\frac{}{\Rightarrow t = t}\ \text{(reflexivity)}$$

$$\frac{}{t_1 = t_2 \Rightarrow t_2 = t_1}\ \text{(symmetry)}$$

$$\frac{}{t_1 = t_2, t_2 = t_3 \Rightarrow t_1 = t_3}\ \text{(transitivity)}$$

$$\frac{}{t_1 = t_1', \ldots, t_n = t_n' \Rightarrow f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')}\ \text{(congruence)}$$

Counter Rules

$$\frac{}{czero = csucc(\kappa) \Rightarrow}\ \text{(minimal\_element)}$$

$$\frac{}{csucc(\kappa_1) = csucc(\kappa_2) \Rightarrow \kappa_1 = \kappa_2}\ \text{(injectivity\_of\_csucc)}$$

$$\frac{}{\varphi(czero), \forall \kappa'.\Big(\varphi(\kappa') \rightarrow \varphi(csucc(\kappa'))\Big) \Rightarrow \forall \kappa.\varphi(\kappa)}\ \text{(induction)},^{32}\kappa' \text{ new}$$

Quantifier Rules

$$\frac{[\alpha]\varphi, \forall x.\varphi, , \ \Rightarrow \Delta}{\forall x.\varphi, , \ \Rightarrow \Delta}\ \text{(all\_l)},\ \alpha \in^{33} PROG(SIG, X)\ \text{with}^{34} asg(\alpha) \subseteq \{x\}$$

$$\frac{, \ \Rightarrow \varphi_x^{x'}, \Delta}{, \ \Rightarrow \forall x.\varphi, \Delta}\ \text{(all\_r)},\ x' \text{ new}$$

[31] In the KIV system equational reasoning is (mainly) done by the so called simplifier tactic, which works in the manner of a rewrite system.

[32] $\varphi(czero)$ is the formula resulting from substituting all free occurrences of κ by $czero$ in $\varphi(\kappa)$, etc.

[34] Notice that α is *not* any program from $EPROG(SIG, X)$, e.g. function updates and domain extensions are not allowed in it.

[34] $asg(\alpha)$ is the set of all variables occurring on the left-hand side of assignments in a program $\alpha \in PROG(SIG, X)$.

Rules for Basic Program Constructs

$$\frac{\varphi, , \ \Rightarrow \Delta}{[\mathbf{skip}]\varphi, , \ \Rightarrow \Delta}\ (\text{skip\_l}) \qquad \frac{, \ \Rightarrow \varphi, \Delta}{, \ \Rightarrow [\mathbf{skip}]\varphi, \Delta}\ (\text{skip\_r})$$

$$\frac{x' = t, \varphi_x^{x'}, , \ \Rightarrow \Delta}{[x := t]\varphi, , \ \Rightarrow \Delta}\ (\text{assign\_l}) \qquad \frac{x' = t, , \ \Rightarrow \varphi_x^{x'}, \Delta}{, \ \Rightarrow [x := t]\varphi, \Delta}\ (\text{assign\_r}),\ x' \text{ new}$$

$$\frac{[\alpha]\varphi, [\beta]\varphi, , \ \Rightarrow \Delta}{[(\alpha \cup \beta)]\varphi, , \ \Rightarrow \Delta}\ (\text{choice\_l}) \qquad \frac{, \ \Rightarrow [\alpha]\varphi, \Delta \quad , \ \Rightarrow [\beta]\varphi, \Delta}{, \ \Rightarrow [(\alpha \cup \beta)]\varphi, \Delta}\ (\text{choice\_r})$$

$$\frac{[\alpha][\beta]\varphi, , \ \Rightarrow \Delta}{[(\alpha; \beta)]\varphi, , \ \Rightarrow \Delta}\ (\text{comp\_l}) \qquad \frac{, \ \Rightarrow [\alpha][\beta]\varphi, \Delta}{, \ \Rightarrow [(\alpha; \beta)]\varphi, \Delta}\ (\text{comp\_r})$$

$$\frac{\epsilon, [\alpha]\varphi, , \ \Rightarrow \Delta \quad [\beta]\varphi, , \ \Rightarrow \epsilon, \Delta}{[\mathbf{if}\ \epsilon\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]\varphi, , \ \Rightarrow \Delta}\ (\text{cond\_l})$$

$$\frac{\epsilon, , \ \Rightarrow [\alpha]\varphi, \Delta \quad , \ \Rightarrow \epsilon, [\beta]\varphi, \Delta}{, \ \Rightarrow [\mathbf{if}\ \epsilon\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]\varphi, \Delta}\ (\text{cond\_r})$$

$$\frac{\forall\kappa.[\mathbf{loop\ if}\ \epsilon\ \mathbf{then}\ \alpha\ \mathbf{times}\ \kappa](\epsilon \vee \varphi), , \ \Rightarrow \Delta}{[\mathbf{while}\ \epsilon\ \mathbf{do}\ \alpha]\varphi, , \ \Rightarrow \Delta}\ (\text{while\_l}),\ \kappa \text{ new}$$

$$\frac{, \ \Rightarrow [\mathbf{loop\ if}\ \epsilon\ \mathbf{then}\ \alpha\ \mathbf{times}\ \kappa](\epsilon \vee \varphi), \Delta}{, \ \Rightarrow [\mathbf{while}\ \epsilon\ \mathbf{do}\ \alpha]\varphi, \Delta}\ (\text{while\_r}),\ \kappa \text{ new}$$

$$\frac{[\alpha]\forall\kappa.[\mathbf{loop}\ \alpha\ \mathbf{times}\ \kappa]\varphi, \varphi, , \ \Rightarrow \Delta}{\forall\kappa.[\mathbf{loop}\ \alpha\ \mathbf{times}\ \kappa]\varphi, , \ \Rightarrow \Delta}\ (\text{loop\_unwind})$$

$$\frac{\varphi, , \ \Rightarrow \Delta}{[\mathbf{loop}\ \alpha\ \mathbf{times}\ czero]\varphi, , \ \Rightarrow \Delta}\ (\text{czero\_loop\_l})$$

