Minimal Transition System Semantics for Lightweight Class– and Behavior Diagrams*

Roel Wieringa
Faculty of Computer Science, *University of Twente*
P.O. Box 217, 7500 AE Enschede, the Netherlands
roelw@cs.utwente.nl

Jan Broersen†
Faculty of Mathematics and Computer Science, *Vrije Universiteit*
De Boelelaan 1081a, 1081 HV Amsterdam, the Netherlands
broersen@cs.vu.nl

Abstract

We give a semantics for lightweight UML class diagrams and ultra-lightweight statecharts in terms of labeled step transition systems that embody a minimal change, maximal step semantics, and in which changes generated in a step have effect in the following step. In order to define the semantics, we introduce dynamic step logic.

1 Introduction

1.1 Purpose

In this paper we define a mixed formal specification language whose diagram syntax is a very simple subset of the UML syntax [27] and whose textual syntax is a version of order-sorted dynamic logic. We refer to this language as ultra-lightweight UML (ultra-LUML). The visual part of ultra-LUML consists of ultra-leightweight class diagrams (ultra-LCDs), which declare the decomposition of the system into objects, and of extended Mealy state diagrams (EMDs), which define object behavior. The textual part of utra-LUML is a version of dynamic logic and can be used to write a specification that is equivalent to a diagrammatic specification. In addition, the textual part can be used to specify static constraints, the effect of operations on the objects, and the guards of object state transitions.

A model of an ultra-LUML specification is a **transition system**, which is a possibly infinite set of possible states and state transitions, where each state transition is labeled with a finite set of operations, all executed by different objects during that transition. This finite set of object operations is called a **step**. Among all possible models of an ultra-LUML specification, we define a unique intended model that has the following properties.

*Supported by Esprit Working Group Aspire, contact nr. 22704.
†Supported by USF-VU as part of the SINS contract.

- **Maximal steps.** Each step contains a maximal set of enabled object transitions that are mutually nonconflicting.
- **Maximal reachability.** Each maximal step is enabled whenever this is compatible with the guards of object state transitions and with the static constraints.
- **Minimal change**. The effect of each step is the smallest possible change compatible with the operation effect specifications.
- **Sequential supersteps.** A step may generate signals that trigger other object state transitions and operations. These are performed in the next step and not in the current step. The total response to external events may thus consist of a sequence of steps, that we call a **superstep**.

We next motivate this semantics and then compare our approach with that of others.

1.2 Motivation

To motivate our semantic choices, we distinguish a number of kinds of specifications. A **requirements specification** is a description of the desired way of working in the external environment of a system. Each system is ultimately embedded in a social world in which people try to achieve goals. The system under design is intended to help people achieving their goals, and the requirements specification is a description of what people wish to do to achieve their goals. So a requirements specification should not refer to the system under design but to the desired way of working of people.

An **external property specification** of a system is a description of the desired externally observable properties of a system that would contribute to realizing the requirements of people. An important class of such properties can be described in terms of the desired response to external events. The desired system properties are derived from and motivated by the requirements. Where the requirements specification is about the things people wish to do in a social world and not about the system under design, an external property specification is about the system.

An **essential decomposition** of the system is a decomposition that the system would have if it were implemented in perfect technology. Because this decomposition cannot be motivated by reference to an underlying implementation technology, it must be motivated by reference to entities or activities in the external environment, i.e. by reference to the requirements and desired external properties. This means, among other things, that a response to an event must be produced instantly, unless a delay is desired by the requirements. An essential decomposition is really an implementation that assumes perfect technology. An essential decomposition specification is therefore analogous to McMenamin and Palmer's essential model [24] and also to Syntropy's specification model [8]. The ability to respond to external events instantly is called the synchrony hypothesis [2]. It is satisfied by an actual implementation if that implementation finishes the response to each event before the next event is available. The methodological role of an essential decomposition is that it allows us to show that the desired external properties can, in principle, be implemented. Also, because an essential decomposition is defined in terms of the external environment, it can be validated, at least in principle, by a domain expert who is not a computer expert. Finally, an essential decomposition shows in an implementation-independent way what any implementation of the system must realize. It remains invariant under changes of implementation technology and is therefore a useful reference point for exploring alternative implementations. For the same reason, it helps realizing traceability of external properties to implementation components.

An **implementation decomposition** is any decomposition that takes the underlying implementation technology into account. In an implementation decomposition, actions take time, there is a finite number of processors, processors may be unavailable, memory is limited, errors may occur, connections may break down, etc. There may be more than one implementation decompositions, each at a different level of abstraction. For example, one decomposition may show the physical computer network and another may show the decomposition of essential components into sequential processes that can be run by a processor.

The motivation for our semantics of LUML stems from our intention to use LUML for essential decomposition specifications. At this level, the system consists of a finite number of machines that communicate instantly and that have infinite processing speed. Let us call these machines *objects*. When an external event arrives at the system, it may trigger one or more objects, each of which may respond by performing a state transition. Each of these transitions may trigger other transitions, etc. until all responses have been produced. There are many different ways to make this precise. This is the classical problem of statechart semantics, to which we return shortly. We expect our perfect technology to behave as follows.

- When an event e arrives, it may trigger more than one transition. If an object is nondeterministic, it may even trigger two transitions in one object. Since these cannot be taken at the same time, they are conflicting. This means that in general, more than one step (finite set of transitions, each of which is taken by a different object) may be triggered, where each step consists of nonconflicting transitions. Our first semantic choice is that we only allow maximal steps of nonconflicting transitions to take place.

- Each object transition is guarded by a condition that must be true if the transition is to be taken. Our second semantic choice is that each step has a guard that consists of the conjunction of the guards of the object transitions in it. This means that a step is enabled if all transitions in it are enabled. Together, our first and second choice mean that an event triggers maximal steps of nonconflicting object transitions that are enabled.

- When a step is taken, all transitions in it are taken. This means that there are no lazy objects, that do nothing when they are enabled and triggered. Each transition is performed by an object and may change the state of this object. These changes are specified by means of pre/postcondition statements for actions performed by objects. These only specify what changes, and leave open that some other things may change as well. If each action performed by an object were specified by saying "and nothing else changes", then actions of two different objects could not be performed in parallel and so a step could not contain more than one action. The phrase "nothing else changes" should therefore be applied to steps as a whole and not to the object transitions in the step. This is our third semantic choice.

- A step may generate events that may trigger other objects. There are two options to give this a semantics. In the Statemate semantics of statecharts [18, 16], these events are responded to in the next step performed by the system. We call this the **sequential** semantics of statecharts. It is easy to show that there are systems whose response to a step never terminates if this response is produced sequentially. In the mathematically more pleasing **fixed point** semantics of Pnueli and Shalev [26], the triggered transitions are added to the step. This is continued until no new transitions are triggered by the step that triggered them. Since there are finitely many objects, this process always terminates. The fixed point semantics is mathematically pleasing but leads to counterintuitive statechart behavior [22].[‡] For this reason, we chose a sequential step semantics.

These semantic choices agree with our intended use of the UML for representing essential decompositions. They differ from what seems to become the OMG semantics of the UML, as informally outlined [28]. A proposal for a formalization of the OMG semantics of the statechart part of the UML is under development [14, 15]. This semantics represents the execution behavior of an object-oriented program that implements a UML model. It is a semantics that is appropriate to the implementation level and not to the essential level. For example, in the current version of the UML semantics, actions take time and communication takes place by means of message-passing. Under the OMG semantics, the UML would develop into a graphical programming language. The intention we have in this paper is to outline a semantics that should be appropriate for higher levels of software modeling.

1.3 Comparison with other work

This paper continues research undertaken earlier on the formalization of visual object-oriented software specification techniques [33, 34, 35, 36]. It contains a summary of the essentials of this formalization, sufficient to formalize ultra-LUML. The additions to this earlier work consist of, first, a different version of dynamic logic, that includes a simple form of parallelism with a transition system semantics. Second, we now give a precisely defined minimal model semantics for class specifications, where this was defined by means of hand-waving earlier. Third, we now use Mealy machines where we used process graphs earlier, and fourth, the definition of the intended transition system semantics of ultra-LUML specifications is new.

