A Comparison of the Notations Used in the Shlaer-Mellor Method and in TCM[1]

R.J. Wieringa[2]
G. Saake[3]

October 9, 1995

[1]Partially supported by ESPRIT BRA WG 6071 IS-CORE and by ESRIT BRA WG 8319 ModelAge.
[2]Faculty of Mathematics and Computer Science, Free University, De Boelelaan 1081a, 1081HV Amsterdam. Email: roelw@cs.vu.nl. http://www.cs.vu.nl/~roelw.
[3]Institut für Technische Informationssysteme, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, Universitätsplatz 2, D - 39106 Magdeburg, Germany. Email: saake@iti.cs.tu-magdeburg.de.

Abstract

This report compares two notation systems for requirements modeling, the notations used in the Shlaer-Mellor method for object-oriented analysis and the notations uses in TCM (Toolkit for Conceptual Modeling). The notations used in the Shlaer-Mellor method are semi-formal, i.e. they consist of diagrams annotated by natural language text. The notations used in TCM are semi-formal, as in the Shlaer-Mellor notation, but there is also a formal part. The formal part of a TCM specification is written down in LCM (Language for Conceptual Modeling), a language based on order-sorted dynamic logic. The formal and semi-formal parts of a TCM specification supplement each other and each can be used without using the other. Because the semi-formal and formal notations in TCM are precisely related, the semi-formal notations have unambiguous definitions, and the formal notations have simple and clear diagram representations.

The report analyzes the notations used for information model, state model, process model and communication model used in the Shlaer-Mellor method and shows how these relate to the notations used for the class model, life cycle model and communication model in TCM. It is shown that the notations of the Shlaer-Mellor method contain ambiguities and redundancies, that are resolved when we transform these notations into TCM notations. However, some of these redundancies provide useful information and TCM is extended with a simplified form of some of the Shlaer-Mellor notations.

The report repeatedly refers to figures from [39]. For copyright reasons, these figures are not reproduced here. The reader is expected to have a copy of this book closeby when reading this report.

Contents

Chapter 1

Introduction

Object-oriented research and practice has moved from a focus on programming systems to a concern with all stages of the development process, from requirements analysis to design and programming. Thus, object-oriented development methods like OMT [32], OOA [37, 39], Coad/Yourdon [5] and Objectory [25] include all stages of the software development process from requirements analysis to software implementation. Much of the research in object-oriented analysis, design and programming is concerned with the practical affairs of building systems, but in parallel to practical orientation, theoretical research on formal specification and formal semantics has arisen. For example, there is a large body of research on the foundations of object-oriented languages such as polymorphism and inheritance and there and on formal specification of object-oriented analysis models. This has resulted in a number of formally grounded object-oriented analysis methods [6, 3, 10, 22, 27, 36, 48]. This report is concerned with the formal underpinning of an object-oriented analysis method, the Shlaer/Mellor method [37, 39].

The purpose of our research is to mutually enhance formal and semi-formal notations for conceptual models of required system behavior. By **semi-formal notations** we mean notations consisting of diagrams and natural language and by **formal notations** we mean notations based on mathematics and formal logic. Formal notation systems always have a set of precisely defined manipulation rules for the elements of the notation, and these rules always have a precisely defined meaning. Semi-formal notations typically employ visual means to enhance the understanding, but do not have a set of precisely defined sound manipulation rules.

Much has been written already on the advantages and disadvantages of formal specification of software system requirements: Formal specifications enhance clarity and reduce ambiguity of the specification, and they allow us to prove properties of a system and to verify correctness of an implementation [1, 8, 9, 14]. Here, we want to point out the advantages of combining a formal and a semi-formal notation system for conceptual modeling.

- Formal techniques tend to be developed with ample logical or mathematical underpinning of the notation systems but with little methodological support. By contrast, semi-formal notations typically come with methodological advice for finding and validation models written in those notations. By coupling a notation used in practice with a formal notations, we can make methodological advice available on how to use the formal notation.

- Since formal notations tend to have a forbidding appearance to nonspecialists, coupling them with semi-formal notations allows us to combine the communicative aspect of semi-formal notations with the rigor of formal notations.

- Coupling a formal with a semi-formal notation forces us to make a totally explicit distinction between specification and implementation in both notations. An semi-formal representation of system requirements can create the illusion of completeness, and therefore may leave modeling decisions unmade. Since these decisions must be made at some point during development, they will be made at a later point, during design or implementation. However, at that point, these decisions are not validated with the user and they are made on the basis of design or implementation considerations rather. A formal notation technique does not give the developer the freedom to leave loose ends in the specification. Formalization of an informal notation technique allows us to tidy up loose ends in the notation and to force the developer to make all modeling decisions in a way that is visible to the customer.

- Linking the concepts used in a semi-formal notation with a formal notation forces us to define the semi-formal concepts unambiguously, remove redundancy and eliminate mutually inconsistent parts. We believe that part of the problem of the current generation of semi-formal notations (including most object-oriented notations) is that they are ambiguous and, sometimes, contain redundant or mutually inconsistent parts. This makes them more difficult to use than they should be. Disambiguation and elimination of redundancy and of inconsistency makes semi-formal notations easier to use.

- Nowadays, formal specification languages come with extensive tool support to prove properties from the specification, to execute specifications, and to check the correctness of implementations. Coupling a semi-formal notation with a formal specification language allows us to use these techniques to check the quality of the specification or implementation.

Our research platform for combining formal and semi-formal notations is LCM/TCM. LCM (Language for Conceptual Modeling) [13] is a language based on order-sorted dynamic logic with equality, that can be used for the specification of object-oriented conceptual models of system behavior. TCM (Toolkit for Conceptual Modeling), is a set of methods and techniques that can be used to write a semi-formal and/or formal specification of a conceptual model.[1] TCM is a toolkit, not a method. It provides a number of well-defined semi-formal and formal notations and a set of heuristics for using these notations. The formal notation that can be used in TCM is LCM; however, the intention is that other formal notations, such as Oblog [7], FOOPS [16], and `TROLL` [26], can be used as alternatives, provided the link between the chosen formal notation and the semi-formal notations of TCM is defined.

TCM has evolved from a detailed study of existing methods for structure requirements modeling, viz. the ER notation, NIAM, Structured Analysis and Jackson System Development [45]. The aim of current research is to evaluate and, where necessary, extend the toolkit with elements of real-time and object-oriented methods [43]. The procedure each time is the same:

- Investigate whether models represented in notation X can be represented in the notations available in TCM. Because the two representations use different notations, strict equivalence (in the sense that two formal specification can be equivalent) cannot be proven, but it has turned out that it is always possible to show an intuitively convincing semi-formal equivalence.

- If some part of the representation in notation X cannot be represented in TCM, extend TCM with a notation to represent this part, subject to the following constraints:
  - The added notation should resemble as much as possible some existing and widely used notation and

[1] TCM is a further development of MCM (Method for Conceptual Modeling) [44]. The motivation for this change in name is that MCM is built from elements taken from existing methods, possibly after simplifying these elements. It is not intended to be a method but a toolkit of methods and techniques that can be used as needed by the development situation. The tools and techniques come with directions for use that, if they are followed, are guaranteed to deliver mutually consistent results. This intention is better expressed by the name Toolkit for Conceptual Modeling.

- the added notation should have a clear and unambiguous link to the other notations already in TCM.

There are a number of advantages of this way of working.

- It is a one–off effort, resulting in a description of the way in which the constructs of a semi-formal notation can be formally specified, and of the way in which the constructs of the formal notation can be semi-formally represented. Even if we never use the formal notation in TCM, we benefit from the clarity, unambiguity, nonredundancy and mutual consistency of the semi-formal notations in TCM.

- Developers used to structured analysis and entity-relationship modeling will find it easier to use formal notations if there is a clear indication of the connection of these notations with the methods that they know.

- Assuming that notation X is used in practice, we can identify areas not covered by the formal notation but useful in practice. This gives direction to future research in formal notations.

In previous work, we have analyzed the OMT method using `TROLL` as means of analysis [27, 47]. In this report, we continue our research by analyzing the notations used in the Shlaer-Mellor method of object-oriented analysis (OOA) [37, 39]. One of the reasons for this choice is that the Shlaer-Mellor method is, next to OMT, one of the more popular object-oriented methods. In addition, it has close relationships with Structured Analysis for Real Time Systems [41, 42, 29] and is therefore a good representative of a method that can bridge the gap between classical structured methods and modern object-oriented methods.

In chapter 2, we survey the deliverables of the Shlaer-Mellor method. In the rest of the report, we focus on three deliverables, the *information model* (discussed in chapter 3), the *state model* (discussed in chapter 4), the *process model* (discussed in chapter 5) and the *object communication model* (discussed in chapter 6). These analyses result in the identification of some ambiguities in the Shlaer-Mellor notation and in some extensions of TCM. Chapter 7 summarizes the results and indicates directions for further research.

Chapter 2

Deliverables of the Shlaer-Mellor method

The Shlaer-Mellor method is a method for problem analysis in the development of software systems. The problem is modeled as a set of interacting objects. For example, when the method is used for the specification of software or system requirements, then the software or the entire system is modeled as a set of interacting objects.

To manage the complexity of large systems, the entire system is modeled at four levels of aggregation: system, domains, subsystems, and objects (figure 2.1). The deliverables are structured into corresponding levels, shown in figure 2.2.

A **domain** is a distinct subject matter, inhabited by its own set of objects that behave according to a closely related set of rules and policies. For example, the *application domain* of a system is the subject matter of the system from the user's point of view. A system may contain several *service domains* that contain system utilities. Examples of service domains are a user interface domain, a data archiving domain, and a process input/output domain. Coupling between domains should be low and cohesion within one domain should be high. An object is member of exactly one domain, and each domain has its own namespace. There are existence-dependencies between objects that inhabit the same domains, but not between objects that inhabit different domains. Rules and policies typically apply to one domain and not across different domains. Domains communicate through **bridges**, which are connections between events in different domains, or between attributes of objects in different domains. A bridge is a binary link between domains in which one domain plays the role of client and the other of server.

The system-level deliverables are:

- A **domain chart** that represents the entire system as a set of domains connected by bridges.

```
System
   |
Domain
   |
Subsystem
   |
Object
```

Figure 2.1: Levels of aggregation in a model delivered by the Shlaer-Mellor method.

- For each system:
  - Domain chart
  - Project matrix
- For each domain:
  - Subsystem relationship model
  - Subsystem access model
  - Subsystem communication model
- For each subsystem:
  - Information model
  - State model (state transition diagrams, state transition tables, event list)
  - Process model (ADFDs, object access models, state process table, process specifications)
  - Object communication model

Figure 2.2: Deliverables of the Shlaer-Mellor method.