$$\frac{, \ \Rightarrow \varphi, \Delta}{, \ \Rightarrow [\mathbf{loop}\ \alpha\ \mathbf{times}\ czero]\varphi, \Delta}\ (\text{czero\_loop\_r})$$

$$\frac{[\alpha][\mathbf{loop}\ \alpha\ \mathbf{times}\ \kappa]\varphi, , \ \Rightarrow \Delta}{[\mathbf{loop}\ \alpha\ \mathbf{times}\ csucc(\kappa)]\varphi, , \ \Rightarrow \Delta}\ (\text{csucc\_loop\_l})$$

$$\frac{, \ \Rightarrow [\alpha][\mathbf{loop}\ \alpha\ \mathbf{times}\ \kappa]\varphi, \Delta}{, \ \Rightarrow [\mathbf{loop}\ \alpha\ \mathbf{times}\ csucc(\kappa)]\varphi, \Delta}\ (\text{csucc\_loop\_r})$$

Rules for Additional Program Constructs

$$\frac{\begin{array}{l} f'_1\underline{t}_1 = t_1, \ldots, f'_n\underline{t}_n = t_n, \\ \quad \bigwedge_{1\le i\le n} \forall \underline{x}_i.\Big(\bigwedge_{1\le j\le n} \neg\, affects\Big(f_j\underline{t}_j := t_j, f_i\underline{x}_i\Big) \\ \qquad\qquad \rightarrow f'_i\underline{x}_i = f_i\underline{x}_i\Big), \\ \quad \varphi^{f'_1,\ldots,f'_n}_{f_1,\ldots,f_n}, , \;\Rightarrow \Delta \end{array}}{\bigvee_{1\le i\le n} \bigvee_{i<j\le n} clash\Big(f_i\underline{t}_i := t_i, f_j\underline{t}_j := t_j\Big), , \;\Rightarrow \Delta} \text{ (sim\_update\_l)}$$

$$\frac{\begin{array}{l} f'_1\underline{t}_1 = t_1, \ldots, f'_n\underline{t}_n = t_n, \\ \quad \bigwedge_{1\le i\le n} \forall \underline{x}_i.\Big(\bigwedge_{1\le j\le n} \neg\, affects\Big(f_j\underline{t}_j := t_j, f_i\underline{x}_i\Big) \\ \qquad\qquad \rightarrow f'_i\underline{x}_i = f_i\underline{x}_i\Big), \\ \quad , \;\Rightarrow \varphi^{f'_1,\ldots,f'_n}_{f_1,\ldots,f_n}, \Delta \end{array}}{, \;\Rightarrow [(f_1\underline{t}_1 := t_1, \ldots, f_n\underline{t}_n := t_n)]\varphi, \Delta} \text{ (sim\_update\_r)}$$

where the $f'_1, \ldots, f'_n$ are new function symbols (or new variables), the $\underline{x}_i$ are tuples of new distinct variables, and

$$affects\Big(f_j\underline{t}_j := t_j, f_i\underline{x}_i\Big) \;:\equiv\; \begin{cases} \underline{x} = \underline{t}_j & \text{, if } f_j \equiv f_i \\ \textbf{false} & \text{, otherwise} \end{cases}$$

$$clash\Big(f_i\underline{t}_i := t_i, f_j\underline{t}_j := t_j\Big) \;:\equiv\; \begin{cases} \underline{t}_i = \underline{t}_j \wedge t_i \neq t_j & \text{, if } f_i \equiv f_j \\ \textbf{false} & \text{, otherwise} \end{cases}$$

$$\frac{[(\beta,\alpha)]\varphi, , \;\Rightarrow \Delta}{[(\alpha,\beta)]\varphi, , \;\Rightarrow \Delta} \text{ (sim\_com\_l)} \qquad \frac{, \;\Rightarrow [(\beta,\alpha)]\varphi, \Delta}{, \;\Rightarrow [(\alpha,\beta)]\varphi, \Delta} \text{ (sim\_com\_r)}$$

$$\frac{[(\alpha_1,(\alpha_2,\alpha_3))]\varphi, , \;\Rightarrow \Delta}{[((\alpha_1,\alpha_2),\alpha_3)]\varphi, , \;\Rightarrow \Delta} \text{ (sim\_ass\_l)} \qquad \frac{, \;\Rightarrow [(\alpha_1,(\alpha_2,\alpha_3))]\varphi, \Delta}{, \;\Rightarrow [((\alpha_1,\alpha_2),\alpha_3)]\varphi, \Delta} \text{ (sim\_ass\_r)}$$

$$\frac{[\alpha]\varphi, , \;\Rightarrow \Delta}{[(\textbf{skip},\alpha)]\varphi, , \;\Rightarrow \Delta} \text{ (sim\_skip\_l)} \qquad \frac{, \;\Rightarrow [\alpha]\varphi, \Delta}{, \;\Rightarrow [(\textbf{skip},\alpha)]\varphi, \Delta} \text{ (sim\_skip\_r)}$$

$$\frac{[((\alpha_1,\beta)\cup(\alpha_2,\beta))]\varphi, , \;\Rightarrow \Delta}{[((\alpha_1\cup\alpha_2),\beta)]\varphi, , \;\Rightarrow \Delta} \text{ (sim\_choice\_l)}$$

$$\frac{,\ \Rightarrow [((\alpha_1,\beta)\cup(\alpha_2,\beta))]\varphi,\Delta}{,\ \Rightarrow [((\alpha_1\cup\alpha_2),\beta)]\varphi,\Delta}\ (\text{sim\_choice\_r})$$

$$\frac{[\textbf{if } \epsilon \textbf{ then } (\alpha_1,\beta) \textbf{ else } (\alpha_2,\beta)]\varphi,,\ \Rightarrow \Delta}{[(\textbf{if } \epsilon \textbf{ then } \alpha_1 \textbf{ else } \alpha_2,\beta)]\varphi,,\ \Rightarrow \Delta}\ (\text{sim\_if\_l})$$