Dynamic logic has been used to specify database updates in modal action logic (MAL) [12, 19, 21]. However, our semantics is different, because we give a two-level semantics to update operation symbols, as explained below. Also, our data terms are rigid designators (have the same interpretation in all possible states), which agrees well with the concept of object identity central to object-oriented conceptual modeling. Later developments by Fiadeiro, Maibaum and Ryan focus on formalizing objects as modal theories and structuring specifications by putting these theories together by morphisms [9, 30].

[‡]Leveson [22] attributes this semantics mistakenly to Harel [16] and Statemate [18].

Bourdeau and Chen [3] give an algebraic semantics to class diagrams, using Larch. This is replaced in recent work by Lotos [31]. The data specification part of Lotos is still used to specify the static system structure and the process algebra part is used to specify behavior diagrams (statecharts). It is not clear what the relationship is between this statechart semantics and other published execution semantics of statecharts. Their approach of using equational logic for the static as well as dynamic model resembles the algebraic approach of our earlier work, where we used an OBJ-like language combined with process algebra [37, 32].

France et al. [10] formalize 1994 vintage Fusion models [7] in terms of Z and present an environment that translates Fusion models into Z specifications that are amenable to type checking and animation. The precise semantics of the integration of behavior (life cycles) and class models is left open. The 1994 version of Fusion defines life cycles at the system rather than at the object level. In later versions of Fusion, a different life cycle notation is used [6] and when Fusion adopts the UML, statecharts will be used.

Breu et al. [4] sketch an approach to formalizing the UML using a stream-based model of object behavior and asynchronous message-passing as communication model, where each object has a queue of input messages. The underlying behavior model is a timed version of I/O automata [23], described by [13]. This differs from our approach in that we ignore time (for the moment), and use distributed transition systems as a semantic structure, with synchronous step-based semantics for state machine behavior and communication. We believe this kind of model is more appropriate for an execution semantics.

The structure of the rest of this paper is as follows. In section 2, we define the version of dynamic logic used in this paper. Section 3 defines the syntax and semantics of LCDs and section 4 extends this with the syntax and semantics of EMDs. Section 5 winds up the paper with a summary of results and a discussion of the extension of ultra-LUML to LUML. All proofs are omitted in this workshop paper. They are given by Broersen and Wieringa [5].

2 Dynamic Logic

2.1 Order-sorted logic

We start from an order-sorted logic that can be used to specify abstract datatypes. This section is based upon the approach of Goguen and Meseguer [11]. An **order-sorted equational signature** $Sig = ((\mathcal{S} \leq), \mathcal{F}, \mathcal{P})$ is a tuple consisting of the following sets:

- A partially ordered set $(\mathcal{S}, \leq)$ of **sort names**.
- A set $\mathcal{F}$ of **function symbol declarations** of the form $f : S_1 \times \cdots \times S_n \to S_0$ with $S_i \in \mathcal{S}$ for $i = 1, \ldots, n$ and with $n \geq 0$. If $n = 0$, then f is called a **constant**.
- A set $\mathcal{P}$ of **predicate symbol declarations** of the form $P : S_1 \times \cdots \times S_n$ with $S_i \in \mathcal{S}$ for $i = 1, \ldots, n$ and with $n \geq 0$. If $n = 0$, P is called a **proposition**.
- There is a universal sort $U \in \mathcal{S}$ such that $S \leq U$ for all $S \in \mathcal{S}$ and there is an equality symbol declared as $=: U \times U$ in $\mathcal{P}$.

We assume that for every $S \in \mathcal{S}$ there is an infinite set X_S of variables and that all these sets are mutually disjoint for different S. We write $X = \{X_S \mid S \in \mathcal{S}\}$. The set $T_{Sig,S}(X)$ of **terms** of sort S is defined in the usual manner. A term containing variables is called **open**, a term without variables is called **closed**. The declarations must satisfy a number of conditions that guarantee that each term has a unique least sort, not spelled out here. The least sort of a term t is called $sort(t)$.

The **language** $L_{Sig,X}$ defined by Sig and X consists of **formulas** defined in the usual manner:

- If $P : S_1 \times \cdots \times S_n \in \mathcal{P}$ and $sort(t_i) \leq S_i$, then $P(t_1, \ldots, t_n)$ is a formula, also called an **atom**. We write $t_1 = t_2$ instead of $=(t_1, t_2)$.
- If ϕ and ψ are formulas, then so are $\phi \vee \psi$ and $\neg\phi$.
- If ϕ is a formula and $x \in X_S$, then $\forall x : S :: \phi$ is a formula.

The operators $\wedge$, $\rightarrow$ and $\exists$ and the concepts of **free** and **bound** variables are introduced in the usual manner. A **literal** is an atom or a negation of an atom. A formula is **closed** if each variable is bound by some quantifier. A **Horn clause** is a formula of the form $\phi \rightarrow \psi$ where ϕ and ψ are conjunctions of literals.

An **algebra** A for Sig consists of the following sets.

- For each $s \in \mathcal{S}$ a set S^A such that if $S_1 \leq S_2$, then $S_1^A \subseteq S_2^A$.
- For each $f : S_1 \times \cdots \times S_n \rightarrow S_0 \in \mathcal{F}$ a function $f^A : S_1^A \times \cdots \times S_n^A \rightarrow S_0^A$.
- For each $P : S_1 \times \cdots \times S_n \in \mathcal{P}$ a set $P^A \subseteq S_1^A \times \cdots \times S_n^A$.

We call S^A a **data type** and the elements of S^A **data elements**. The sorts in Sig are called **data sorts** and their terms are called **data terms**.

Given a set of variables $V \subseteq X$, an **assignment** to V is a sort-preserving function $V \rightarrow A$. (A function is sort-preserving iff it maps variable x into $sort(x)^A$.) The interpretation of a term t in A given I is written as $t^{A,I}$. Truth of formula ϕ in an algebra A under an assignment I is denoted $A, I \models \phi$. This is defined in the usual way but we give the two important clauses:

- $A, I \models x = y$ if $I(x) = I(y)$.
- $A, I \models t_1 = t_2$ if $t_1^{A,I} = t_2^{A,I}$.

An **equational specification** is a pair $Spec = ((Sig, X), E)$ where (Sig, X) is a signature with a set of variables, and E is a set of Horn clauses from $L_{Sig,X}$ containing only conditional equations, i.e. Horn clauses of which the literals are equations. Henceforth, we assume abstract data types (ADTs) are specified equationally.

Equational deduction is first-order deduction extended with the rules for dealing with equality, such as substituting equals for equals. Any set H of Horn clauses in $L_{Sig,X}$ has an initial algebra semantics. In the initial algebra semantics, two closed terms are equal precisely if they can be proven equal from H by equational deduction. Details are spelled out by Goguen and Meseguer [11].

2.2 Order-sorted first-order dynamic logic

We need first-order logic in order to represent object attributes in LUML specifications whose result types may be infinite (e.g. the integers). We use order-sorted logic in order to formalize the taxonomic hierarchy of classes in LUML class diagrams. Finally, we use dynamic logic to specify object operations in a declarative way. These are update actions on objects and we follow UML terminology in calling them operations. Adding this together, this leads us to order-sorted first-order dynamic logic. We deviate from the usual presentation of dynamic logic in that we introduce a sort Act for the actions in the language. This allows us to reason about equations such as $a(t_1, \dots, t_n) = a(t'_1, \dots, t'_n)$, where t_i and t'_i are closed terms and a is an action name.

2.2.1 Signature and language

A **dynamic logic signature** $Sig = ((\mathcal{S}, \leq), \mathcal{F}, \mathcal{P})$ is a tuple consisting of the following sets ($\uplus$ denotes disjoint union):

- $\mathcal{S} = \mathcal{S}_{data} \uplus \{Act\}$, where $\mathcal{S}_{data}$ is a set of sort names. Act is not related by $\leq$ to any other sort name.
- $((\mathcal{S}, \leq), \mathcal{F}_{Eq}, \mathcal{P}_{Eq})$ is an equational signature, referred to as Sig_{Eq}.
- $\mathcal{F} = \mathcal{F}_{Eq} \uplus \mathcal{F}_{upd}$;
- $\mathcal{P} = \mathcal{P}_{Eq} \uplus \mathcal{P}_{upd}$;
- Act only occurs in function declarations of the form $a : S_1 \times \cdots \times S_n \to Act$, where $S_i \neq Act$. All these declarations are in $\mathcal{F}_{Eq}$. We call a an **action symbol**.