- A **project matrix** lists all domains and their subsystems in the horizontal dimension, and the three subsystem views (information, state, process) along the vertical dimension. The project matrix is a tool for management of the development process. The cells of the matrix represent tasks to be accomplished in the project, and can be used to record resources allocated to each task, work products delivered, etc.

Each domain is partitioned into one or more **subsystems** such that each object exists in exactly one subsystem and never moves between subsystems. For each domain, three kinds of relations between subsystems are represented.

- In the **subsystem relationship model**, each subsystem connection represents a relationship between object types in different subsystems.
- In the **subsystem access model**, each subsystem connection represents a synchronous communication between objects in different subsystems. In a synchronous communication, one object accesses the state vector of another object in one atomic access operation.
- In the **subsystem communication model**, each subsystem connection represents an asynchronous communication between objects in different subsystems. In an asynchronous communication, one object sends an event to another object.

Each subsystem is a collection of interacting objects. These are modeled from three points of view, the *information*, *state* and *process* views. This gives us three kinds of related subsystem models:

1. The **information model** shows the classes of objects that make up the subsystem, and the relationships between these object classes. The OOA notation is a variant of the ER diagram notation.
2. For each object class with interesting behavior, a **state model** is produced. A state model is represented in the OOA notation by a finite state Moore machine, in which an action is executed upon arrival in a state. Each instance of a class executes an instance of the state model of the

OOA	TCM semi-formal notations	LCM
Information Model	Class-relationship model	X
State model	Life cycle model	X
	Transaction decomposition tables	X
	Transaction data flow diagram	
	Transaction communication flow diagram	
Process model	Transaction data flow diagram	
	Transition descriptions in data dictionary	X
Object communication model	Transaction communication flow diagram	

Figure 2.3: Correspondence between the OOA and TCM notations.

class, called the **state machine** (rather than state *model*) of the object. In addition, a class may have a single unique state model called an *assigner*, which can be used to create class instances and to synchronize state transitions of different objects in one subsystem.

3. Each action in each state model is specified in more detail by an **action data flow diagram** (ADFD). There is a single ADFD for each action in each state model — and, because the Moore representation of state models is used, therefore for each state in each state model. An ADFD is a network of processes and data stores connected by directed arcs, that represent data flows.

These three models are supplemented by a number of other models, some of which are derivable from the three main models, and others of which provide extra information. Two important supplementary models deserve special mention:

4. Data store accesses in ADFDs are represented on a separate **object access diagram**. As explained in chapter 5, this diagram represents *synchronous communication* between objects. In the OOA notation, the object access diagram is part of the process model.

5. An **object communication diagram** represents messages sent from one state model to another state model. As explained in chapter 6, this diagram represents *asynchronous communications* between objects.

We focus our discussion at the subsystem level. In chapter 7, we return to the domain and system level. As a result of our analyses, we will extend TCM with two semi-formal notations:

- A **transaction data flow diagram** to represent the flow of data in one system transaction.
- A **transaction communication flow diagram** to represent the flow of initiative in one system transaction.

To keep track of the discussion, figure 2.3 shows the correspondence between the OOA and TCM notations. The information in the transaction data flow and communication flow diagrams is not represented in the formal LCM specification. They can be used to provide useful supplementary information.

Chapter 3

The information model

3.1 OOA notation

The information model in OOA uses a variant of the Entity-relationship (ER) notation. As an example we discuss figure 2.3.2 from [39, page 23]. To understand this diagram, the following remarks are in order:

- Each box in the diagram represents an **object class**. Shlaer and Mellor call this an *object*, and call instances *object occurrences*. In order to reduce the gap with other methods and with the TCM notations, we talk of *object classes* and call instances *objects*.
- Each object class has something called an *identifier* by Shlaer and Mellor, which is a collection of one or more attributes whose combination always has a unique value for each instance of the class. These attributes are marked by an asterisk.
- All relationships in the information model are binary. They are represented by bidirectional arrows. Cardinalities are one-one, many-one, or many-many, with mandatory or conditional participation on either or on both sides. The "one" in these cardinalities is represented by a single arrowhead. It is exactly 1 if that side has mandatory participation, and it is 0 or 1 if that side has conditional participation. The "many" cardinality is represented by a double arrowhead. It means ≥ 1 if that side has mandatory participation and it means ≥ 0 if it has conditional participation. Conditional participation is represented by the letter "c" at the arrowhead.

OOA allows the definition of relationship attributes. As shown in figure 4.2.1 of [39, page 68], these are collected into something called an **associative object class** (`ITEM ORDER`), which is connected by an arrow to the relationship whose attributes are represented.

3.2 Comparison with class-relationship diagrams

The information model corresponds with **class-relationship diagrams** (CRDs) in TCM. Figure 3.1 shows the class-relationship diagram corresponding to figure 2.3.2 of [39, page 23]. There are some minor differences.

- In TCM, a class can be represented by a box containing the class name, and optionally:
  - a list of attributes and predicates applicable to class instances and
  - a list of state transitions of class instances.

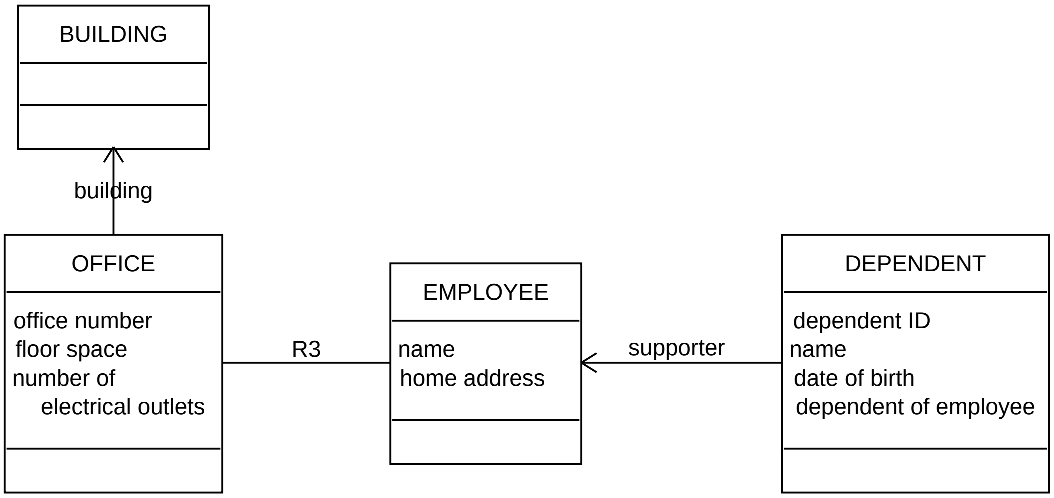


Figure 3.1: Class-relationship diagram corresponding to figure 2.3.2. (Role names have not been implemented in TCRD yet.)

All attributes, predicates and state transitions listed in the class box must also be present in the formal LCM specification (but the formal specification may contain more).

- A binary relationship is represented by a line. By default, the cardinality is many; a cardinality of 1 is expressed by an arrowhead. The relationship line must be annotated with at least one of the following: the name of the relationship or the names of the roles of the participating objects. This naming convention differs from that in the OOA notation.
- The identifiers of the OOA notation are called **keys** in TCM. Keys are not indicated in CRDs. An **identifier** in TCM is an attribute that, for objects that have ever existed, has a unique value, and that, once it has received a value for an existing object, never changes its value. Identifiers are indicated by an exclamation mark. For illustrative purposes, we transformed the `building` identifier in the information model into a many-one relationship, and transformed the other identifiers into TCM identifiers, thereby changing the meaning of the model.

Figure 3.2 contains the CRD corresponding to figure 4.2.1 of [39, page 68]. Two features of this CRD are:

- A relationship that itself has attributes, predicates or state transitions, is represented by a box connected with dashed arrows to the components of the relationship.
- A relationship is a also called a **link class** and its instances are called **links**.
- R2 is a derived relationship. This constraint can be specified in more detail in the data dictionary (not shown here) and, if there is a formal specification, must be specified by means of axioms in the formal specification.

TCM allows some additional constructs, that allow the specification of more powerful models. We briefly summarize these.

- LCM allows the specification of arbitrary first-order static integrity constraints. Cardinality constraints in LCM are more general than in OOA: for each component class of a relationship class, we can define a set C of natural numbers such that an existing component object

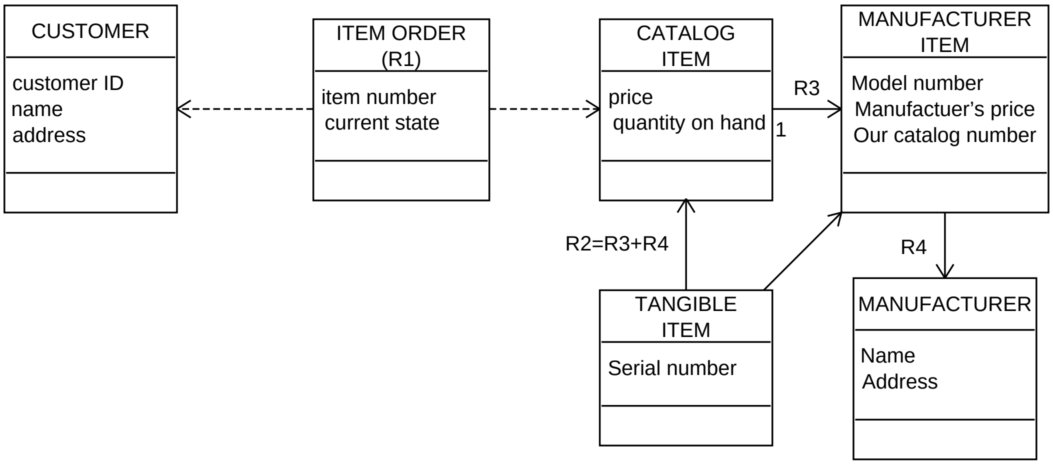


Figure 3.2: Class-relationship diagram corresponding to figure 4.2.1.

must participate in c existing instances of the relationship class, with $c \in C$. Other types of (noncardinality) constraints can be defined as well.

- Attributes and predicates can be declared to be `fixed` (once they have a value on an existing object, they will have this same value during the existence of this object). For each attribute or predicate, an initial value can be defined. For each attribute, it can be specified that there is an *inverse* attribute (which is set-values if the attribute is not an injective functions).

3.3 Specifying the class-relationship model in LCM

LCM is a syntactically sugared version of order-sorted dynamic logic with equality, combined with some elementary process algebra. For the class-relationship model, only the non-modal part of the dynamic logic fragment of LCM is needed. A specification of an information model in LCM consists of a collection of **value type specifications** and a collection of **class specifications**.