$$\frac{,\ \Rightarrow [\textbf{if } \epsilon \textbf{ then } (\alpha_1,\beta) \textbf{ else } (\alpha_2,\beta)]\varphi,\Delta}{,\ \Rightarrow [(\textbf{if } \epsilon \textbf{ then } \alpha_1 \textbf{ else } \alpha_2,\beta)]\varphi,\Delta}\ (\text{sim\_if\_r})$$

$$\frac{[\textbf{extend } s \textbf{ by } x' \textbf{ with } (\alpha_x^{x'},\beta)]\varphi,,\ \Rightarrow \Delta}{[(\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha,\beta)]\varphi,,\ \Rightarrow \Delta}\ (\text{sim\_extend\_l}),\ x' \text{ new}$$

$$\frac{,\ \Rightarrow [\textbf{extend } s \textbf{ by } x' \textbf{ with } (\alpha_x^{x'},\beta)]\varphi,\Delta}{,\ \Rightarrow [(\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha,\beta)]\varphi,\Delta}\ (\text{sim\_extend\_r}),\ x' \text{ new}$$

$$\frac{new(x',freevars,fcts),[\alpha_x^{x'}]\varphi,,\ ^{|x'}\Rightarrow \Delta^{|x'}}{[\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha]\varphi,,\ \Rightarrow \Delta}\ (\text{extend\_l})$$

$$\frac{new(x',freevars,fcts),,\ ^{|x'}\Rightarrow [\alpha_x^{x'}]\varphi,\Delta^{|x'}}{,\ \Rightarrow [\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha]\varphi,\Delta}\ (\text{extend\_r})$$

where $x' \in X_s$ is a new variable of sort s, $freevars$ denotes the set of all variables occurring free in the conclusion and $fcts$ denotes the set of all function symbols occurring in the conclusion. $,\ ^{|x'},\Delta^{|x'}$ are constructed from $,,\Delta$ by recursively replacing all quantified subformulas $\forall y.\psi$ with $y \in X_s$ by $\forall y.(y = x' \vee \psi^{|x'})$. For any system of finite sets of variables $\tilde{X} \subseteq X$ and any finite set of function symbols $\tilde{F} \subseteq F$ the formula $new(x',\tilde{X},\tilde{F})$ is defined by

$$new(x',\tilde{X},\tilde{F}) \ :\equiv\ \bigwedge_{y\in\tilde{X}_s}\big(y \neq x'\big) \wedge \bigwedge_{\underline{s}\in S^*}\bigwedge_{f\in\tilde{F}_{\underline{s},s}} \forall \underline{y}.\big(f\underline{y} \neq x'\big).$$

B Postponed Proofs

This section presents the more complicated and technical proofs. It starts with providing some facts or lemmata used later.

B.1 Useful Lemmata

Fact B.1 (program semantics and state operations)
Let $SIG = (S, F)$ a signature, X a system of variables for SIG, $s \in S$, $x \in X_s$, $\alpha \in EPROG(SIG, X)$, and $st = (\mathcal{A}, v)$ and $st' = (\mathcal{A}', v')$ states from $STATE(SIG, X)$. Then it holds:

(a) *If $st \,[\![\alpha]\!]\, st'$, $a \in A_s$, and x does not occur in α then $(st[x \leftarrow a]) \,[\![\alpha]\!]\, (st'[x \leftarrow a])$.*

(b) *If $st \,[\![\alpha]\!]\, st'$ and $a \notin A_s \cup A'_s$ then[35] $(st +_s \{a\}) \,[\![\alpha]\!]\, (st' +_s \{a\})$.*

(c) *If $(st[x \leftarrow a]) \,[\![\alpha]\!]\, st'$ and $x' \in X_s$ does not occur in α then $(st[x' \leftarrow a]) \,[\![\alpha^{x'}_{x}]\!]\, (st'[x \leftarrow st(x)][x' \leftarrow st'(x)])$.*

(d) *If $(st[x \leftarrow a]) \,[\![\alpha]\!]\, st'$ and x does not occur in α then $st \,[\![\alpha]\!]\, st'([x \leftarrow st(x)])$.*

(e) *If $(st +_s \{a\}) \,[\![\alpha]\!]\, st'$ then there is some state $st'' \in STATE(SIG, X)$ such that $st \,[\![\alpha]\!]\, st''$ and $st' = st'' +_s \{a\}$ for some $a \in A'_s \setminus A''_s$.*

Proof. All proofs work by structural induction on programs α. ■

Lemma B.2 (join and other state operations)
Let $SIG = (S, F)$ a signature, X a system of variables for SIG, $s \in S$, $x \in X_s$, and st, st', and st'' states from $STATE(SIG, X)$.

(a) *If $st''(x) = st(x)$, $a \in A_s \cap A''_s$, and $st \rightsquigarrow st'$ and $st \rightsquigarrow st''$ are consistent then are $st[x \leftarrow a] \rightsquigarrow st'$ and $st[x \leftarrow a] \rightsquigarrow st''[x \leftarrow a]$ consistent with $st' \oplus_{st[x \leftarrow a]} st''[x \leftarrow a] = st' \oplus_{st} st''$.*

(b) *If $a \in A'_s \setminus A''_s$ and the state changes $st \rightsquigarrow st'$ and $st \rightsquigarrow st''$ are consistent then are $(st +_s \{a\}) \rightsquigarrow st'$ and $(st +_s \{a\}) \rightsquigarrow (st'' +_s \{a\})$ consistent with $st' \oplus_{(st +_s \{a\})} (st'' +_s \{a\}) = st' \oplus_{st} st''$.*

(c) *If $st''(x) = st(x)$, $a \in A'_s$, and the state changes $st \rightsquigarrow st'$ and $st \rightsquigarrow st''$ are consistent then are $st \rightsquigarrow st'[x \leftarrow a]$ and $st \rightsquigarrow st''$ consistent with $(st'[x \leftarrow a]) \oplus_{st} st' = (st' \oplus_{st} st'')[x \leftarrow a]$.*

[35] This does not hold if random assignments — which are fortunately not available in EDL — occur in α!