The symbols in Sig_{Eq} are called **nonupdatable**, the other symbols are called **updatable**. Note that the equality sign is in $\mathcal{P}_{Eq}$ and can be applied to terms of sort Act. These terms are not updatable. Rather, they are (names of) actions.

An example dynamic logic signature follows. The lines are numbered for ease of reference.

sort names

1. PersonId, BusinessId, String, Date, Money

nonupdatable symbols

2. (Declaration of functions and predicates on String, Date, Money)
3. employ : PersonId $\times$ BusinessId $\times$ Date $\times$ Money $\to Act$
4. increase_sal : PersonId $\times$ BusinessId $\times$ Money $\times$ Money $\to Act$

updatable symbols

5. Person(PersonId, String, Date, Address)
6. Business(BusinessId, String, Address)
7. Employment(PersonId, BusinessId, Date, Money)

Line 1 introduces sort names and line 2 stands for the declaration of function and predicate names for abstract data types needed later on. Lines 3 and 4 introduce two action names. These are nonupdatable, which means that these are rigid designators that

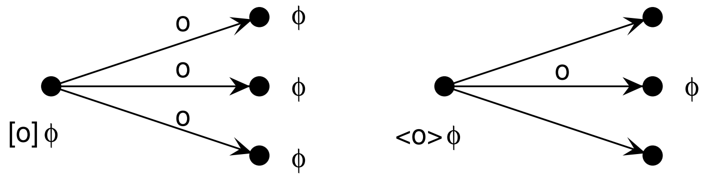

Figure 1: The semantics of $[a]\phi$ and $\langle a\rangle\phi$.

refer to the same actions in all possible states of the system. Lines 5, 6 and 7 declare updatable predicate symbols. Their extension may change in state transitions of the system. Person(p, n, b, a) means that the person identified by p has name n, birthdate b and address a. Business(b, n, a) means that the business identified by b hase name n and address a. Employment(p, b, d, s) means that person p is employed by business b, and that this relationship started at date d and involves salary a.

The set $T_{Sig,S}(X)$ of **terms** of sort S is defined in the usual manner. Terms of sort Act are called **action terms**

The language $L_{Sig,X}$ is defined in the usual manner as follows.

- If $P : S_1 \times \cdots \times S_n \in \mathcal{P}$ and $sort(t_i) \leq S_i$, then $P(t_1, \ldots, t_n)$ is a formula, also called an **atom**.
- If ϕ and ψ are formulas, then so are $\phi \vee \psi$ and $\neg\phi$.
- If ϕ is a formula and a is an action term, then $[a]\phi$ is a formula.
- If ϕ is a formula and $x \in X_S$, then $\forall x : S :: \phi$ is a formula.

The operators $\wedge$, $\rightarrow$ and $\exists$ are introduced in the usual manner. Note that $L_{Sig_{Eq},X} \subseteq L_{Sig,X}$.

We define $\langle a\rangle\phi \equiv \neg[a]\neg\phi$. The intuitive meaning of $[a]\phi$ is that after executing action a, the formula ϕ is true (figure 1). The intuitive meaning of $\langle a\rangle\phi$ is that there is a possible execution of a that terminates, and after which ϕ is true. So in particular, $\langle a\rangle true$ tells us that a can terminate from the current state. The formal semantics is defined as follows.

2.2.2 Semantics

A **labeled transition system** $LTS = (A, \Sigma, R)$ for $L_{Sig,X}$ consists of the following sets.

- A is an algebra that interprets $L_{Sig_{Eq},X}$. We call this the **domain** of LTS. The interpretation of a sort name S or nonupdatable function name f or nonupdatable predicate name P in this algebra is denoted as $S^{\mathcal{A}}$, $f^{\mathcal{A}}$ and $P^{\mathcal{A}}$, respectively.
- Σ is a set of **possible states**, where each state σ defines an interpretation of function and predicate symbols as follows:
  - If $f : S_1 \times \cdots \times S_n \rightarrow S_0 \in \mathcal{F}_{Eq}$, then $f^\sigma = f^A$.
  - If $f : S_1 \times \cdots \times S_n \rightarrow S_0 \in \mathcal{F}_{Upd}$, then f^σ is a function $S_1^A \times \cdots \times S_n^A \rightarrow S_0^A$
  - If $P : S_1 \times \cdots \times S_n \in \mathcal{P}_{Eq}$, then $P^\sigma = P^A$.

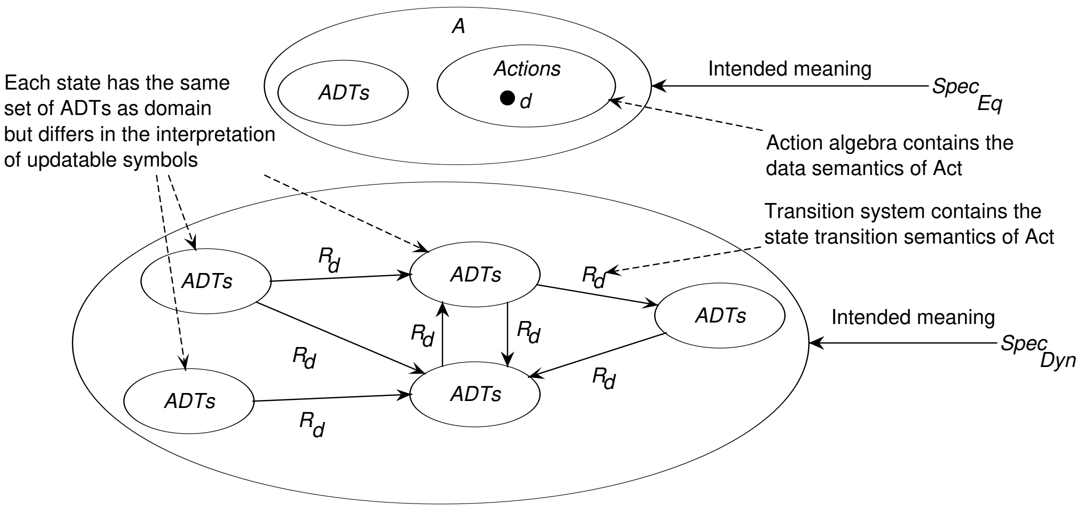

Figure 2: Two-level semantics for action terms.

- If $P : S_1 \times \cdots \times S_n \in \mathcal{P}_{Upd}$, then $P^\sigma \subseteq S_1^A \times \cdots \times S_n^A$.

- For each element $d \in Act^A$, there is a **state transition relation** $R_d \subseteq \Sigma \times \Sigma$. Instead of $R_d(\sigma_1\sigma_2)$, we write $\sigma_1 \xrightarrow{d} \sigma_2$.

If a is a closed action term, then by $\sigma_1 \xrightarrow{a} \sigma_2$ we mean $\sigma_1 \xrightarrow{a^A} \sigma_2$ and we call a the label of R.

To explain the definition, note that all states have the same domain A. This means that all closed terms in $T_{Sig_{Eq}}$ are rigid designators. Stated more simply, 3+2=5 in all states of the system. Because of this, the only difference between states is the interpretation of the updatable symbols.

Action terms are given a **two-level semantics**, illustrated in figure 2. First, each closed action term denotes an element of the ADT Act^A. So action symbols are interpreted as data elements in Act^A. For an action symbol declared as $act : S_1 \times \cdots \times S_n \to Act$, this allows us to say by equational reasoning that $act(t_1, \ldots, t_n) = act(t'_1, \ldots, t'_n)$ if $t_i = t'_i$ for $i = 1, \ldots, n$. This identifies different "calls" to the action if they have actual parameters that have the same value.