3.3.1 Value type specifications

The value type specification part of LCM is an order-sorted equational language that is a simplified form of OBJ [17, 18]. Figure 3.3 contains some example value type specifications. The value type `ZERO` contains one element, `0`, and is a subtype of `NATURAL`. Another subtype of `NATURAL`, not defined here, is `POSINTEGER`. In a meaningful specification, the specification of `POSINTEGER` must be included. A more elaborate number hierarchy is specified in [34].

A value type specification declares a signature consisting of a partially ordered set of sort names, zero or more function and predicate declarations, and a finite set of universally quantified positive conditional equational axioms. This part of the language uses the inference rules of equational logic, which allow substitution of equals for equals. The intended semantics of value specifications is the initial algebra semantics. A simple way to look at this is to view each closed term (a term not containing variables) as the syntactic representation of a value; different closed terms represent the same value if and only if they can be proved equal by using the axioms in the specification and the rules of equational logic. The **extension** of a type is the set of all instances of the type; this is the

```
begin value type ZERO
  functions
    0 : ZERO;
end value type ZERO;

begin value type NATURAL
  specializing POSINTEGER;
  specialized by ZERO;
  functions
    inc       : NAT -> POSINTEGER;
    dec       : POSINTEGER -> NAT;
    _ + _     : NAT x NAT  -> NAT,
  axioms
    forall n: NATURAL :: dec(inc(n)) = n;
    forall p: POSINTEGER :: inc(dec(p)) = p;
    forall n : NATURAL :: n + 0 = n;
    forall n, m : NATURAL :: n + inc(m) = inc(n + m);
end value type NATURAL;

begin value type OFFICE
  functions
    o: OFFICE;
    new : OFFICE -> OFFICE;
end value type OFFICE;
```

Figure 3.3: Example value type specifications in LCM.

set of all **values** that can be denoted by a closed term. The functions declared in a type specification are the operations that can be applied to the elements in the type extension.

3.3.2 Object class specifications

For each object class in the information model, we introduce a value type called the **internal identifier type** of the class, that declares an infinite set of closed terms of that type. The elements of the extension of the identifier type of a class are used as identifiers for the set of potentially existing objects of that class. For example, the `OFFICE` class has an internal identifier type specified in figure 3.3. Note that we use *values* as object identifiers. This makes object identification resistant to changes in notation. Different closed terms of type `OFFICE` that happen to be equal (denote the same value) therefore represent the same object identifier.

For each object class, we introduce a class specification that declares the attributes of the class as shown in figure 3.4. Each attribute is a function from object identifiers to a codomain. For example, `office_number` is a function `OFFICE` $\rightarrow$ `NATURAL` and in an application `office_number(o)`, the term `o` denotes an `OFFICE` identifier. The class `OFFICE` is called the **domain** of `office_number` and the type `NATURAL` its **codomain**. Note that the codomain may itself be an identifier type, such as `BUILDING`. The use of identifier types thus allows the declaration of "object-valued" attributes: the value of `building(o)` is the identifier of a `BUILDING` object. In different states of the specified system, attributes may have different interpretations. Thus, the value of `building(o)` depends upon the current state of the system.

Class specifications may declare unary predicates in addition to attributes. For each object class there is an implicitly declared predicate `Exists` that defines which instances actually exist. The set of class instances that currently exists is called the **existence set** of the class.

The `keys` section of a class specification declares one or more sets of attributes to be keys. This means that in any existence set, there is a one-one correspondence between key values and existing identifiers of the class. For example, in each existence set of `OFFICE`, there is a one-one correspondence between the values of `building(o)` and the identifiers of existing offices. The `keys` section is syntactic sugar for the axiom

```
forall o1, o2 : OFFICE ::
  Exists(o1) and Exists(o2) and building(o1) = building(o2) -> o1=o2;
```

As pointed out earlier, this is exactly what Shlaer and Mellor call an *identifier*.

LCM allows the declaration of **external identifiers**. `building` can be declared to be an external identifier by deleting it from the `keys` section and adding an `identifiers` section:

```
identifiers
  building;
```

There can be any number of external identifiers for a class. The meaning of the identifier declaration is that there is a one-one correspondence between on the one hand the internal identifiers of `OFFICE` objects that have ever existed or that currently still exist, and on the other hand the values of `building(o)` for `o` of type `OFFICE`. The difference between internal and external identifiers is that external identifiers are visible to the user and may be assigned by the user of the system. A detailed comparison of keys and identifiers is given elsewhere [46].

3.3.3 Link class specifications

Identification

Each link class is defined in LCM as a labeled Cartesian product of classes, called **component classes**. Component classes may themselves be link classes or object classes. A link is identified

```
begin object class OFFICE
  attributes
    building : BUILDING;
    office_number : NATURAL;
    floor_space: NATURAL;
    number_of_electrical_outlets: NATURAL;
  keys
    building;
end object class OFFICE;


begin link class R3
  components
    office : OFFICE;
    worker : EMPLOYEE;
  axioms
    --- Component existence axiom
    forall r : R3 ::
      Exists(r) -> Exists(office(r)) and Exists(worker(r));
end link class R3;

begin object class DEPENDENT
  attributes
    dependent_ID : NATURAL;
    name : STRING;
    date_of_birth : DATE;
    supporter : EMPLOYEE;
  identifiers
    dependent_ID;
  axioms
    --- Component existence axiom
    forall d : DEPENDENT::
      Exists(d) -> Exists(supporter(d));
end object class DEPENDENT;
```

Figure 3.4: Example class specifications in LCM.

```
begin link class R3'
  components
    office : OFFICE             mandatory;
    worker : EMPLOYEE           unique;
end link class R3;
```

Figure 3.5: Syntactic sugar for cardinality constraints in LCM.

by the labeled tuple of component identifiers. This composite identification is the only difference between links and objects. The `components` section of a relationship specification is syntactic sugar for the specification of an identifier type for relationships , in which the identifiers are labeled tuples of component identifiers:

```
begin value type R3
  functions
    R3_id : EMPLOYEE x  OFFICE -> R3
    office : R3 -> OFFICE;
    worker : R3 -> EMPLOYEE;
  axioms
    forall r : R3, e : EMPLOYEE, o : OFFICE::
      office(R3_id(e, o)) = o;
    forall r : R3, e : EMPLOYEE, o : OFFICE::
      worker(R3_id(e, o)) = e;
end value type R3;
```

This type supplies the identifiers of instances of class `R3`. These identifiers can be viewed as labeled tuples of `EMPLOYEE` and `OFFICE` identifiers. Because `works_in` and `assigned_to` are declared to be functions, their interpretation is the same in all possible states of the system (just like the interpretation of integer addition is the same in all states of the system). This means that one cannot "replace" a link component by another component; such a "replacement" is the destruction of one link and the creation of another. The projection functions `office` and `worker` are called the **component functions** of `R3`.

Component existence

LCM assumes a **component existence constraint** for links, which for each existing link require the link components to exist. These axioms are explicitly listed in figure 3.4 but are assumed to be implicitly present in each LCM relationship specification. We omit these axioms henceforth.

Cardinalities

The default cardinality in LCM is *many* and *optional*. Thus, for each existing `OFFICE` there are 0, 1 or more existing `EMPLOYEE` instances related to it by `R3`. If the cardinality is *one* or *mandatory*, this can be expressed by flagging the component functions by the keywords `unique` for cardinality *one*) or `mandatory`. Figure 3.5 illustrates this for R3. A **unique** component function says that the component function is mathematically an *injection*, because it says that different existing component instances belong to different existing links. Thus, in `R3'`, each existing employee is related to at most one existing office. A **mandatory** component function says that the component function is

```
begin object class CUSTOMER
  attributes
    customer_ID : NATURAL;
    name : STRING;
    address : STRING;
  keys
    customer_ID;
end object class CUSTOMER;

begin object class CATALOG_ITEM
  attributes
    catalog_number : NATURAL;
    price : MONEY;
    quantity_on_hand : NATURAL;
  keys
    catalog_number;
end object class CATALOG_ITEM;

begin link class R1
  components
    is_ordered_by : CUSTOMER         mandatory;
    has_ordered : CATALOG_ITEM       mandatory;
  attributes
    customer_ID : NATURAL;
    catalog_number : NATURAL;
    item_number : NATURAL;
    current_state : R1_STATES;
  keys
    customer_ID, catalog_number, item_number;
end link class R1;
```

Figure 3.6: Specification of alink class with attributes in LCM.

mathematically a *surjection*, because it says that each existing component is related to at least one existing link. In `R3'`, each existing office is related to at least one existing employee.

Associative objects

Object and link classes can contain attribute and predicate declarations, as well as event declarations axioms, and life cycle definitions (explained below). This means that associative object classes need not be specified in LCM. Attributes of relationships are declared as part of the relationship specification. Relationship `R1` of figure 4.2.1 of [39, page 68] can be specified in LCM as shown in figure 3.6. The component functions of `R1` have as codomain the identifier types of `CUSTOMER` and `CATALOG_ITEM`.

3.3.4 Intended semantics

The entire information model specification in LCM consists of a number of value type and class specifications, with the following intended semantics:

- The intended model is a Kripke structure of possible states, which all share the same domain. The domain of all possible states is the extension of the value type specifications. The sort names, function names and predicate names declared in the value type specifications thus have the same interpretation in all possible states. We denote the extension of type T by $ext(T)$. For example, the extensions of `OFFICE` and `NATURAL` are sets denoted by $ext(\texttt{OFFICE})$ and $ext(\texttt{NATURAL})$. Thus, $ext(\texttt{OFFICE})$ is the set of internal identifiers of all possible objects of the `OFFICE` class.
- Different states may differ in their interpretation of the attributes and predicates declared in the class specifications. For example, the existence predicate may have a different extension in different states. The set of existing objects of a class `C` in state σ is is defined as the set of elements of C for which `Exists` evaluates to true in σ. This is called the **existence set** of `C` in σ and is denoted in this paper by $ext_\sigma(\texttt{C})$. We regard the existence set of value types that are not identifier types to be equal to the entire extension of the type. Thus, $ext_\sigma(\texttt{NATURAL}) = ext(\texttt{NATURAL})$
- Attributes are interpreted as total functions from the existence set of the identifier type of the class into their codomain. For example, in each state, `floor_space` is interpreted as a total function $ext_\sigma(\texttt{OFFICE}) \to ext(\texttt{NATURAL})$. In different states, this may be a different function.

3.4 Taxonomic structures

3.4.1 OOA representation

OOA has a representation for subtypes and supertypes, shown in figure 2.6.2 of [39, page 29]. The intention is that in each state of the system, the existence set of the supertype is partitioned by the existence sets of the subtypes. Each existing instance of `TANK` is therefore also an existing instance of exactly one of the `TANK` subtypes.