(d) *If* $a \in A'_s \setminus (A''_s \cup A_s)$ *and the state changes* $(st +_s \{a\}) \rightsquigarrow st'$ *and* $(st +_s \{a\}) \rightsquigarrow (st'' +_s \{a\})$ *are consistent then are* $st \rightsquigarrow st'$ *and* $st \rightsquigarrow st''$ *consistent with* $st' \oplus_{st} st'' = st' \oplus_{(st +_s \{a\})} (st'' +_s \{a\})$.

Proof. Essentially, the proofs work by simply unfolding the definitions. ■

B.2 Proof for Fact 3.8

Proof. **(a)** follows obviously from (the symmetry in) definitions 3.6 and 3.7. The proof of **(b)** needs more effort. We start by unfolding the definition of consistency.

- state changes $st \rightsquigarrow st'$ and $st \rightsquigarrow st''$, and
  state changes $st \rightsquigarrow (st' \oplus_{st} st'')$ and $st \rightsquigarrow st'''$ are consistent
  $\Leftrightarrow$

  (1a) $A'_s \cap A''_s = A_s$
  $(A'_s \cup A''_s) \cap A'''_s = A_s$

  (2a) if $\underline{a} \in A_{\underline{s}}$ then:
  $f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}''}(\underline{a})$
  $f_{\mathcal{A}'}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a})$ implies
  $\Big(f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}'''}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}'''}(\underline{a})\Big)$
  $f_{\mathcal{A}''}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a})$ implies
  $\Big(f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}'''}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}'''}(\underline{a})\Big)$
  $f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ implies
  $\Big(f_{\mathcal{A}}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}'''}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}}(\underline{a}) = f_{\mathcal{A}'''}(\underline{a})\Big)$

  (3a) $v'_s(x) = v_s(x)$ or $v''_s(x) = v_s(x)$ or $v'_s(x) = v''_s(x)$
  $v'_s(x) \neq v_s(x)$ implies
  $\Big(v'_s(x) = v_s(x)$ or $v'''_s(x) = v_s(x)$ or $v'_s(x) = v'''_s(x)\Big)$
  $v''_s(x) \neq v_s(x)$ implies
  $\Big(v''_s(x) = v_s(x)$ or $v'''_s(x) = v_s(x)$ or $v''_s(x) = v'''_s(x)\Big)$
  $v'_s(x) = v''_s(x) = v_s(x)$ implies
  $\Big(v_s(x) = v_s(x)$ or $v'''_s(x) = v_s(x)$ or $v_s(x) = v'''_s(x)\Big)$

- state changes $st \rightsquigarrow st''$ and $st \rightsquigarrow st'''$, and
  state changes $st \rightsquigarrow st'$ and $st \rightsquigarrow (st'' \oplus_{st} st''')$ are consistent
  $\Leftrightarrow$

  (1b) $A''_s \cap A'''_s = A_s$
  $A'_s \cap (A''_s \cup A'''_s) = A_s$

  (2b) if $\underline{a} \in A_{\underline{s}}$ then:
  $f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}'''}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}'''}(\underline{a})$

$f_{\mathcal{A}''}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a})$ implies
$\Big(f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}''}(\underline{a})\Big)$
$f_{\mathcal{A}'''}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a})$ implies
$\Big(f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}'''}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}'''}(\underline{a})\Big)$
$f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}'''}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ implies
$\Big(f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}}(\underline{a}) = f_{\mathcal{A}}(\underline{a})$ or $f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}}(\underline{a})\Big)$

(3b) $v_s''(x) = v_s(x)$ or $v_s'''(x) = v_s(x)$ or $v_s''(x) = v_s'''(x)$
$v_s''(x) \neq v_s(x)$ implies
$\Big(v_s'(x) = v_s(x)$ or $v_s''(x) = v_s(x)$ or $v_s'(x) = v_s''(x)\Big)$
$v_s'''(x) \neq v_s(x)$ implies
$\Big(v_s'(x) = v_s(x)$ or $v_s'''(x) = v_s(x)$ or $v_s'(x) = v_s'''(x)\Big)$
$v_s''(x) = v_s'''(x) = v_s(x)$ implies
$\Big(v_s'(x) = v_s(x)$ or $v_s(x) = v_s(x)$ or $v_s'(x) = v_s(x)\Big)$

We proceed with proving

$$\begin{array}{ccc} (1\text{a}) & \Leftrightarrow & (1\text{b}) \\ (2\text{a}) & \Leftrightarrow & (2\text{b}) \\ (3\text{a}) & \Leftrightarrow & (3\text{b}). \end{array}$$

$$(1a) \;\Leftrightarrow\; \boxed{\begin{array}{r} A_s' \supseteq A_s \\ A_s'' \supseteq A_s \\ A_s''' \supseteq A_s \\ A_s' \cap A_s'' \subseteq A_s \\ A_s' \cap A_s''' \subseteq A_s \\ A_s'' \cap A_s''' \subseteq A_s \end{array}} \;\Leftrightarrow\; (1b).$$

$$(2a) \;\Leftrightarrow\; \boxed{\begin{array}{l} f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \text{ or } f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \text{ or } f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}''}(\underline{a}) \\ f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \text{ or } f_{\mathcal{A}'''}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \text{ or } f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}'''}(\underline{a}) \\ f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \text{ or } f_{\mathcal{A}'''}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \text{ or } f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}'''}(\underline{a}) \end{array}}$$

$$\Leftrightarrow \boxed{\begin{array}{l} f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \text{ or } f_{\mathcal{A}'''}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \text{ or } f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}'''}(\underline{a}) \\ f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \text{ or } f_{\mathcal{A}''}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \text{ or } f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}''}(\underline{a}) \\ f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \text{ or } f_{\mathcal{A}'''}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \text{ or } f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}'''}(\underline{a}) \end{array}}$$