Second, each element $d \in Act^A$ defines a state transition relation R_d that defines the effect of the action on the state of the system. We call a^A the **data semantics** of a, because in this semantics, we treat action terms as any other data term. We call R_{a^A} the **state transition semantics** of a.

Truth of a formula ϕ is now defined in the standard way. In the following, I is an assignment to all free variables in the interpreted formula. Let $LTS = (A, \Sigma, R)$ be a labeled transition system and σ a state in LTS. The two main clauses of the semantics are as follows.

- $LTS, \sigma, I \models P(t_1, \ldots, t_n)$ if $(t_1^\sigma, \ldots, t_n^\sigma) \in P^\sigma$.

- $LTS, \sigma, I \models [a]\phi$ if for all $\sigma' \in \Sigma$ with $\sigma \xrightarrow{a^{A,I}} \sigma'$ we have $LTS, \sigma', I \models \phi$.

The interpretation of $\vee$, $\neg$ and $\forall$ is standard.

The two-level semantics of action terms allows us to reason about equality of action terms. This motivates the following inference rule, which is sound with respect to the semantics given above:

$$\frac{a_1 = a_2}{[a_1]\phi \leftrightarrow [a_2]\phi}.$$

Details and other inference rules are given elsewhere [36, 35].

2.2.3 Specifications

A **specification** $Spec = (Sig, \Phi)$ consists of a dynamic logic signature Sig and a finite subset $\Phi \subset L_{Sig,X}$ that contains the following classes of axioms.

- **ADT axioms** for the abstract data types. These are formulas in $L_{Sig_{Eq},X}$.
- **Static constraints** are closed non-modal formulas from $L_{Sig,X}$.
- **Effect formulas** have the form $\phi \rightarrow [a]\psi$, where ϕ and ψ are non-modal formulas and a is an Act term. We call ϕ a **sufficient precondition** of a with respect to achieving **postcondition** ψ.
- **Guard formulas** have the form $\langle a\rangle true \rightarrow \phi$, where ϕ is a non-modal formula. This says that if a is currently enabled (it can terminate from the current state), then currently ϕ is true. We call ϕ a necessary precondition for the success of a, or briefly, a **guard** for a. If the guard is false, then a cannot occur because $\neg\phi \rightarrow [a]false$. We distinguish **weak** guard formulas, which give only necessary preconditions for success, from **strong** guard formulas, which give necessary and sufficient conditions for success. Strong guard formulas have the form $\langle a\rangle true \leftrightarrow \phi$.

A specification is called **consistent** if it has a model. An example specification follows.

equational axioms

8. (axioms of the abstract data types)

static constraints

9. $\forall$ b: BusinessId $\exists$ p: PersonId , d: Date, s: Money :: Employment(p, b, d, s).
10. $\forall$ b_1, b_2: BusinessId $\exists$ p: PersonId, d: Date, s: Money::
    Employment(p, b_1, d, s) $\wedge$ Employment(p, b_2, d, s) $\rightarrow$ $b_1 = b_2$.

effect axioms

11. $\forall$ b: BusinessId $\exists$ p: PersonId, d: Date, s, s': Money::
    Employment(p, b, d, s) $\rightarrow$ [increase_sal(p, b, s')] Employment(p, b, d, s+s').
12. $\forall$ p: PersonId, b: BusinessId, d: Date, s: Money:: [employ(p, b, d, s)] Employment(p, b, d, s).

guard axioms

13. $\forall$ p: PersonId, b: BusinessId, d: Date, s: Money::
    $\langle$ employ(p, b, d, s)$\rangle true \rightarrow \neg$ Employment(p, b, d, s).
14. $\forall$ p: PersonId, b: BusinessId, s': Money::
    $\langle$ increase_sal(p, b, s')$\rangle true \rightarrow$ ($\exists$ d: Date, s: Money:: Employment(p, b, d, s)).

The lines are numbered for ease of reference. The equational axioms define the semantics of actions on the abstract data types. The constraints are the static constraints (9 and 10), effect axioms (11 and 12) and guard axioms (13 and 14) for the actions and updatable symbols. Axiom 13 says that $\neg$ Employment(p, b, d, s) is a guard for employ(p, b, d, s). The employ action can only occur if the person is not already employed. Axiom 12 is an effect axiom that says that employ(p, b, d, s), leads to employment of p by b with starting date d and salary s.

The reason for distinguishing guard from effect axioms is that they are intuitively about different things: Guard formulas state when actions can happen, effect formulas tell us what their effect is if they happen. This distinction was already made by Khosla and Maibaum [20] and turns out to lead to specifications with improved understandability.

2.3 Dynamic Step Logic

2.3.1 Steps

We now introduce steps, which are finite sets of actions executed simultaneously. We do this by introducing a data type *Step* of finite sets of action terms, and defining for each *Step* data element a state transition relation.

sorts
  $Step$
functions
  $\emptyset : Step$
  $insert : Act \times Step \to Step$
  $\_ \cup \_ : Step \times Step \to Step$
  $\_ \cap \_ : Step \times Step \to Step$
  $\neg\_ : Step \to Step$
predicates
  $\_ \in \_ : Act \times Step$
  $\_ \subseteq \_ : Step \times Step$
axioms
  (Axioms of finite sets.)

The axioms are well-known and not particularly important here. They imply that *Step* is a Boolean algebra, so that $\neg\neg s = s$, $s_1 \cup s_2 = s_2 \cup s_1$, etc.

Terms of sort *Step* are called **step terms**. We call $\cup$ **step choice**, $\cap$ **step conjunction** and $\neg$ **step negation**, respectively and $\subseteq$ is called **step inclusion**.

A specification in **dynamic step logic** (DSL) is required to include the *Step* ADT specification. Formally, we have that a dynamic step logic signature $Sig = ((\mathcal{S}, \leq), \mathcal{F}, \mathcal{P})$ is a dynamic logic signature satisfying the following conditions.

- $\mathcal{S} = \mathcal{S}_{data} \uplus \{Act, Step\}$, where $\mathcal{S}_{data}$ is a set of sort names. Neither *Act* nor *Step* are related by $\leq$ to any other sort name.
- $\mathcal{F}_{Eq}$ contains the signature of the *Step* specification. These are the only declarations in which *Step* occurs.

The language $L_{Sig,X}$ generated by a DSL signature *Sig* is defined in the same way as before, except that modal formulas now contain step terms rather than action terms:

- If ϕ is a formula and s is a step term, then $[s]\phi$ is a formula.

If a is an action term, then we write $[a]\phi$ instead of $[\{a\}]\phi$. Finally, we define the form of the specification axioms to be the same as before, with a replaced by s.

Step terms are given a two-level semantics as follows. We define a **step transition system** as an LTS that uses finite sets of actions as transition labels. As before, for each data element $d \in Act^A$, there is a state transition relation $R_d \subseteq \Sigma \times \Sigma$. We assume that each element of $Step^A$ is a finite set of zero or more elements of Act^A. This is the case for the initial algebra semantics. In the sequel, we assume that $Step^A$ is the initial algebra of the *Step* data type.

We define a state transition relation R_{d_s} for each $d_s \in Step^A$ as follows.

- $R_\emptyset = \Sigma \times \Sigma$
- For each $d_s \in Step$ and $d_a \in Act$, $R_{d_s \cup \{d_a\}} = R_{d_s} \cap R_{d_a}$.

This captures the intuition that a step s occurs in a transition whenever the actions in s occur. The empty step occurs trivially in every transition. In addition, when s occurs, then every step $s' \subseteq s$ occurs as well. Looking at it from the other end, a step s' occurs in every transition in which $s \supseteq s'$ occurs. This agrees with the definition of step semantics given earlier [25, 36], but we now give a transition system semantics, where the earlier semantics consists of step traces.

Observe that if $\{a_1, a_2\}$ occurs in a transition $\sigma_1 \to \sigma_2$, then apparently, the postconditions of a_1 and a_2 do not conflict for this transition.