3.4.2 Static subclass partitions

There are two different representations of taxonomies in TCM, depending upon whether an instance can migrate between subtypes or not. If migration is impossible, specialization is formalized by a **static subclass partition**. If migration is possible, specialization is formalized by a **mode partition**. Figure 3.7 shows a CRD with a static partition corresponding to figure 2.6.2. Figure 3.8 shows the corresponding formal specification in LCM. The `partitioned by` clause is syntactic sugar that generates implicit identifier type declarations in such a way that the identifier type of `TANK` is the immediate supertype of the identifier types of `STORAGE_TANK`, `MIXING_TANK`, and `HEATING_TANK`, and that identifiers are only defined for these subtypes (figure 3.9). The effect is that a `TANK` identifier is also the an identifier of exactly one of its subtypes. Because values cannot change type, objects cannot migrate between classes in a static partition.

Just as in OOA, subclasses may be further partitioned. In general, a taxonomic structure partitions the universe of all possible objects into *smallest classes*, called **species** in LCM. Identifiers are defined only for these smallest classes. Due to the subset relationships between the extensions of the identifier types, this creates the desired taxonomic structure.

3.4.3 Mode partitions

Figure 3.10 shows a mode partition corresponding to figure 2.6.2. Figure 3.11 shows the LCM declaration of a mode partition of `TANK`. Each of the subclasses is now a *mode* of `TANK` objects. The state space of a `TANK` object is partitioned exhaustively into mutually disjoint subspaces, each of which corresponds to a mode of the object.

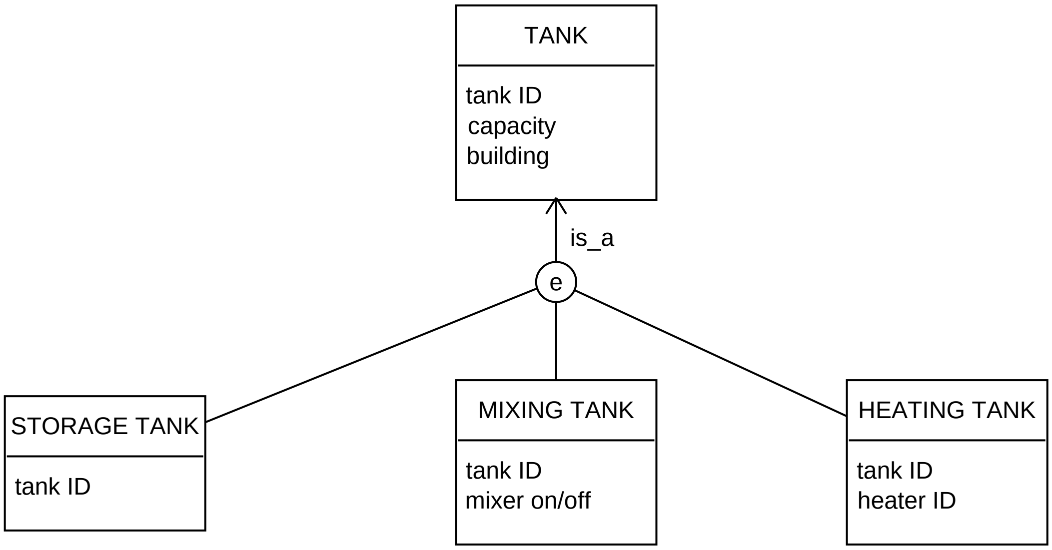


Figure 3.7: Representation of a static partition in TCM.

A **mode** is a particular kind of subclass. Modes are always subclasses (of other modes or of object or link classes). A mode can be a subclass of another mode, but after a finite sequence of supermodes we must always meet an object or link class, that we will call the **natural kind** of the mode. Each mode must have exactly one natural kind. Mode instances are instances of their natural kind that are in a particular state in which extra attributes, extra predicates, or extra constraints are applicable, or in which extra state transitions are applicable.

To formalize this intended semantics, the `mode partition` clause is syntactic sugar that generates implicit value type and object class specifications. The value type specifications merely declare the modes to be subtypes of their immediate supertypes, without declaring constants of these subtypes (figure 3.12). This means that the identifiers of instances of the modes `STORAGE_TANK` etc. are identifiers of `TANK`. However, in an arbitrary model, there may not be any instances of the modes. The expanded object class specifications in figure 3.12 enforce the intended semantics. In the expanded specification, we use for each mode a **mode predicate**, which in any state of the system is true exactly for the instances of the superclass that, in that system state, are in that mode. We can use this predicate to move an object into and out of a mode. In addition, for each mode a **retract attribute** is defined, which is used to find out which mode an object is currently in. For example, we use the retract attribute to conclude that a term of type `TANK` is also of type `STORAGE_TANK` when the mode predicate `Storage_tank` is true. Retracts are borrowed from Goguen and Meseguer [15]. The two axioms in the expanded `STORAGE_TANK` specification have the following meaning:

- The **mode predicate axiom** says that the mode predicate `Storage_tank` evaluates to true for all instances of `STORAGE_TANK`. In combination with the intended minimal semantics this says that `STORAGE_TANK` instances are *all* instances for which `Storage_tank` is true.

- The **retract axiom** defines the retract attribute to be the identity function on all `TANK` instances for which `Storage_tank` is true; in combination with the intended minimal semantics, this means that these are *all* the instances for which the retract is the identity function. The effect is that the retract axiom says that exactly when `TANK` objects are also instances of `STORAGE_TANK`.

- The mode predicates are used in an axiom that requires the mode classes in one partition to

```
begin object class TANK
  static partition STORAGE_TANK, MIXING_TANK, HEATING_TANK;
  attributes
    tank_id : NATURAL;
    capacity : NATURAL;
    building : BUILDING;
end object class TANK;

begin object class STORAGE_TANK
end object class STORAGE_TANK;

begin object class MIXING_TANK
  predicates
    mixer_on/off;
end object class MIXING_TANK;

begin object class HEATING_TANK
  attributes
    heater_id : NATURAL;
end object class HEATING_TANK;
```

Figure 3.8: Specification of a static class partition in LCM.

```
begin value class TANK
  specialized by  STORAGE_TANK, MIXING_TANK, HEATING_TANK;
end value class TANK;

begin value class STORAGE_TANK
  specializing TANK;
  functions
    s0 : STORAGE_TANK;
    new : STORAGE_TANK -> STORAGE_TANK;
end value type STORAGE_TANK;
```

Figure 3.9: Meaning of a static class partition specification in LCM.

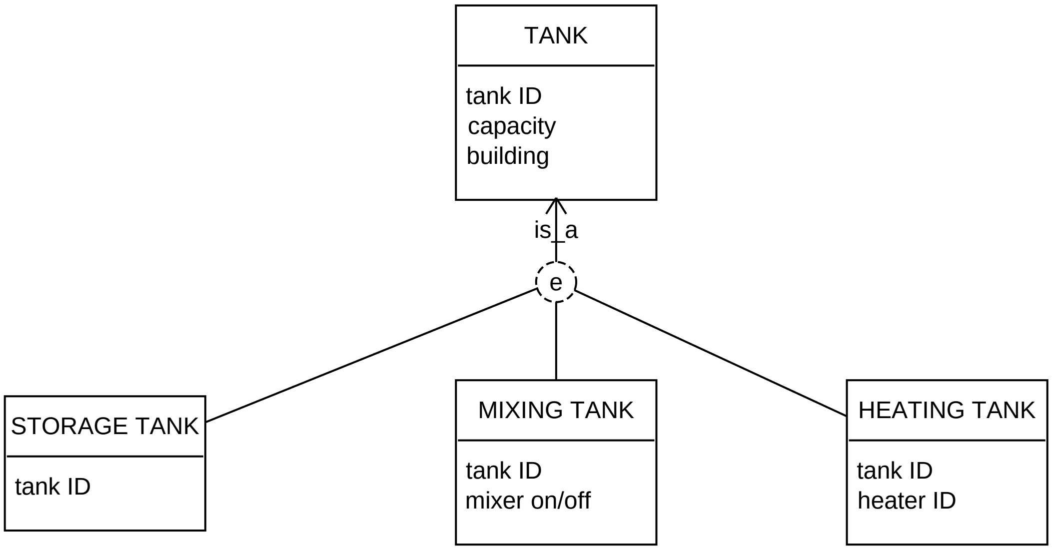


Figure 3.10: Representation of a mode partition in TCM.

```
begin object class TANK
  mode partition STORAGE_TANK, MIXING_TANK, HEATING_TANK;
  attributes
    tank_id : NATURAL;
    capacity : NATURAL;
    building : BUILDING;
end object class TANK;

begin mode STORAGE_TANK of TANK
end mode STORAGE_TANK;

begin mode MIXING_TANK of TANK
  predicates
    mixer_on/off;
end mode MIXING_TANK;

begin mode HEATING_TANK of TANK
  attributes
    heater_id : NATURAL;
end object HEATING_TANK;
```

Figure 3.11: Specification of a mode partition in LCM.

```
begin value class TANK
  specialized by  STORAGE_TANK, MIXING_TANK, HEATING_TANK;
end value class TANK;

begin value class STORAGE_TANK
  specializing TANK;
end value type STORAGE_TANK;

begin object class TANK
  mode partition STORAGE_TANK, MIXING_TANK, HEATING_TANK;
  attributes
    tank_id : NATURAL;
    capacity : NATURAL;
    building : BUILDING;
  axioms
--- TANK is partitioned by the three modes.
    forall t : TANK ::
      Storage_tank(t) or Mixing_tank(t) or Heating_tank(t) and
      not (Storage_tank(t) and Mixing_tank(t)) and
      not (Storage_tank(t) and Heating_tank(t)) and
      not (Heating_tank(t) and Mixing_tank(t));
end object class TANK;

begin mode STORAGE_TANK of TANK
  attributes
    retract_1 : TANK;
  predicates
    Storage_tank;                  --- Mode predicate
  axioms
--- Retract axiom
--- If Storage_tank is true only for terms of type STORAGE_TANK
    forall t : TANK :: Storage_tank(t) -> retract_1(t) = t;

--- Mode predicate axiom
--- If Storage_tank is true for all terms of type STORAGE_TANK
  forall s : STORAGE_TANK :: Storage_tank(s);
end mode  STORAGE_TANK;
```

Figure 3.12: Expanded specification of a mode partition in LCM. The expansions of the specifications of HEATING_TANK and MIXING_TANK are omitted.

be a partition of their immediate superclass. Note that this axiom is included in the `TANK` specification even though the mode predicates are declared elsewhere. In LCM, the scope of the names declared in a class specification extends to the entire text of the system specification.

Although this seems to be redundant, the mode predicate axiom and the retract axiom are both needed. In the intended minimal semantics, the mode predicate axiom says that the extension of the model predicate is the same as the extension of the mode class. Under this semantics, the retract axiom seems to be superfluous. However, if `Storage_tank(s)` is true, we cannot derive that the type of `s` is `STORAGE_TANK` unless we also have the retract axiom.