$$\Leftrightarrow \quad (2b).$$

$$(3a) \;\Leftrightarrow\; \boxed{\begin{array}{l} v_s'(x) = v_s(x) \text{ or } v_s''(x) = v_s(x) \text{ or } v_s'(x) = v_s''(x) \\ v_s'(x) = v_s(x) \text{ or } v_s'''(x) = v_s(x) \text{ or } v_s'(x) = v_s'''(x) \\ v_s''(x) = v_s(x) \text{ or } v_s'''(x) = v_s(x) \text{ or } v_s''(x) = v_s'''(x) \end{array}}$$

$$\Leftrightarrow \boxed{\begin{array}{l} v_s''(x) = v_s(x) \text{ or } v_s'''(x) = v_s(x) \text{ or } v_s''(x) = v_s'''(x) \\ v_s''(x) = v_s(x) \text{ or } v_s'(x) = v_s(x) \text{ or } v_s'(x) = v_s''(x) \\ v_s'''(x) = v_s(x) \text{ or } v_s'(x) = v_s(x) \text{ or } v_s'(x) = v_s'''(x) \end{array}}$$

$$\Leftrightarrow \quad (3b).$$

It remains to show that

$$st^{ll} := \underbrace{(st' \oplus_{st} st'')}_{=:st^l} \oplus_{st} st''' = st' \oplus_{st} \underbrace{(st'' \oplus_{st} st''')}_{=:st^r} =: st^{rr}$$

if the corresponding state changes are consistent. With

$$\begin{aligned} st^{ll} &= (\mathcal{A}^{ll}, v^{ll}) \\ st^{l} &= (\mathcal{A}^{l}, v^{l}) \\ st^{rr} &= (\mathcal{A}^{rr}, v^{rr}) \\ st^{r} &= (\mathcal{A}^{r}, v^{r}) \end{aligned}$$

it holds

$$\begin{aligned} A_s^{ll} &= (A'_s \cup A''_s) \cup A'''_s \\ &= A'_s \cup (A''_s \cup A'''_s) \\ &= A_s^{rr}. \end{aligned}$$

Furthermore

$$\begin{aligned} f_{\mathcal{A}^{ll}}(\underline{a}) &= \begin{cases} UNDEF_s & \text{, if } \underline{a} \in A_{\underline{s}}^{ll} \setminus (A_{\underline{s}}^{l} \cup A_{\underline{s}}'''') \\ f_{\mathcal{A}^{l}}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}}^{l} \setminus A_{\underline{s}} \\ f_{\mathcal{A}'''}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}}''' \setminus A_{\underline{s}} \\ f_{\mathcal{A}^{l}}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}} \text{ and } f_{\mathcal{A}^{l}}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a}) \\ f_{\mathcal{A}'''}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}} \text{ and } f_{\mathcal{A}'''}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a}) \\ f_{\mathcal{A}}(\underline{a}) & \text{, otherwise} \end{cases} \\ &= \begin{cases} UNDEF_s & \text{, if } \underline{a} \in A_{\underline{s}}^{ll} \setminus (A'_{\underline{s}} \cup A''_{\underline{s}} \cup A'''_{\underline{s}}) \\ f_{\mathcal{A}'}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}}' \setminus A_{\underline{s}} \\ f_{\mathcal{A}''}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}}'' \setminus A_{\underline{s}} \\ f_{\mathcal{A}'''}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}}''' \setminus A_{\underline{s}} \\ f_{\mathcal{A}'}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}} \text{ and } f_{\mathcal{A}'}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a}) \\ f_{\mathcal{A}''}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}} \text{ and } f_{\mathcal{A}''}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a}) \\ f_{\mathcal{A}'''}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}} \text{ and } f_{\mathcal{A}'''}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a}) \\ f_{\mathcal{A}}(\underline{a}) & \text{, otherwise} \end{cases} \\ &= \begin{cases} UNDEF_s & \text{, if } \underline{a} \in A_{\underline{s}}^{rr} \setminus (A'_{\underline{s}} \cup A_{\underline{s}}^{r}) \\ f_{\mathcal{A}'}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}}' \setminus A_{\underline{s}} \\ f_{\mathcal{A}^{r}}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}}^{r} \setminus A_{\underline{s}} \\ f_{\mathcal{A}'}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}} \text{ and } f_{\mathcal{A}'}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a}) \\ f_{\mathcal{A}^{r}}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}} \text{ and } f_{\mathcal{A}^{r}}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a}) \\ f_{\mathcal{A}}(\underline{a}) & \text{, otherwise} \end{cases} \\ &= f_{\mathcal{A}^{rr}}(\underline{a}) \end{aligned}$$

where we have used that

$$\begin{aligned}
& f_{\mathcal{A}^{ll}}(\underline{a}) = UNDEF_s \\
\Leftrightarrow\ & \underline{a} \in \left(A^{ll}_{\underline{s}} \setminus (A^{l}_{\underline{s}} \cup A'''_{\underline{s}})\right) \cup \left(A^{l}_{\underline{s}} \setminus (A'_{\underline{s}} \cup A''_{\underline{s}})\right) \\
\Leftrightarrow\ & \underline{a} \in A^{ll}_{\underline{s}} \setminus (A'_{\underline{s}} \cup A''_{\underline{s}} \cup A'''_{\underline{s}}) \\
\Leftrightarrow\ & \underline{a} \in A^{rr}_{\underline{s}} \setminus (A'_{\underline{s}} \cup A''_{\underline{s}} \cup A'''_{\underline{s}}) \\
\Leftrightarrow\ & \underline{a} \in \left(A^{rr}_{\underline{s}} \setminus (A'_{\underline{s}} \cup A^{r}_{\underline{s}})\right) \cup \left(A^{r}_{\underline{s}} \setminus (A''_{\underline{s}} \cup A'''_{\underline{s}})\right) \\
\Leftrightarrow\ & f_{\mathcal{A}^{rr}}(\underline{a}) = UNDEF_s.
\end{aligned}$$