We have that $R_{s_1 \cup s_2} = R_{s_1} \cap R_{s_2}$. So $s_1 \subseteq s_2$ implies $R_{s_2} \subseteq R_{s_1}$: The possible ways of doing the smaller step s_1 include the possible ways of doing the larger step s_2. But then everything that holds after s_1 also holds after s_2. This gives us the **step inclusion** rule:

$$\frac{s_1 \subseteq s_2}{[s_1]\phi \to [s_2]\phi}.$$

This implies that for example the following formulas are valid.

- $[s_1]\phi \vee [s_2]\phi \to [s_1 \cup s_2]\phi$.
- $[s_1]\phi \wedge [s_2]\phi \leftarrow [s_1 \cap s_2]\phi$.

We generalize step inclusion to **step implication** $s_1 \gg s_2$ defined as $R_{s_2} \subseteq R_{s_1}$. Obviously, if $s_1 \subseteq s_2$, then $s_1 \gg s_2$, but there is no equivalence. For example, we may define actions a_1 and a_2 such that $R_{a_2} \subseteq R_{a_1}$. We abbreviate $\{a_1\} \gg \{a_2\}$ as $a_1 \gg a_2$ and refer to this as **action implication**.

2.3.2 Minimal change semantics

This section and the following are based upon research reported recently [5]. We define the minimal LTS of a specification as the LTS in which each transition causes a minimal state change compatible with the effect axioms, and in which guards are interpreted as sufficient for the occurrence of update actions. Given a labeled transition system $LTS = (A, \Sigma, R)$ for $L_{Sig,X}$, we define the **difference** between two states $\sigma, \sigma' \in \Sigma$ as follows.
$Diff(\sigma, \sigma') =$
$\{(P, \langle d_1, \ldots, d_n\rangle) \mid P : S_1 \times \cdots \times S_n \in \mathcal{P}$ and $\langle d_1, \ldots, d_n\rangle \in P^\sigma$ and $\langle d_1, \ldots, d_n\rangle \notin P^{\sigma'}\} \cup$

$\{(P, \langle d_1, \ldots, d_n \rangle) \mid P : S_1 \times \cdots \times S_n \in \mathcal{P}$ and $\langle d_1, \ldots, d_n \rangle \notin P^\sigma$ and $\langle d_1, \ldots, d_n \rangle \in P^{\sigma'}\} \cup$
$\{(f, \langle d_1, \ldots, d_n \rangle) \mid f : S_1 \times \cdots \times S_n \to S_0 \in \mathcal{F}$ and $f^\sigma(d_1, \ldots, d_n) \neq f^{\sigma'}(d_1, \ldots, d_n)\}$.
This set just collects all the differences in interpretation of symbols between two states. Each data tuple is labeled by the symbol in whose interpretation it is different. Note that in case of overloading of symbols, we choose to count each difference only once. Had we used the entire symbol declaration as label, a data tuple would be counted once for each declaration of an overloaded symbol. However because we use *Diff* only to compare differences, this choice does not affect the following definitions. We call $\mathit{Diff}(\sigma, \sigma')$ the **change** in any transition $\sigma \to \sigma'$.

There are two ways to formalize the concept of minimal change: Ordering differences according to the subset ordering, or ordering them according to their cardinality [5, 38]. We here define the subset ordering $\sqsubseteq_s$. Let $LTS_1 = (A, \Sigma, R_1)$ and $LTS_2 = (A, \Sigma, R_2)$ be two labeled transition system for $Sig = ((\mathcal{S}, \leq), \mathcal{F}, \mathcal{P})$. Then we define $LTS_1 \sqsubseteq_s LTS_2$ iff for all step terms s, interpretations I and $\sigma \in \Sigma$ we have the following:

- If an s transition from σ is possible in LTS_1, then an s transition from σ is possible in LTS_2 with a larger or smaller change than the LTS_1 transition:

  $$\begin{array}{l}\exists \sigma' \in \Sigma \text{ with } R_{1,s^{A,I}}(\sigma, \sigma') \text{ then } \exists \sigma'' \in \Sigma \text{ with } R_{2,s^{A,I}}(\sigma, \sigma'') \\ \qquad\qquad\qquad\qquad \text{and } \mathit{Diff}(\sigma, \sigma') \substack{\subseteq \\ \supseteq} \mathit{Diff}(\sigma, \sigma'),\end{array}$$

  and

- The changes brought about by the steps in LTS_1 are not larger than the changes they bring about in LTS_2.

  $$\begin{array}{l}\forall \sigma' \in \Sigma \text{ if } R_{1,s^{A,I}}(\sigma, \sigma') \text{ then } \exists \sigma'' \in \Sigma \text{ with } R_{2,s^{A,I}}(\sigma, \sigma'') \\ \qquad\qquad\qquad\qquad \text{and } \ Diff(\sigma, \sigma') \subseteq Diff(\sigma, \sigma'').\end{array}$$

We only compare LTSs with the same domain A and state space Σ. The choice for the same algebra is motivated by the fact that we want to order the models of one specification. The choice to require the same state space is motivated by the fact that models with different state spaces would not be compatible anyway. Dropping this requirement makes the definition harder to understand but does not affect our definition of intended model below. The first condition above means that the subset ordering does not deal with the possibility of the occurrence of steps; an ordering based upon possibility is defined below. It only compares LTSs in which the transition relations allow the the occurrence of the same steps in the same states.

Suppose two actions a_1 and a_2 have effect axioms $[a_1]P$ and $[a_2]Q$ and in the current system state σ, we have $\sigma \models \neg P \wedge \neg Q$. Then the minimal change interpretation of step $\{a_1\}$ leads to a state with $P \wedge \neg Q$ and the minimal change interpretation of $\{a_2\}$ leads to a state with $\neg P \wedge Q$. The minimal change interpretation of $\{a_1, a_2\}$ leads to a state with $P \wedge Q$. So the **free** interpretation of a step is: "it achieves its postconditions and anything else may change". The **intended** interpretation, defined below, is "it achieves its postconditions and nothing else changes".

It is easy to see that $\sqsubseteq_s$ is a reflexive and transitive relation on LTSs.

Proposition *If a specification has a model in which each action has a non-empty transition relation, then the specification has a $\sqsubseteq_s$ minimal model [5].*

So $\sqsubseteq_s$ allows us to define the models that give a minimal-change interpretation to the effect axioms.

2.4 Maximal reachability semantics

We now define the maximal reachability ordering, which allows us to define models that give a maximally nondeterministic interpretation to the guard axioms that is compatible with the integrity constraints. Let $LTS_1 = (A, \Sigma_1, R_1)$ and $LTS_2 = (A, \Sigma_2, R_2)$ be two labeled transition system for $Sig = ((\mathcal{S}, \leq), \mathcal{F}, \mathcal{P})$. Then we define $LTS_1 \sqsubseteq_r LTS_2$ iff

- $\Sigma_1 \subseteq \Sigma_2$ and
- $R_{1,a^{A,I}} \subseteq R_{2,a^{A,I}}$ for all action terms a.

An LTS that is maximal under the reachability ordering contains maximal nondeterminism compatible with the guard axioms. Each weak guard axiom $\langle a \rangle true \to \phi$ gives a necessary condition for the possibility of a. In a $\sqsubseteq_r$ maximal LTS, the conjunction of all such necessary preconditions is jointly sufficient for the possibility of a. It is easy to see that $\sqsubseteq_r$ is a partial order on LTSs.
Proposition *A consistent specification has a $\sqsubseteq_r$ maximal model [5].*

2.4.1 Intended semantics

Let $Spec = (Sig, \Phi)$ be a dynamic logic specification. We assume in the following that a semantics A for its equational part has been chosen, such as for example the initial algebra. We define the min-max model of $Spec$ as the unique $\sqsubseteq_r$ maximal element of the set of $\sqsubseteq_s$ minimal models of $Spec$. $\sqsubseteq_s$ minimalization chooses the models in which the updates cause minimal changes to the updatable symbols compatible with the effect axioms. Among those models, $\sqsubseteq_r$ maximalization then chooses the models that only contain, from every state, the largest possible set of step transitions that are compatible with the guard axioms. Now, the **intended model** of $Spec$ is obtained from the min-max model by deleting, for each state σ in the min-max model, all transitions $\sigma \xrightarrow{s} \sigma'$ for which there is a transition $\sigma \xrightarrow{s'} \sigma''$ with $s \subset s'$. So in the intended model, only the largest possible steps are taken (consisting of as many actions as possible). Due to $\sqsubseteq_r$ maximalization, there are as many of these as possible, and due to $\sqsubseteq_s$ minimalization, these change as little as possible. Note that the intended model may be nondeterministic.
Proposition *A specification with a model in which each action has a non-empty transition relation has a unique intended model.*