Conversely, the retract axiom allows us to derive the type of a term but only for those terms for which the mode predicate axiom is true. If we would use the mode predicate axiom to derive that the mode predicate is true, then we would already know the type of the term; rather, the mode predicate is set to true or false by mode-changing actions. So the mode predicate axiom seems to be superfluous here. However, if we declare a term `s` of sort `STORAGE_TANK`, then the retract axiom does not tell us that `Storage_tank(s)` is true; we need the mode predicate axiom for this.

A problem with the retract functions is that they are partial functions, which gives a problem with the initial semantics. This problem would be avoided if we use sort constraints [17] rather than axioms to define model classes. If we view the mode predicate axiom as a sort constraint, then given a particular model, it defines the extension of the `STORAGE_TANK` sort in that model. Since we are not aware of a sound and complete axiom system or an operational semantics for sort constraints, we used the retract approach.

3.4.4 More complex taxonomic structures

TCM allows more complex taxonomic structures, which we briefly summarize here.

- Link classes can be specialized just as object classes can.
- Modes can be specialized by more refined modes. It is required though that for each mode there is a unique object or link class that is its natural kind.
- In LCM, any number of static or mode partitions may be defined for a class. This gives us multiple inheritance structures. Because each partition is exhaustive, we still have that the universe of all possible objects is partitioned into species. With multiple partitions per class, some of these species will be intersection classes.
- TCM allows the specification of **role classes** as an alternative for modes.

3.5 Discussion

All constructs that can be expressed in the OOA notation can be expressed in TCM and can specified formally in LCM. However, to specify an OOA model in LCM, some decisions must be made that are left open in the OOA model.

- For each attribute, its codomain must be declared in LCM (but not in TCM). This in turn requires an explicit specification of the relevant value types. This kind of explicitness is not desirable in the early stages of specification. Formalization is feasible once the requirements model is stabilized and not earlier.
- Because the difference between static subclasses and modes is made explicit in TCM, the writer of the specification to decide whether a subclass partition is static or whether it is a mode partition. The distinction has important consequences for the way we identify class instances and we will see that it also has important consequences for the inheritance of object life cycles.

- TCM distinguishes *internal identifiers* from *external identifiers* and *keys*. The identifier concept of OOA corresponds to the key concept of TCM.

Chapter 4

The State Model

4.1 The OOA notation

For each object or relationship class with interesting behavior, a **state model** can be defined that represents the typical life cycle of class instances. The instantiation of a state model for class instances is called a **state machine** in OOA. Each class instance has a **state attribute** that indicates the current state of the object in its state machine. State models and state machines are **finite state automata**, which can be represented in two (equivalent) ways: the Mealy and the Moore representation. OOA uses the Moore representation. An example of a state model using the Moore representation is given in figure 3.3.1 of [39, page 40]. The state models use the following conventions:

- Each transition is labeled by an **event**, which is an incident in the real world to which the instance must respond. The event may be generated by an object in the system, or it may be generated in the environment of the system.
- In the Moore representation, actions can attached to states, with the meaning that the action is performed upon entry in the state. OOA allows the following actions [39, page 45]:
  - Read or write attributes of any object in the system.
  - Perform any calculation.
  - Generate an event. This event may be received by the environment of the system, by another object, or by the object itself (in its next transition).
  - Create, delete, set, reset or read a timer. A timer is an object that functions as an alarm clock. Any number of timers may be created in the system.

Relationship classes may have two state models, one that describes the life cycles of instances and one, called the **assigner**, that monitors the creation of relationship instances. There is at most one copy of the relationship assigner for each relationship class. We ignore assigners in what follows, because they can be represented in the same way as state models that describe the life of instances.

4.2 Specifying the Account model in TCM

In this section, we analyze the representation of the state model of a bank account class, given in figure 3.3.1 of [39, page 40].

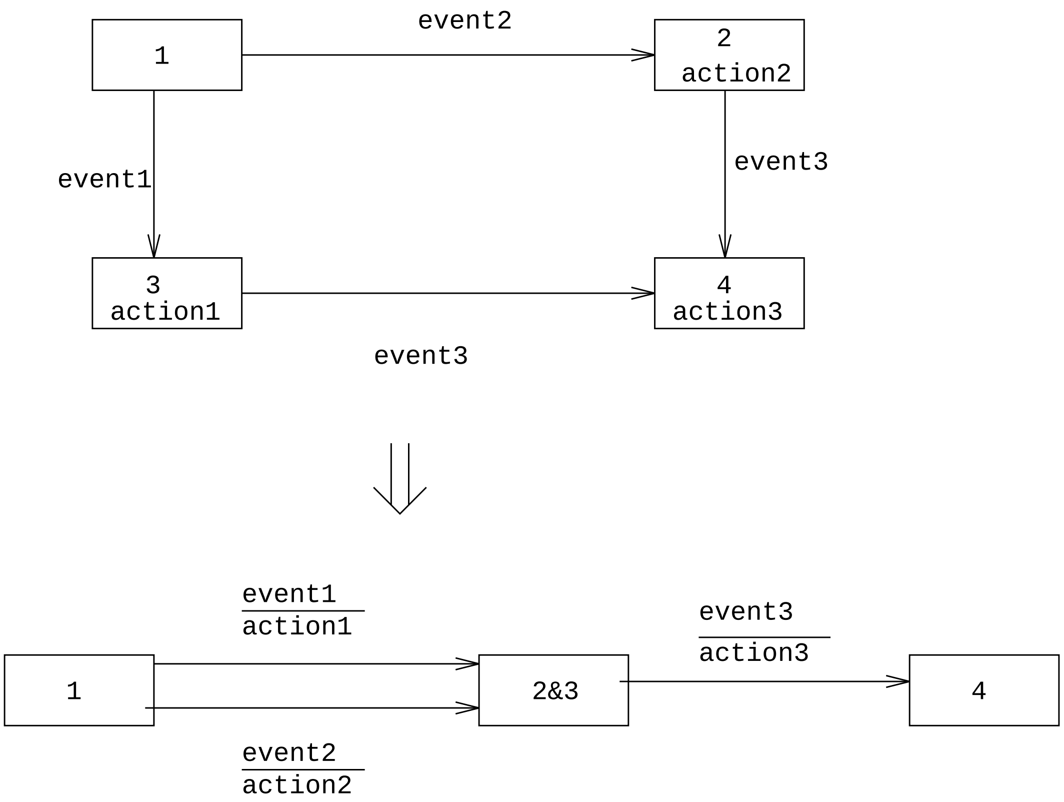


Figure 4.1: Transformation of a Moore automaton into an equivalent Mealy automaton.

4.2.1 Moore and Mealy automata

For each Moore automaton, there is an equivalent Mealy automaton, in which actions are attached to transitions rather than to states. Equivalence means that both automatons recognize the same input strings and generate the same output strings [23, page 44]. We obtain the Mealy automaton by moving the action associated with a state in the Moore automaton to all incoming transitions of the state (figure 4.1). Transitions are labeled by an event (input) above the line and/or an action (output) below the line. This represents the fact that the transition is triggered by the event and, when taken, generates an action.

The Mealy representation of finite state automatons often is simpler than the Moore representation. In a Mealy representation, incoming arcs of a state may generate different actions, whereas in a Moore representation, all transitions into a state lead the same action, viz. the action generated upon entry in the state. Translation of a Moore representation into a Mealy representation therefore often reveals states that were introduced only to be able to define different actions, and that are not necessary in the Mealy representation. Figure 4.1 illustrates this. Transforming the `ACCOUNT` life cycle of figure 3.3.1 [39, page 40] into a Mealy representation, we get the representation of figure 4.2. The states in figure 4.2 have been numbered in the same way as the states in figure 3.3.1 [39, page 40].

4.2.2 Stable and transitory states

There are two kinds of states in figure 4.2, stable and transitory. In a **stable** state of an object, the object is ready to respond to events initiated by its environment. In a **transitory** state, the

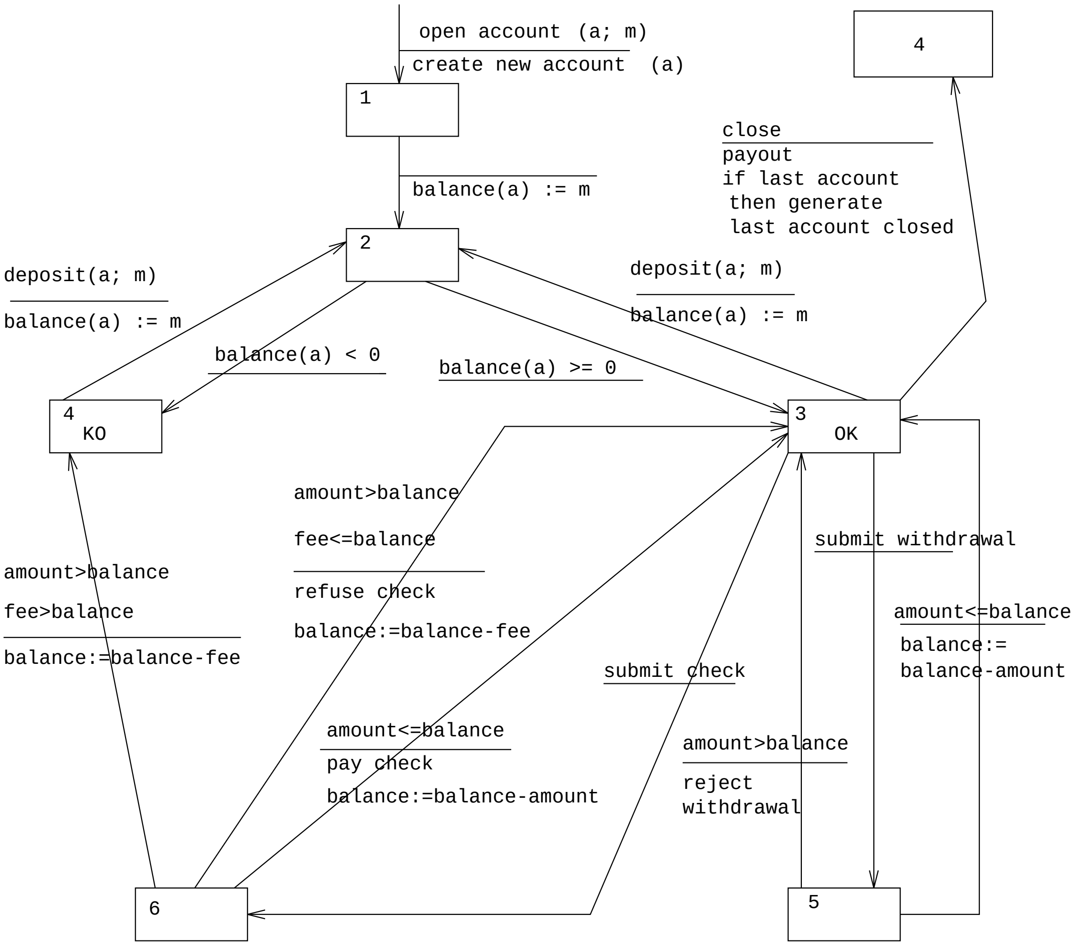


Figure 4.2: Mealy automaton with transitory states.

object is busy responding to an event that it has received. All events received when the object is busy producing a response to a previously received event, will be responded to after the current response has been completed. Events received by an object never get lost in OOA [39, page 107].