Finally it holds

$$\begin{aligned}
v^{ll}_s(x) &= \begin{cases} v^{l}_s(x) & \text{, if } v^{l}_s(x) \neq v_s(x) \\ v'''_s(x) & \text{, if } v'''_s(x) \neq v_s(x) \\ v_s(x) & \text{, otherwise} \end{cases} \\
&= \begin{cases} v'_s(x) & \text{, if } v'_s(x) \neq v_s(x) \\ v''_s(x) & \text{, if } v''_s(x) \neq v_s(x) \\ v'''_s(x) & \text{, if } v'''_s(x) \neq v_s(x) \\ v_s(x) & \text{, otherwise} \end{cases} \\
&= \begin{cases} v'_s(x) & \text{, if } v'_s(x) \neq v_s(x) \\ v^{r}_s(x) & \text{, if } v^{r}_s(x) \neq v_s(x) \\ v_s(x) & \text{, otherwise} \end{cases} \\
&= v^{rr}_s(x).
\end{aligned}$$

which finishes the proof of $st^{ll} = st^{rr}$ and so of **(b)**.

In **(c)** the consistency of the state changes $st \rightsquigarrow st$ and $st \rightsquigarrow st'$ is quite obvious.

$$st^{\oplus} := st \oplus_{st} st' \quad = \quad st' \quad \text{where } st^{\oplus} =: (\mathcal{A}^{\oplus}, v^{\oplus})$$

follows from:

$$A^{\oplus}_s = A_s \cup A'_s = A'_s$$

$$\begin{aligned}
f_{\mathcal{A}^{\oplus}}(\underline{a}) &= \begin{cases} UNDEF_s & \text{, if } \underline{a} \in A^{\oplus}_{\underline{s}} \setminus (A_{\underline{s}} \cup A'_{\underline{s}}) = \emptyset \\ f_{\mathcal{A}'}(\underline{a}) & \text{, if } \underline{a} \in A'_{\underline{s}} \setminus A_{\underline{s}} \\ f_{\mathcal{A}'}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}} \text{ and } f_{\mathcal{A}'}(\underline{a}) \neq f_{\mathcal{A}}(\underline{a}) \\ f_{\mathcal{A}}(\underline{a}) & \text{, if } \underline{a} \in A_{\underline{s}} \text{ and } f_{\mathcal{A}'}(\underline{a}) = f_{\mathcal{A}}(\underline{a}) \end{cases} \\
&= f_{\mathcal{A}'}(\underline{a})
\end{aligned}$$

$$\begin{aligned}
v^{\oplus}_s(x) &= \begin{cases} v'_s(x) & \text{, if } v'_s(x) \neq v_s(x) \\ v'_s(x) & \text{, if } v'_s(x) = v_s(x) \end{cases} \\
&= v'_s(x).
\end{aligned}$$

This concludes the proof for fact 3.8. ■

B.3 Proof for Theorem 3.10

Proof. **(a)**, **(b)**, and **(c)** follow directly from fact 3.8. In **(c)** it is employed that no program from $EPROG(SIG, X)$ is able to discard elements from universes.

The proof for **(d)** works as follows:

$st\,[\![((\alpha_1 \cup \alpha_2), \beta)]\!]\,st'$

$\Leftrightarrow$ there are st_α, st_β such that ($st\,[\![\alpha_1]\!]\,st_\alpha$ or $st\,[\![\alpha_2]\!]\,st_\alpha$) and $st\,[\![\beta]\!]\,st_\beta$, and the state changes $st \rightsquigarrow st_\alpha$ and $st \rightsquigarrow st_\beta$ are consistent with $st' = st_\alpha \oplus_{st} st_\beta$

$\Leftrightarrow$ (there are st_α, st_β such that $st\,[\![\alpha_1]\!]\,st_\alpha$ and $st\,[\![\beta]\!]\,st_\beta$, and the state changes $st \rightsquigarrow st_\alpha$ and $st \rightsquigarrow st_\beta$ are consistent with $st' = st_\alpha \oplus_{st} st_\beta$)
or
(there are st_α, st_β such that $st\,[\![\alpha_2]\!]\,st_\alpha$ and $st\,[\![\beta]\!]\,st_\beta$, and the state changes $st \rightsquigarrow st_\alpha$ and $st \rightsquigarrow st_\beta$ are consistent with $st' = st_\alpha \oplus_{st} st_\beta$)

$\Leftrightarrow$ $st\,[\![((\alpha_1, \beta) \cup (\alpha_2, \beta))]\!]\,st'$

(e) can be proved very similar to **(d)**. It remains to show **(f)**, which is done in the rest of this subsection. The two implications of the equivalence

$$st[\![(\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha, \beta)]\!]st' \Leftrightarrow st[\![\textbf{extend } s \textbf{ by } x' \textbf{ with } (\alpha_x^{x'}, \beta)]\!]st'$$

are proven separately. For the first one we assume

$$st\,[\![(\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha, \beta)]\!]\,st'$$

which means that there are states $st_\alpha, st_\beta, st''$ and an $a \in A''_s \setminus A_s$ such that

(1) $(st +_s \{a\})[x \leftarrow a]\,[\![\alpha]\!]\,st''$

(2) $st_\alpha = st''[x \leftarrow st(x)]$

(3) $st\,[\![\beta]\!]\,st_\beta$

(4) $st \rightsquigarrow st_\alpha$ and $st \rightsquigarrow st_\beta$ consistent with $st_\alpha \oplus_{st} st_\beta = st'$.

By applying fact B.1(c) on (1) and choosing

$$\overline{st_\alpha} \;:=\; st''[x \leftarrow st(x)][x' \leftarrow st''(x)]$$

we get

(1') $(st +_s \{a\})[x' \leftarrow a] \, [\![\alpha_x^{x'}]\!] \, \overline{st_\alpha}$.

By applying fact B.1(b) and B.1(a) on (3) and choosing

$$\overline{st_\beta} \quad := \quad (st_\beta +_s \{a\})[x' \leftarrow a]$$

we get

(2') $(st +_s \{a\})[x' \leftarrow a] \, [\![\beta]\!] \, \overline{st_\beta}$.