3 Lightweight Class Diagrams

3.1 Syntax

We choose UML 1.1 static structure diagrams consisting of the following syntactic constructs [27]: Class boxes contain, from the top down, a class name, attribute declarations,

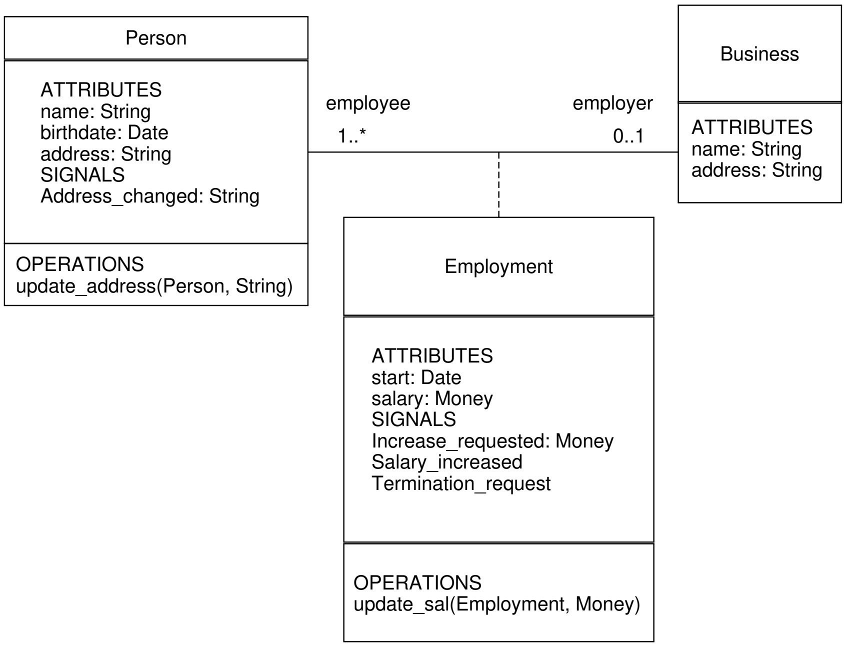

Figure 3: An example class diagram.

signal declarations and operation declarations. Signals are predicates that formalize messages sent among objects and between the system and its external environment. Associations have names, multiplicities and optional role names. We allow association classes and n-ary associations and generalizations. We refer to the resulting small subset as **lightweight class diagrams** (LCDs). Figure 3 contains an example LCD. Signals are not shown separately in UML 1.1, but the UML syntax definition is flexible enough to add a "signal" property to the diagram syntax.

3.2 Semantics

This section summarizes the formalization of class diagrams spelled out elsewhere [33, 34, 35]. We give a semantics to LCDs by defining, for each syntactically legal LCD, a dynamic logic specification. The formal meaning of the LCD is then the intended transition system semantics of the LCD. The core of the formalization concerns the representation of identity and existence of objects and associations, and of the taxonomic structure of classes. To save space, we omit the formalization of existence and taxonomic structures in this paper.

The translation begins by providing equational specifications for the data types used in the LCD, such as String, Date and Money. In our semantics, we assume the initial algebra as intended semantics for these specifications.

For each object class with name C we declare a sort name C, called the **identifier sort** of the class. The data elements in these sorts are called **identifiers**. The simplest way to populate these sorts is to declare for each object identifier sort C a constant $i_C : C$ that is different for different classes, and a function $next : C \to C$. This gives us infinitely many object identifiers for each object class. This rather crude way to populate the identifier sorts must be refined to deal with taxonomic structures appropriately, but we omit this here.

Associations are formalized as Cartesian products. The formal specification of the Employment association follows.

sorts

1. Employment

nonupdatable symbols

2. employee: Employment $\rightarrow$ Person
3. employer: Employment $\rightarrow$ Business
4. $\langle \_, \_ \rangle$: Business $\times$ Person $\rightarrow$ Employment
5. update_sal : Employment $\times$ Money $\rightarrow$ Act

updatable symbols

6. start: Employment $\rightarrow$ Date
7. salary : Employment $\rightarrow$ Money
8. Increase_requested: Employment $\times$ Money
9. Salary_increased : Employment
10. Termination_request: Employment

equational axioms

11. (Axioms for the abstract data types)
12. $\forall$ p : Person, b: Business :: employee($\langle$ b, p $\rangle$) = p.
13. $\forall$ p : Person, b: Business :: employer($\langle$ b, p $\rangle$) = b.
14. $\forall$ e: Employment :: e = $\langle$ employer(e),employee(e)$\rangle$.

constraints

15. $\forall$ b: Business :: $\exists_1$ p: Person $\exists$ e : Employment :: employee(e) = p $\wedge$ employer(e) = b.
16. $\forall$ p: Person :: $\exists_{\leq 1}$ b: Business , $\exists$ e : Employment :: employee(e) = p $\wedge$ employer(e) = b.
18. $\forall$ e : Employment, $\forall$ s, s': Money:: salary(e) = s $\rightarrow$ [update_sal(e, s')] salary(e) = s+s'.

For each association class name R we declare an association identifier sort name R. If R relates classes $C_1, \ldots, C_n$, there is a constructor function $\langle \_, \ldots, \_ \rangle$ for the Cartesian product. The order of the components of the product must be decided upon, e.g. by alphabetical ordering of class names or some other means. There are projection functions that pick elements of a tuple, as defined by equational axioms. We use the role names as names of the projection functions. If there are no role names in the LCD, some unique name should be used.

Associations differ from objects in that their identifiers are tuples of identifiers of objects. Other than the structure of their identifiers, there is no difference, in the formalization, between objects and association. Both can have attributes, signals and operations, and are subject to constraints.

Multiplicity constraints are formalized using generalized quantifiers such as $\exists_n$ and $\exists_{\leq n}$ for n a natural number. These can be expanded to first-order formulas using only ordinary quantification.

Attributes are represented as unary updatable (function or predicate) symbols on object identifier sorts with the type of the attribute as result type of the function. The result sort s may be any abstract data type. In particular, it can be an identifier sort. This allows the representation of "nested objects". Boolean attributes can be formalized as unary predicates on identifier sorts.

All signals are formalized as predicates whose first argument type is the class and whose other arguments are the types listed in the class diagram for the signal. For example, the declaration Increase_requested : Money in the class box is formalized in line 8 of the Employment specification.

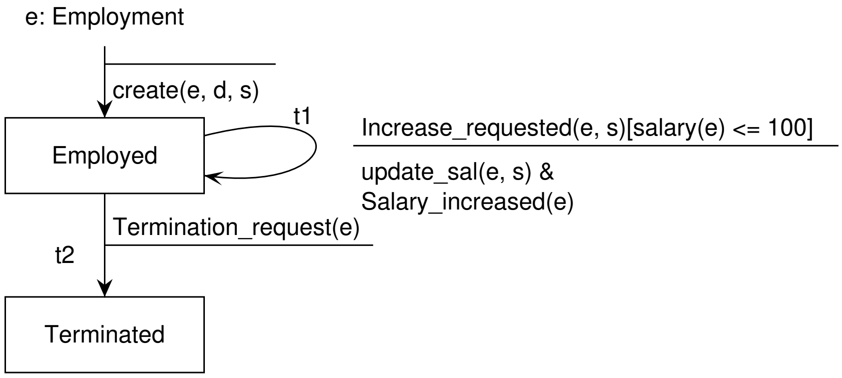

Figure 4: An example STD for the Employment class.

Operations are formalized as actions in dynamic step logic. The interpretation of operations is defined by our minimal change, maximal step semantics. Each object class specification contains a possibly empty set of effect and guard axioms for each operation. In the intended LTS, the maximal step transitions occur that are compatible with the guard axioms and static constraints; and these go to states that differ minimally from the current state, compatible with the guard axioms and static constraints. As we see in the next section, state diagrams are a convenient way of specifying a large number of effect and guard axioms in a compact and intuitive way.