Let us define an object **transaction** as an atomic interaction between the object and its environment. An interaction is **atomic** if its intermediary states are not observable to the environment of the object. Viewed from the environment of the object, the object is in a state where either the transaction has occurred or it has not occurred. A transaction is always a sequence of one or more transitions from a stable state to a stable state. Transitory states occur as intermediary states during a transaction. Transitory states are not observable from the environment of the object.

In the automaton of figure 3.3.1 [39, page 40], only states 3 and 7 are stable (and therefore observable):

- Opening an account brings one into state 1, in which the Moore automaton generates a deposit action that brings the automaton into state 2. If the balance is non-negative, the current states moves to state 3, which is the OK state.

- In state 3, if a withdrawal is submitted, the automaton moves to state 5, in which it checks if the withdrawal is allowed. The automaton then performs a withdrawal or refusal action and brings the current state to 3.

- In state 3, if a check is submitted, the automaton moves to state 6. Acceptance of the check leads us back to state 3. If the check is refused, a fee is deduced from the balance and depending upon the resulting balance, we go to the OK state (3) or the KO state (7).

- In state 7, withdrawals and checks are refused. A deposit may lead us back to the OK state.

4.2.3 Object transactions and transitions

The Mealy automaton in figure 4.2 represents object transactions as follows: the event to be responded to causes a transition to a transitory state, in which a condition is tested that determines which next transition to take. When that next transition is taken, a response may be sent to the environment of the object, after which the object is in a stable state again. For example, when the event `submit check` occurs when the automaton is in state 3 (`OK`), the automaton moves to transitory state 6, in which it compares the amount on the check with the balance of the account. Depending upon the outcome of this test, the account moves to a stable state (`KO` or `OK`), possible sending a response to its environment.

We can represent an object transaction by a single state transition between stable states of a Mealy automaton if we add the possibility to test preconditions to the Mealy automaton convention. This is a common extension, shown figure 4.3. Each transition is labeled with an event and an action, possibly preceded by a precondition, which is a Boolean expression on the parameters of the event and on the state of the objects in the system. If the event occurs but the precondition evaluates to false, the transition cannot be taken.

Note that the precondition may test the global system state. In the Moore representation of the account state model, actions consist of a test on the event parameters and the object state, followed by an action that can be sent to the object itself (in its next transition) or to the environment of the object. An action can read attribute values from any other object in the system, so that the next transition to be taken depends upon the state of any other object in the system.

We can now simplify the Mealy automaton as follows:

- Remove all transitory states.

- Represent each object transaction by a single state transition in the extended Mealy notation.

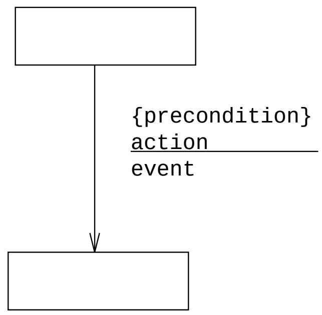


Figure 4.3: Mealy representation of a transition with preconditions.

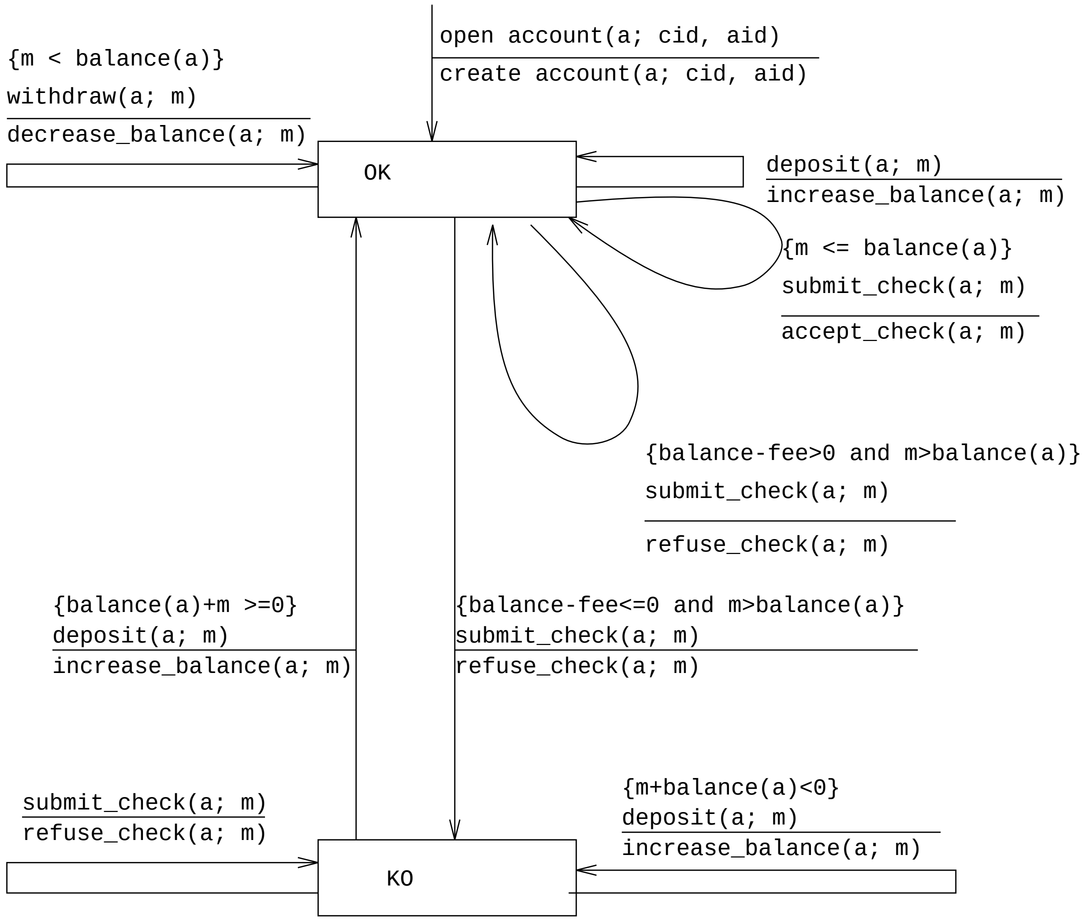


Figure 4.4: Mealy automaton with preconditions instead of transitory states.

The result for the `ACCOUNT` state model is shown in figure 4.4. (We added the `submit check` action in state `KO`. This action is absent from figure 3.3.1 [39, page 40] but there seems to be no reason to exclude it.)

We can always perform this simplification by compressing all actions performed in a sequence of transitions into one action. The only thing not representable in a Mealy automaton is an event sent to the automaton itself. However, such an event is sent to the automaton itself only in order to choose between different possible continuations of an object transaction that is started but not yet finished. This can be represented by means of preconditions on different possible object transactions in the simplified representation.

4.2.4 Separating local behavior from object communications

We now assume that each state transition of an object is a transaction of the object. For each object transaction, the following things are specified:

- the event(s) that trigger the object transaction;
- the precondition of the object transaction;
- the change of local state of the object caused by the object transaction (such as a change in value of value of the `current_state` attribute);
- the actions that are generated by the transition.

Note that the first and last items are part of the communication structure of the system. Events and actions are messages received from or sent to the environment of the object (i.e. other objects in the system or the environment of the system). On the other hand, the preconditions and local effects of a transition are part of the local behavior of the object. The preconditions may be global (they may test the state of any object in the system), but they guard a transition that is local to the object. We therefore split the specification into a local part, containing preconditions and a local state transition, and into a communication part, containing events and actions. Removing the communication part from the transitions in the account state model, we get the diagram is shown in figure 4.5.

The life cycle diagram must be supplemented with a CRD (figure 4.6) and a data dictionary entry for the class, that defines the attributes and state transitions of account objects (figure 4.7). The data dictionary defines the meaning of the attributes, gives any constraints on the states of the object, and defines the transitions. Each state transition of an object is assumed to be an object transaction. For each transition, the effect is defined (which must be local) and the precondition is defined (which may be global). Note that the preconditions of `accept_check` and `refuse_check` are mutually exclusive. Note also that the effect of `refuse_check` depends upon the state of the account. The preconditions and effects of the events are examples of transition constraints.

4.2.5 System transactions

In order to specify the events and actions, we should specify the possible communications between the objects in the system, and between the objects and the system environment. This is done in TCM by specifying **system transactions**. In TCM, a system transaction consists of a set of one or more local transitions of different objects. A system transaction has two important properties:

- If a system transaction consists of more than one local object transaction, then there is a partial ordering between these object transactions that represents *causality*. In this ordering, one object transaction may be an action generated by some object, that plays the role of event with respect to another object.