If we further set $\overline{st''} := (st_\alpha \oplus_{st} st_\beta)[x' \leftarrow st''(x)]$ it follows

(3') $st' = \overline{st''}[x' \leftarrow st(x')]$

from (4) and the fact that x' does not occur in α or β. Applying lemma B.2(b) and B.2(a) on (4) yields that the state changes $(st +_s \{a\})[x' \leftarrow a] \rightsquigarrow st_\alpha$ and $(st +_s \{a\})[x' \leftarrow a] \rightsquigarrow \overline{st_\beta}$ are consistent with

$$st_\alpha \oplus_{(st +_s \{a\})[x' \leftarrow a]} \overline{st_\beta} \quad = \quad st_\alpha \oplus_{st} st_\beta.$$

Together with lemma B.2(c) and the fact that $st_\alpha[x' \leftarrow st''(x)] = \overline{st_\alpha}$ (which is due to (2)) follows that

(4') $(st +_s \{a\})[x' \leftarrow a] \rightsquigarrow \overline{st_\alpha}$ and $(st +_s \{a\})[x' \leftarrow a] \rightsquigarrow \overline{st_\beta}$ are consistent with $\overline{st_\alpha} \oplus_{(st +_s \{a\})[x' \leftarrow a]} \overline{st_\beta} = \overline{st''}$.

Summarizing, we have shown that there are states $\overline{st''}, \overline{st_\alpha}, \overline{st_\beta}$ and an $a \in \overline{A_s''} \setminus A_s$ such that (1'), (2'), (3') and (4') hold. By definition this implies

$$st \, [\![\textbf{extend } s \textbf{ by } x' \textbf{ with } (\alpha_x^{x'}, \beta)]\!] \, st'.$$

For the proof of the other implication we assume

$$st \, [\![\textbf{extend } s \textbf{ by } x' \textbf{ with } (\alpha_x^{x'}, \beta)]\!] \, st'$$

i.e. that there are states $\overline{st''}, \overline{st_\alpha}, \overline{st_\beta}$ and an $a \in \overline{A_s''} \setminus A_s$ with

(1') $(st +_s \{a\})[x' \leftarrow a] \, [\![\alpha_x^y]\!] \, \overline{st_\alpha}$

(2') $(st +_s \{a\})[x' \leftarrow a] \, [\![\beta]\!] \, \overline{st_\beta}$

(3') $st' = \overline{st''}[x' \leftarrow st(x')]$

(4') $(st +_s \{a\})[x' \leftarrow a] \rightsquigarrow \overline{st_\alpha}$ and $(st +_s \{a\})[x' \leftarrow a] \rightsquigarrow \overline{st_\beta}$ are consistent with $\overline{st_\alpha} \oplus_{(st +_s \{a\})[x' \leftarrow a]} \overline{st_\beta} = \overline{st''}$.

By applying fact B.1(c) on (1') and choosing

$$st'' \quad := \quad \overline{st_\alpha}[x' \leftarrow (st +_s \{a\})(x')][x \leftarrow \overline{st_\alpha}(x)]$$

we get

(1) $(st +_s \{a\})[x \leftarrow a] [\![\alpha]\!] st''$

since $(\alpha_x^{x'})_{x'}^x \equiv \alpha$. Because of (1') is $\overline{st_\alpha}(x) = st(x)$, and so

$$\begin{aligned} \overline{st_\alpha}[x' \leftarrow st(x')] &= \overline{st_\alpha}[x' \leftarrow st(x')][x \leftarrow st(x)] \\ &= st''[x \leftarrow st(x)] \end{aligned}$$

Thus we get

(2) $st_\alpha = st''[x \leftarrow st(x)]$

for $st_\alpha := \overline{st_\alpha}[x' \leftarrow st(x')]$. Applying fact B.1(d) on (2') yields

$$(st +_s \{a\}) [\![\beta]\!] \overline{st_\beta}[x' \leftarrow (st +_s \{a\})(x')].$$

Together with fact B.1(e) follows that there is a state st_β with

(3) $st [\![\beta]\!] st_\beta$

and $st_\beta +_s \{a\} = \overline{st_\beta}[x' \leftarrow st(x')]$ (for an $a \notin A_s^\beta$). From (4') follows that $(st +_s \{a\}) \rightsquigarrow \overline{st_\alpha}[x' \leftarrow st(x')]$ and $(st +_s \{a\}) \rightsquigarrow \overline{st_\beta}[x' \leftarrow st(x')]$ are consistent with

$$\overline{st_\alpha}[x' \leftarrow st(x')] \oplus_{(st +_s \{a\})} \overline{st_\beta}[x' \leftarrow st(x')] = \overline{st''}[x' \leftarrow st(x')]$$

where the last equation is due to (3'). which means that $(st +_s \{a\}) \rightsquigarrow st_\alpha$ and $(st +_s \{a\}) \rightsquigarrow (st_\beta +_s \{a\})$ are consistent with $st_\alpha \oplus_{(st +_s \{a\})} (st_\beta +_s \{a\}) = \overline{st''}[x' \leftarrow st(x')] = st'$, where the last equation is due to (3'). Applying lemma B.2(d) results in

(4) $st \rightsquigarrow st_\alpha$ and $st \rightsquigarrow st_\beta$ are consistent with $st_\alpha \oplus_{st} st_\beta = st'$.

Summarizing, we have shown that there are three states $st_\alpha, st_\beta, st''$ and an $a \in A_s'' \setminus A_s$ such that (1), (2), (3) and (4) hold. By definition this implies

$$st [\![(\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha, \beta)]\!] st'.$$

This concludes the proof of theorem 3.10.

■

B.4 Proof for Lemma 5.5

Proof. Intuitively, **(a)** holds because the states $(st +_s \{a\})[x \leftarrow a]$ and $(st +_s \{b\})[x \leftarrow b]$ are isomorphic whenever $a, b \notin A_s$. Technically, **(a)** can be proved by structural induction on the formula φ.