4 Extended Mealy Machines

This section extends earlier research.

4.1 Syntax

A Mealy machine is a finite-state machine that responds to input events by producing output and moving to a next state [17]. An extended Mealy machine is a Mealy machine with local variables, that can be tested before the machine moves to a next state, and that can be updated when the machine moves to a next state [1]. Instead of extended Mealy machines, we here define diagrams that represent such machines.

An **extended Mealy state diagram** (EMD) is a labeled and rooted directed graph in which the nodes represent object states and the edges object state transitions. The edges are labeled by labels of the form $t : e[g]/a$, where t is a name for the object transition that is unique for all transitions in this object, e represents **event** that triggers t, g represents the guard of t, and a represents the **action** performed when the transition is executed. This term “action” is common in Mealy machine theory, but must be distinguished from the same term as used in dynamic logic. It will become clear in a moment why. Henceforth, unless otherwise stated, the term “action” refers to Mealy machine actions.

The syntax of the arrow labels is described below. The nodes are labeled too. Node labels must be unique within one STD. The root is pointed at by a small arrow and represents the initial state. To distinguish transitions (states) in an EMD from transitions in an LTS, we refer to the first as **object transitions (states)** and to the second as

global transitions (states).

For each LCD, there is a collection of zero or more EMDs. Each such EMD belongs to exactly one (object or association) class in the class diagram. The EMD is named by the class name and corresponds to an instance (object or association) of this class. To formalize this, we add the declaration of an identifier variable to the label of the root arrow. (This variable is often called *self* in the literature.) The transition label must have the form $t : e[g]/a$.

- t is the name of the transition.
- The event label e of a transition is a signal declared for this class. This is called the **input signal** of the transition. The parameters of the input signal are called the **input parameters** of the transition.
- Let i be the identifier variable of the EMD, P be the set of input parameters of the transition and let Sig be the signature defined by the entire class diagram. X is the set of all variables as before. Then the guard g is a formula in $L_{Sig,X}$, whose free variables are in $\{i\} \cup P$. We thus allow a guard to refer to the entire state of the model. Its free variables must be bound to i or to the input parameters.
- The action label a consists of a set of operation terms and signal atoms from the class signature of this EMD. The parameters of these terms and atoms must all occur in $\{i\} \cup P$. The signals are called the **output signals** of the transition and their parameters **output parameters**.

4.2 Semantics

The formalization of the EMD in figure 4 consists of the extension of the class specification with a number of declarations and constraints. In the formalization, there are two kinds of dynamic logic actions, object operations and transitions. Note that the Mealy machine actions and events already have been formalized as predicates.

The semantics of the EMD in figure 4 is the following extension of the Employment specification. We explain this line by line below.

nonupdatable symbols

E1. t1: Employment $\times$ Money $\rightarrow$ Act

E2. t2: Employment $\rightarrow$ Act

updatable symbols

E3. Employed : Employment

E4. Terminated: Employment

constraints

E5. $\forall$ e: Employment :: Employed(e) $\leftrightarrow$ $\neg$ Terminated(e).

E6. $\forall$ e: Employment, s: Money ::
$\langle$ t1(e, s) $\rangle$$true$ $\rightarrow$ Employed(e) $\wedge$ Increase_requested(e, s) $\wedge$ salary(e) $\leq$ 100.

E7. $\forall$ e: Employment :: $\langle$ t2(e) $\rangle$$true$ $\rightarrow$ Employed(e) $\wedge$ Termination_requested(e).

E8. $\forall$ e: Employment , s: Money :: [t1(e, s)] Employed(e) $\wedge$ Salary_increased(e).

E9. $\forall$ e: Employment :: [t2(e)] Terminated(e).

E10. $\forall$ e: Employment , s: Money :: [$\neg$ t1(e, s)] $\neg$ Increase_requested(e, s).

E11. $\forall$ e: Employment :: [$\neg$ t2(e)] $\neg$ Termination_request(e).

E12. ∀ e: Employment , s: Money :: t1(e, s) $\gg$ update_sal(e, s).
E13. ∀ e: Employment , s: Money :: [{t1(e, s), t2(e)}]$false$.

Remarks:

- E1, E2: we declare each object transition as an *Act* symbol with the same arguments as the triggering event.
- E3, E4: Each state name is declared as a unary predicate symbol with the class as argument type. We call these the **state predicates** of the class. When an EMD is in a state, its state predicate is true and all other state predicates of this EMD are false.
- E5: In addition, all state predicates are defined to be mutually exclusive. We call this the **or-state** axioms.
- E6, E7: Each object transition is defined to have its source state, input signal and object transition guard as total guard. In the intended semantics, this enables this object transition. More precisely, it contributes to the enabling of all maximal steps containing this object transition.
- E8, E9: Each object transition is defined to have as postcondition its target state and all output signals. In the intended model, t2 leads to a global state in which ¬ Employed(e) holds, but t1 does not. This is a derived update, not stated explicitly in the axioms, called **ramification** elsewhere. In addition, our minimal change semantics implies that as many things as possible do not change. For example, the start_date of an employment association is not changed by t1 or t2.
- E10, E11: For each input signal $P(x_1, \ldots, x_n)$ we determine the finite set T_P of transitions that it triggers in the entire specification. So the transitions in T_P may appear in different EMDs. We then add the **signal frame formula**

  $$\forall x_1 : S_1, \ldots, x_n : S_n :: [\neg(\bigvee_{t \in T} t)]\neg P(x_1, \ldots, x_n).$$

  This says that any step that does not contain a transition from T_P sets the signal to $false$. So an input signal generated (set to true) in one step s remains available during the next step s' only, and is then set to false again unless there is an action in s' that generates it. The signal frame formulas are similar to Reiter's successor state axioms [29] and to the locality principle of Ryan, Fiadeiro and Maibaum [30].
- E12: For each operation o and each transition t in any EMS that executes o as part of its action, we add the axiom $t \gg o$. In other words, it is impossible that t occurs without o occurring in the same step. We assume that the effect of operations has already been defined in the class diagram specification (line 18 of the Employment specification).
- E13: For each state of each EMD from which two or more object transition arrows $t_1, \ldots, t_n$ depart, that all go to the same destination state, we add an axiom of the

form $\forall V[\{t_1, \ldots, t_n\}]false$. This prevents two or more of these object transitions to be taken together in cases where their object transition guards evaluate to true. In the example, E13 is redundant and has been added for illustration only.

This semantics formalizes the intended meaning of essential decompositions discussed in section 1.2. Note that all axioms have the form required for specification axioms defined earlier, by the simple fact that every finite step can be rewritten (in the step algebra) to the form $\{a_1, \ldots, a_n\}$.

The resulting intended model is an LTS in which each state is characterized by (1) the attribute values of objects and (2) the signals to be responded to in the next step. From a state σ_1, there are transitions $\sigma_1 \xrightarrow{s} \sigma_2$ in which s is the maximal step that contains all object transitions triggered by the signals in σ_1, as well as the operations called by those transitions. These operations change the attribute values minimally. All signals are set to $false$ in a step, except those that are generated by object transitions in the step.

We should hasten to add that the analyst is not expected to type in a specification like this. The point of the EMD notation is precisely to avoid having to write such a specification. Also, it is possible to embellish the textual version of the EMD with syntactic sugar that hides most of the textual specification behind defaults and abbreviations.

5 Discussion and Conclusions

Our semantics defines an intended model for ultra-LUML specifications but it does not yet make these models executable. It is not our intention to use the complete axiomatic specification of the intended model as a basis for executing the specification. Rather, we intend to use the transition system as a basis for an execution semantics by defining a Statemate-like execution algorithm. An execution of this algorithm would correspond to a path through the transition system. To simulate the behavior of the specified system, we should introduce **external** signals, that represent inputs from the environment. In a Statemate-like single-step semantics, in each state of the system there may be a non-empty set of external signals, that together with internal signals generated in the previous step, trigger global system transitions. In a superstep semantics, all responses to an external event must be produced before the next event can be responded to (synchrony hypothesis). This means that external signals can only be present in states in which there are no internal signals. A superstep then consists of the sequence of steps that jointly constitute the response to a set of external signals.