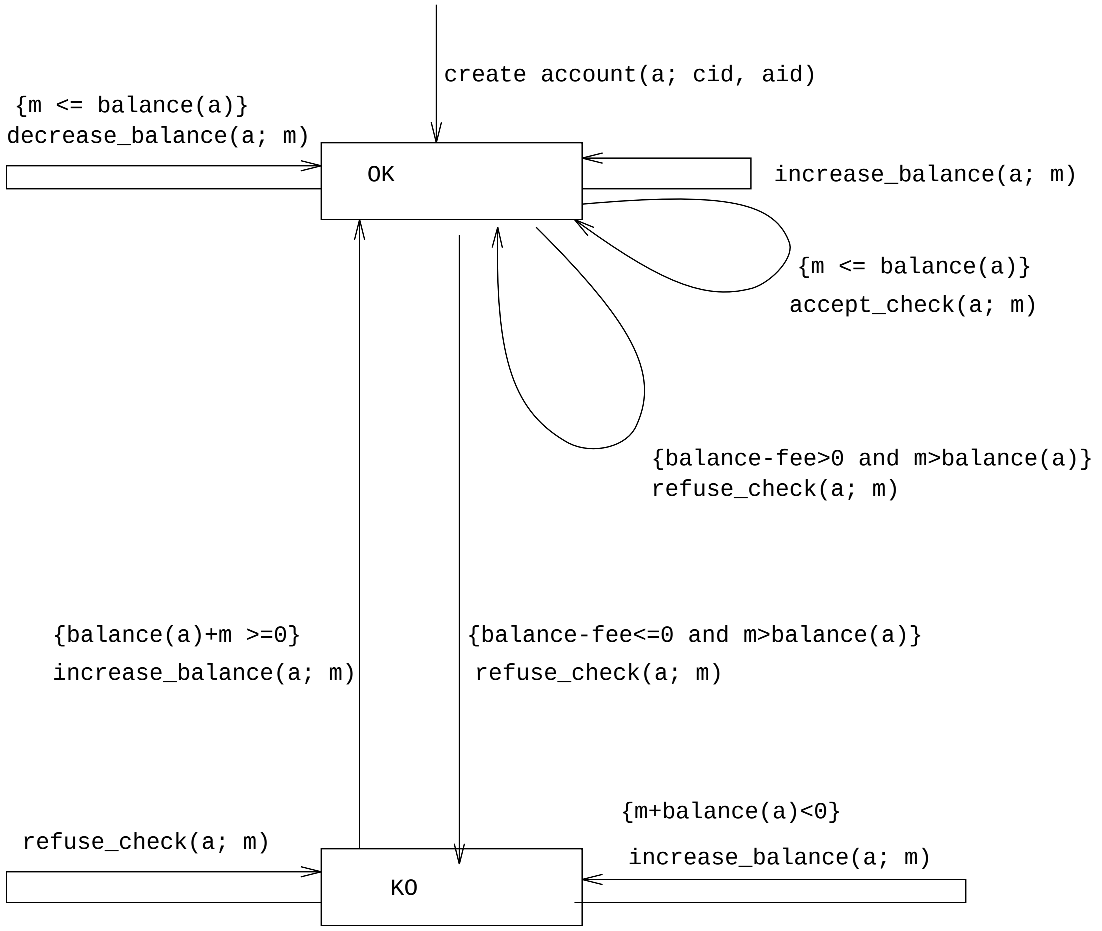


Figure 4.5: Life cycle diagram of account objects.

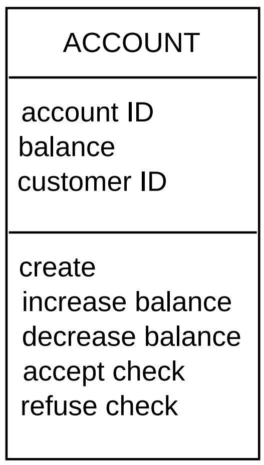


Figure 4.6: Fragment of a class-relationship diagram containing the ACCOUNT class.

ACCOUNT object class.
Attributes:

- account_ID is an external identifier of ACCOUNT objects.
- balance holds the current balance of an account object.
- customer_ID is the external identifier of the owner of the account.
- current_state indicates whether the account is in the OK state or in the KO state.

State constraints:

- The customer of an existing account must exist.

Transitions:

- create(a; account ID: aid, customer ID: cid). Create an ACCOUNT object with a fresh internal identifier a, an account ID = aid, customer ID = cid, balance = 0 and current state = OK.
- increase_balance(a; m).
- decrease_balance(a; m). Subtract m from the balance of a. This happens only if the result of subtraction is non-negative.
- accept_check(a; m). Subtract m from the balance. This is done only if the result is non-negative.
- refuse_check(a; m). Subtract the fee for handling a refused check from the balance of a. If the result is negative, go to the KO state, otherwise go to the OK state. refuse_check(a; m) can happen only if m is larger than the current balance of a.

Figure 4.7: Data dictionary entries for the ACCOUNT transitions.

	open_account	deposit	withdraw	submit_check
ACCOUNT	create	increase_ balance	decrease_ balance	accept_check + refuse_check

Figure 4.8: Transaction decomposition table of ACCOUNT transactions.

- Conceptually, all object transitions in a system transaction occur synchronously. This means that there is not intermediary state of the system transaction. The objects participating in the system transaction may read the states of other objects at the beginning of the system transaction, but they cannot observe a system state in which some object transactions have occurred and others have not.

These two properties jointly form the **synchronicity assumption**, which is known from ESTEREL [4] and Statemate [24, 20] and which is also made by real-time methods such as the Ward-Mellor method [41, 42, 29] and the Hatley-Pirbhai method [21]. In the Shlaer-Mellor notation, communication is asynchronous. The synchronicity assumption is therefore a considerable departure from the treatment of communication in the Shlaer-Mellor notation. We discuss the issue of asynchronous versus synchronous communication in more detail in chapter 6. Here, we illustrate the specification of system transactions for the bank account example.

System transactions can be represented semi-formally by a **transaction decomposition table** such as the one shown in figure 4.8. This transaction decomposition table is extremely simple because each system transaction is decomposed into exactly one object transaction. To get the effect of the system transaction, one simple reads the effect of this object transaction in the data dictionary. (An intelligent CASE tool should be able to compose the system transaction from the data dictionary definitions of the object transactions.)

The submit_check event in the account example is nondeterministic in the sense that its effect depends upon the state of the account object. If we would draw the Mealy automaton of the account object, labeling the transitions with events only, we would get a nondeterministic automaton. The nondeterminism has the following features:

- Depending upon the event parameters and the state of the account object, submit_check may generate actions accept_check or refuse_check. Since an action is a message sent to the environment of the object (to other objects in the system or to the environment of the system), this nondeterminism is visible to the environment.

- Depending upon the event parameters and the state of the account object, submit_check may have different local effects on the account object. For example, it may cause a transition to the OK or to the KO state.

These two forms of nondeterminism can occur separately or jointly.

The transaction decomposition table shows which object transactions occur (synchronously) within a system transaction, but does not show a causal chain within the system transaction. It may enhance our understanding of the system transactions of we represent the causal ordering of transaction components. Figure 4.9 shows two **transaction communication flow diagrams** (CFDs) of the bank account system. A transaction CFD is a rooted directed graph that represents the causal chain of object transactions within a system transaction.

- The root of the graph is an invisible node that represents initiative in the environment of the system.

- The nodes represent (arbitrary) objects that participate in the transaction.

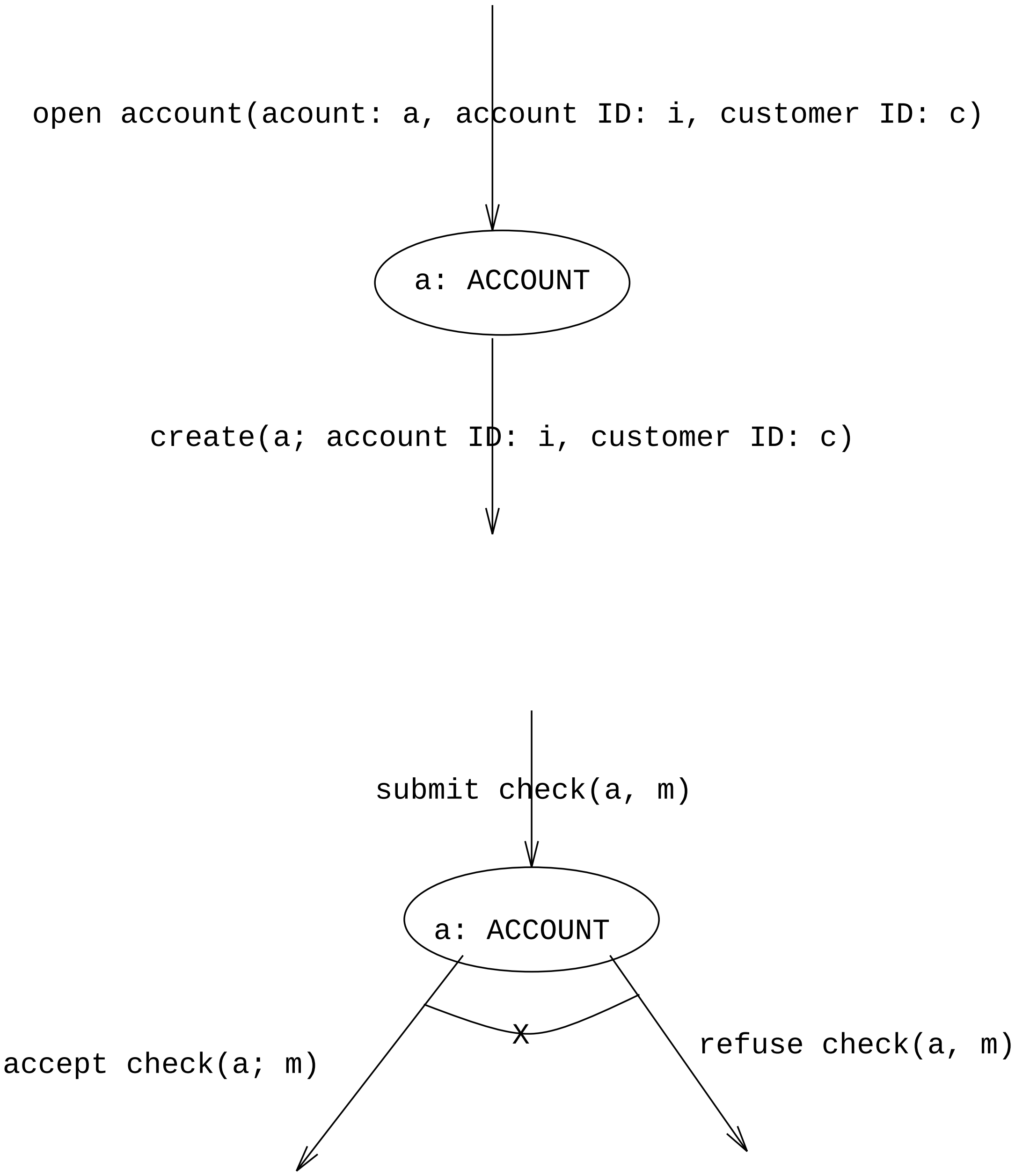
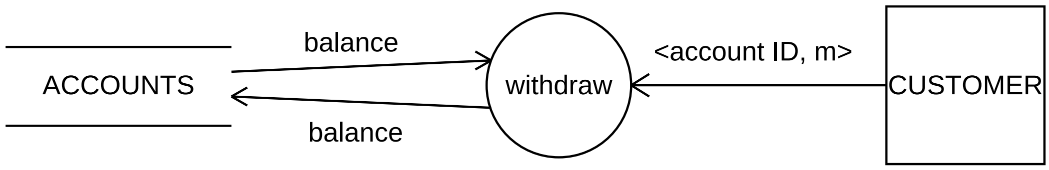


Figure 4.9: Transaction communication flow diagrams of the bank account system.



- `decrease_balance(a; m)`. Subtract `m` from the `balance` of `a`. This happens only if the result of subtraction is non-negative.

Figure 4.10: Transaction data flow diagram of an `ACCOUNT` transaction.