(b) is clear, since $a \notin A_s$.

The proof of **(c)** works by structural induction on the formula φ. We present only the most interesting case, i.e. where $\varphi \equiv \forall x.\psi$ for some $x \in X_s$, $\psi \in EDL(SIG, X)$:

$$
\begin{aligned}
& st \models \forall x.\psi \\
& \quad \Leftrightarrow \quad st[x \leftarrow b] \models \psi \text{ for all } b \in A_s \\
& \quad \Leftrightarrow \quad \big(st[x \leftarrow b] +_s \{a\}\big)[x' \leftarrow a] \models \psi^{|x'} \text{ for all } b \in A_s \\
& \quad \Leftrightarrow \quad (st +_s \{a\})[x' \leftarrow a][x \leftarrow b] \models \psi^{|x'} \text{ for all } b \in A_s \\
& \quad \Leftrightarrow \quad \big(b = a \text{ or } (st +_s \{a\})[x' \leftarrow a][x \leftarrow b] \models \psi^{|x'}\big) \text{ for all } b \in A_s \cup \{a\} \\
& \quad \Leftrightarrow \quad (st +_s \{a\})[x' \leftarrow a][x \leftarrow b] \models (x = x' \vee \psi^{|x'}) \text{ for all } b \in A_s \cup \{a\} \\
& \quad \Leftrightarrow \quad (st +_s \{a\})[x' \leftarrow a] \models \forall x.(x = x' \vee \psi^{|x'}).
\end{aligned}
$$

Here the second equivalence is due to the induction hypothesis.

It remains to show **(d)**. Due to the following equivalences

$$
\begin{aligned}
& st \models [\textbf{extend } s \textbf{ by } x \textbf{ with } \alpha]\varphi \\
& \quad \Leftrightarrow \quad st' \models \varphi \text{ for all } st' \text{ such that there is an } st'' \text{ and an } a \in A''_s \setminus A_s \\
& \qquad \text{with } (st +_s \{a\})[x \leftarrow a] \, [\![\alpha]\!] \, st'' \text{ and } st' = st''[x \leftarrow st(x)] \\
& \quad \Leftrightarrow \quad st''[x \leftarrow st(x)] \models \varphi \text{ for all } st'' \text{ such that there is an } a \in A''_s \setminus A_s \\
& \qquad \text{with } (st +_s \{a\})[x \leftarrow a] \, [\![\alpha]\!] \, st'' \qquad \text{(A)}
\end{aligned}
$$

$$
\begin{aligned}
& (st +_s \{a\})[x' \leftarrow a] \models [\alpha^{x'}_x]\varphi \text{ for all } a \notin A_s \\
& \quad \Leftrightarrow \quad st' \models \varphi \text{ for all } st' \text{ and all } a \in A'_s \setminus A_s \\
& \qquad \text{with } (st +_s \{a\})[x' \leftarrow a] \, [\![\alpha^{x'}_x]\!] \, st' \\
& \quad \Leftrightarrow \quad st' \models \varphi \text{ for all } st' \text{ such that there is an } a \in A'_s \setminus A_s \\
& \qquad \text{with } (st +_s \{a\})[x' \leftarrow a] \, [\![\alpha^{x'}_x]\!] \, st' \qquad \text{(B)}
\end{aligned}
$$

it is sufficient to prove

$$\text{(A)} \quad \Leftrightarrow \quad \text{(B)}.$$

Assuming (A) and choosing $st'' = st'[x' \leftarrow st(x')][x \leftarrow st'(x')]$ one gets that

$$
\begin{aligned}
& st'[x' \leftarrow st(x')][x \leftarrow st'(x')][x \leftarrow st(x)] \models \varphi \\
& \text{for all } st' \text{ such that there is an } a \in A'_s \setminus A_s \text{ with} \\
& (st +_s \{a\})[x \leftarrow a] \, [\![(\alpha^{x'}_x)^x_{x'}]\!] \, st'[x' \leftarrow st(x')][x \leftarrow st'(x')].
\end{aligned}
$$

Together with fact B.1(c) follows that

$$\begin{array}{l} st'[x' \leftarrow st(x')][x \leftarrow st(x)] \models \varphi \\ \text{for all } st' \text{ such that there is an } a \in A'_s \setminus A_s \text{ with} \\ (st +_s \{a\})[x' \leftarrow a] \, [\![\alpha_x^{x'}]\!] \, st'. \end{array}$$

Since x' does not occur in φ and $st'(x) = st(x)$ (because x does not occur in $\alpha_x^{x'}$) it follows

$$st'[x' \leftarrow st(x')][x \leftarrow st(x)] \models \varphi \quad \Leftrightarrow \quad st' \models \varphi$$

and thus (B).

On the other hand, assuming (B) and choosing

$$st' \quad = \quad st''[x \leftarrow st(x)][x' \leftarrow st''(x)]$$

leads to

$$\begin{array}{l} st''[x \leftarrow st(x)][x' \leftarrow st''(x)] \models \varphi \\ \text{for all } st'' \text{ such that there is an } a \in A''_s \setminus A_s \text{ with} \\ (st +_s \{a\})[x' \leftarrow a] \, [\![\alpha_x^{x'}]\!] \, st''[x \leftarrow st(x)][x' \leftarrow st''(x)]. \end{array}$$

Again with fact B.1(c) follows that

$$\begin{array}{l} st''[x \leftarrow st(x)][x' \leftarrow st''(x)] \models \varphi \\ \text{for all } st'' \text{ such that there is an } a \in A''_s \setminus A_s \text{ with} \\ (st +_s \{a\})[x \leftarrow a] \, [\![\alpha]\!] \, st''. \end{array}$$

Since x' does not occur in φ it is

$$st''[x \leftarrow st(x)][x' \leftarrow st''(x)] \models \varphi \quad \Leftrightarrow \quad st''[x \leftarrow st(x)] \models \varphi$$

and thus (A).

This concludes the proof of lemma 5.5. ■