Current work includes working out all the details of dynamic step logic as proposed here. Our transition system semantics also opens possibilities for the application of model checking, because property specification language such as temporal logics are all interpreted in transition structures. Another interesting avenue of further research is the extension of the version of dynamic logic with the object constraint language (OCL) of the UML. Yet another avenue is the addition of real-time constructs that would allow us to specify timeouts and scheduled events as in Statemate. Finally, as other formalizations, our formalization allows the definition of coherence rules for models consisting of ultra-LCDs and EMDs. We intend to specify these rules in the near future.

To extend ultra-LUML to LUML, taxonomic structures, existence, creation events and initialization actions must be added [33, 34, 35, 36]. In order to add intra-object parallelism, we will consider allowing the allocation of more than one EMD to one object class, thus creating shared-memory machines. Addition of state hierarchies such as in statecharts should pose no fundamental problem, but does make the exact definition of enabled transitions and maximal steps considerably more complex. Another interesting extension is the addition of subsystem hierarchies as a complexity-reducing technique.

Acknowledgements

Thanks are due to Pascal van Eck, who provided detailed and constructive comments on an earlier version of the paper, and to the anonymous referees, who pointed out a few omissions.

References

[1] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from protocol Specification*. Prentice-Hall, 1991.

[2] G. Berry and I. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. Brookes and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448, 1985. Lecture Notes in Computer Science 197.

[3] R.H. Bourdeau and B.H.C. Cheng. A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, October 1995.

[4] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the Unified Modeling Language. In *Proceedings of ECOOP'97*. Springer Verlag, LNCS, 1997.

[5] J.M. Broersen and R.J. Wieringa. Minimal semantics for action specifications in first-order dynamic logic. Technical Report IR-439, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, November 1997. ftp://ftp.cs.vu.nl/pub/roelw/97-ActionSpecFol.ps.Z ftp://ftp.cs.vu.nl/pub/roelw/97-ActionSpecFol.ps.Z.

[6] D. Coleman. Fusion with use cases: Extending Fusion for requirements modeling. Available through URL: http://www.hpl.hp.com/fusion/index.html, 1996.

[7] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The FUSION Method*. Prentice-Hall, 1994.

[8] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice-Hall, 1994.

[9] J. Fiadeiro, C. Sernadas, T. Maibaum, and A. Sernadas. Describing and structuring objects for conceptual schema development. In P. Loucopoulos and R. Zicari, editors, *Conceptual Modelling, Databases and CASE*. Wiley, 1993.

[10] R.B. France, J.-M. Bruel, and M.M. Larrondo-Petrie. An integrated object-oriented and formal modeling environment. *Journal of Object-Oriented Programming*, 10(7):25–34, November/December 1997.

[11] J.A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[12] F. Golshani, T.S.E. Maibaum, and M.R. Sadler. A modal system of algebras for database specification and query/update support. In *Proceedings of the Nineth International Conference on Very Large Databases*, pages 331–359, 1983.

[13] Radu Grosu, Cornel Klein, Bernhard Rumpe, and Manfred Broy. State transition diagrams. Technical Report TUM-I9630, Technische Univerität München, 1996.

[14] D. Harel and E. Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, pages 246–257. IEEE Press, 1996.

[15] D. Harel and E. Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, July 1997.

[16] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[17] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[18] i-Logix. The Semantics of Statecharts. Technical report, i-Logix Inc., 22 Third Avenue, Burlington, Mass. 01803, U.S.A., January 1991.

[19] P. Jeremaes, S. Khosla, and T.S.E. Maibaum. A modal (action) logic for requirements specification. In D. Barnes and P. Brown, editors, *Software Engineering 86*, pages 278–294. Peter Peregrinus Ltd., 1986.

[20] S. Khosla and T.S.E. Maibaum. The prescription and description of state based systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, pages 243–294. Springer, 1987. Lecture Notes in Computer Science 398.

[21] S. Khosla, T.S.E. Maibaum, and M. Sadler. Database specification. In T.B. Jr. Steel and R. Meersman, editors, *Database Semantics (DS-1)*, pages 141–158. North-Holland, 1986.

[22] N. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[23] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[24] S.M. McMenamin and J.F. Palmer. *Essential Systems Analysis.* Yourdon Press/Prentice Hall, 1984.

[25] J.-J.Ch. Meyer. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. *Notre Dame Journal of Formal Logic*, 29:109–136, 1988.

[26] A. Pnueli and M. Shalev. What is in a step: on the semantics of statecharts. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 244–264. Springer, 1991. Lecture Notes in Computer Science 526.

[27] Rational. *Unified Modeling Language: Notation Guide, Version 1.1.* Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, 1 September 1997. URL http://www.rational.com/uml/1.1/.

[28] Rational. *Unified Modeling Language: Semantics, Version 1.1.* Rational Software Corporation, 1 September 1997. URL http://www.rational.com/uml/1.1/.

[29] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and the Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[30] M. Ryan, J. Fiadeiro, and T. Maibaum. Sharing actions and attributes in modal action logic. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 569–593. Springer, 1991. Lecture Notes in Computer Science 526.

[31] E.Y. Wang, H.A. Richter, and B.H.C. Cheng. Formalizing and integrating the dynamic model within omt. In *IEEE Proceedings of International Conference on Software Engineering (ICSE97)*, 1997.

[32] R.J. Wieringa. Equational specification of dynamic objects. In R.A. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design, and Construction (DS-4)*, pages 415–438. North-Holland, 1991. ftp://ftp.cs.vu.nl:/pub/roelw/91-EquationalSpecOfOBjects.ps.Z ftp://ftp.cs.vu.nl:/pub/roelw/91-EquationalSpecOfOBjects.ps.Z.

[33] R.J. Wieringa. A formalization of objects using equational dynamic logic. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *2nd International Conference on Deductive and Object-Oriented Databases (DOOD'91)*, pages 431–452. Springer, 1991. Lecture Notes in Computer Science 566. ftp://ftp.cs.vu.nl/pub/roelw/91-DynamicObjects.ps.Z. ftp://ftp.cs.vu.nl/pub/roelw/91-DynamicObjects.ps.Z.

[34] R.J. Wieringa, W. de Jonge, and P.A. Spruit. Roles and dynamic subclasses: a modal logic approach. In M. Tokoro and R. Pareschi, editors, *Object-Oriented Programming, 8th European Conference (ECOOP'94)*, pages 32–59. Springer, 1994. Lecture Notes in Computer Science 821. ftp://ftp.cs.vu.nl/pub/roelw/93-Roles.ps.Z. ftp://ftp.cs.vu.nl/pub/roelw/93-Roles.ps.Z Revised and extended version appears in [35].

[35] R.J. Wieringa, W. de Jonge, and P.A. Spruit. Using dynamic classes and role classes to model object migration. *Theory and Practice of Object Systems*, 1(1):61–83, 1995. ftp://ftp.cs.vu.nl/pub/roelw/95-DynamicClassesAndRoles.ps.Z. ftp://ftp.cs.vu.nl/pub/roelw/95-DynamicClassesAndRoles.ps.Z.

[36] R.J. Wieringa and J.-J.Ch. Meyer. Actors, actions, and initiative in normative system specification. *Annals of Mathematics and Artificial Intelligence*, 7:289–346, 1993. ftp://ftp.cs.vu.nl/pub/roelw/93-DeonticActors.ps.Z. ftp://ftp.cs.vu.nl/pub/roelw/93-DeonticActors.ps.Z.

[37] R.J. Wieringa and R.P. Van De Riet. Algebraic specification of object dynamics in knowledge base domains. In R.A. Meersman, Zhongshi Shi, and Chen-Ho Kung, editors, *Artificial Intelligence in Databases and Information Systems (DS-3)*, pages 411–436. North-Holland, 1990.

[38] M. Winslett. *Updating Logical Databases.* Cambridge University Press, 1990.