- The edges represent events (from the viewpoint of the node mpointed at) and actions (from the viewpoint of the node at the start of the arrow). An edge must be labeled by a local transition participating in the transaction. The root arrow is labeled by the name of the transaction, all other edges are labeled by the name of local transitions of the object at the start of the arrow.
- Multiple edges that leave a node represent multiple actions, sent to the environment of the node conjunctively. If two actions are mutually exclusive, they must be joined by an arc marked with an "x".
- An edge that points to an invisible node represents a response by the system to the transaction.

Note that we cannot merge transaction CFDs into one CFD, because the flow of control would then be lost.

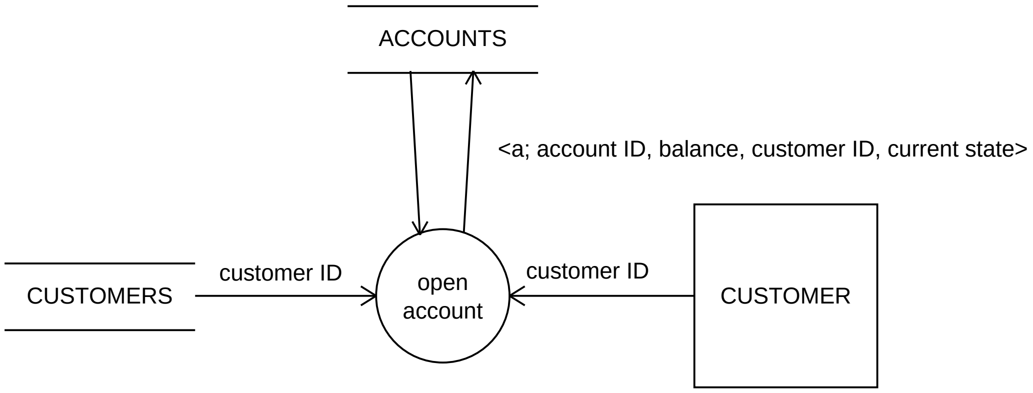
A system transaction can be documented in a third way, that illustrates the flow of data during the transaction. A **transaction data flow diagram** (DFD) is a DFD that satisfies the following rules:

- It contains one bubble, that has the name of the transaction (without parameters).
- It contains one data store for every object or link class that participates in the system transaction. A class participates in a system transaction if one of the preconditions of the transaction must read the state of an instance of the class, or if an instance of the class participates in the transaction.
- Each data flow is labeled by the data that flows through it, using as names either object attributes or the names of transaction parameters. If a parameter is nameless, a formal parameter name can be used.

Figures 4.10 and 4.11 show two transaction DFDs of the bank account system. Each DFD is annotated with the entry of the data dictionary that describes the transaction. This entry is is the conjunction or disjunction of the descriptions of the object transactions participating in the system transaction, where conjunctions and disjunctions are indicated by the transaction decomposition table.

The useful information that a TFDF gives is where the parameters of the transaction comes from and where they go to. There is a simple relationship between the TDFD of a transaction and the decomposition of the transaction shown in the transaction decomposition table: The TDFD shows a write access to the data store of each class that participates with a local event in the transaction, and it contains a read access to each data store that contains instances whose state must be read to check a precondition of the transaction.

If needed, TDFDs can be merged in higher-level DFDs by grouping transactions into functions as indicated by the function decomposition tree. In no circumstance should the data transformation that represents a transaction be decomposed in lower-level transformations, because we then descend to the operational level and would be writing a kind of program to implement the transaction.



- `create(a; account ID: aid, customer ID: cid)`. Create an `ACCOUNT` object with a fresh internal identifier `a`, an `account ID = aid`, `customer ID = cid`, `balance = 0` and `current state = OK`.

Figure 4.11: Transaction data flow diagrams of two `ACCOUNT` transactions.

```
begin object class CUSTOMER
  attributes
    customer_ID : NATURAL                      inverse;
  ...
  identifiers
    customer_ID;
  ...
end object class CUSTOMER;
```

Figure 4.12: The `CUSTOMER` class has `customer_ID` attributes, that is unique on existing customers and that has an inverse attribute.

4.2.6 Formal specification in LCM

The life cycle of an object can be specified in dynamic logic if we introduce for each object an attribute `current_state` that indicates the current stable state of the automaton (just as is done in the Shlaer-Mellor notation). We then use dynamic logic to specify the precondition and postcondition of each transition. Figure 4.13 shows the result for the account state model; this specification presupposes the `customer_ID` attribute of `CUSTOMER` objects (figure 4.12).

- We assume a value type `ACCOUNT_STATE` has been defined, containing the values `OK` and `KO`.
- We added the attribute `current_state` of type `ACCOUNT_STATE`; normally, this would be left implicit.
- All possible actions of `ACCOUNT` objects are declared in the `transitions` section of the specification.
- The axioms use dynamic logic to express preconditions and effects. The formula $[\alpha]\phi$ means that after every possible execution of the action α, the formula ϕ evaluates to *true*. Note that the formula $[\alpha]\phi$ is true for any ϕ if α is nonterminating (does not lead to a next state). Its dual

```
begin object class ACCOUNT
  functions
    fee : MONEY;
  attributes
    account_ID : NATURAL;
    balance : MONEY                              initially 0;
    customer_ID : NATURAL;
    current_state : ACCOUNT_STATE                initially OK;
  identifier
    account_ID;
  transitions
    create(ACCOUNT; account_ID: NATURAL,
                    customer_ID: NATURAL)            creation;
    increase_balance(ACCOUNT; MONEY);
    decrease_balance(ACCOUNT; MONEY);
    accept_check(ACCOUNT; MONEY);
    refuse_check(ACCOUNT; MONEY);
  state axioms
--- Static constraint: the customer must exist
    forall a : ACCOUNT::
      Exists(a) -> Exists(inv_customer_ID(customer_ID(a)));

  transition axioms
--- effect axiom for increase_balance
    forall a : ACCOUNT , b, m : MONEY ::
      balance(a) = b and B+m >= 0 -> [increase_balance(a; m)] balance(a) = b+m;

--- precondition axiom for decrease balance
    forall a : ACCOUNT , m : MONEY ::
      <decrease_balance(a; m)>true -> current_state=OK and m <= balance(a);
--- effect axiom for decrease balance
    forall a : ACCOUNT , b, m : MONEY ::
      balance(a) = b -> [decrease_balance(a; m)] balance(a) = b-m;

--- precondition axiom for accept_check
    forall a : ACCOUNT , m : MONEY ::
      <accept_check(a; m)>true -> current_state=OK and m <= balance(a);
--- effect axiom for accept_check
    forall a : ACCOUNT , b, m : MONEY ::
      balance(a) = b -> [accept_check(a; m)] balance(a) = b-m;

--- precondition axiom for refuse_check
    forall a : ACCOUNT , m : MONEY ::
      <refuse_check(a; m)>true -> m > balance(a);
--- effect axioms for refuse_check
    forall a : ACCOUNT , b, m : MONEY ::
      balance(a) = b and fee > b ->
      [refuse_check(a; m)] balance(a) = b-fee and current_state(a)=KO;
end object class ACCOUNT;
```

Figure 4.13: Specification of the ACCOUNT object class in LCM, using the current_state attribute.

$\langle\alpha\rangle\phi$ means that there is a possible execution of α that terminates and leads to a state where ϕ is true. An axiom of the form $\langle\alpha\rangle true \rightarrow \phi$ says that α only leads to a next state if (currently) ϕ is true. We call ϕ a **necessary precondition for success** of α.

- There are two kinds of axioms, state axioms and transition axioms. A **state axiom** is a sentence, not containing modal operators, that must be true in all states of the system. There is a state axiom in the account specification that says that the customer of an existing account must exist. Note that the `customer_ID` attribute of `CUSTOMER` is overloaded. `inv_customer_ID` is the inverse of the `customer_ID` attribute of `CUSTOMER`. State axioms may be hidden in keywords, such as flags (e.g. `inverse`) or in `identifier` declarations, etc. The declarations that `customer_ID` and `account_ID` are external identifiers of `CUSTOMER` and `ACCOUNT`, respectively, are syntactic sugar for certain state axioms.

- There are two kinds of **transition axioms**, viz. precondition and effect axioms. A **precondition axiom** has the form
$$\langle\alpha\rangle true \rightarrow \phi.$$
This means that if α leads to a next state, then (currently), ϕ is true. This means that ϕ is a precondition for α, i.e. if ϕ is false, then α does not lead to a next state. If α is an atomic action, as it is in the specification, this means that α cannot occur if ϕ is false. **Effect axioms** have the form
$$\phi \rightarrow [\alpha]\psi.$$
This means that if ϕ is currently true, then after execution of α, ψ is true.

- Each action except the action labeled `creation` has as implicit precondition that the object in whose life it occurs exists, i.e. the `Exist` predicate must evaluate to true for the object. The action labeled `creation` has implicit axioms which state as precondition that the object in whose life it occurs does not exist, and as effect that the object exists. These axioms are not shown in the specification. The details of this are straightforward [13].

- The precondition and effect definitions of `refuse_check` entail that `refuse_check` causes the account object to stay in the `KO` state once it is there.

The specification as it stands is incomplete, for it must be supplemented with a **frame assumption** which says that everything that cannot be proved to change, remains invariant during an action. For example, because nothing is said about the effect of `increase_balance` on `current_state`, the frame assumption allows us to conclude that this attribute does not change. In addition, the specification must be supplemented with a **qualification assumption** which says that the conjunction of all necessary preconditions of success for a particular action is also a sufficient precondition for success of the action. For example, the necessary precondition of success for `refuse_check(a; m)` is `m > balance(a)`. Since there is no other precondition for the success of `refuse_check(a; m)`, this is also a sufficient precondition for success.

These assumptions are not expressible as first-order axioms. One way around this is to generate frame and qualification axioms for each individual specification automatically [47]. Other ways of dealing with this are currently under study [12].

Figure 4.14 specifies the transactions of the account system. The following remarks are in order:

- The LCM specification formalizes the transaction decomposition table. Transaction CFDs and transaction DFDs are not formalized, because the flow of initiative and the flow of data are not represented in LCM.

- Because all events of account objects come from the environment of the system, they all appear as system transactions. Each transaction consists of one local transition only.

```
begin system function AccountTransactions
  transactions
    open_account(account: ACCOUNT, account_ID: NATURAL, customer_ID: NATURAL);
    deposit(ACCOUNT, MONEY);
    withdraw(ACCOUNT, MONEY);
    submit_check(ACCOUNT, MONEY);

  decompositions
    forall a: ACCOUNT, aid, cid: NATURAL::
      open_account(account: a, account_ID: aid, customer_ID: cid) =
      create(a; account_ID: aid, customer_ID: cid);

    forall a : ACCOUNT, m: MONEY::
     deposit(a, m) = increase_balance(a; m);

    forall a: ACCOUNT, m :MONEY::
      withdraw(a, m) = decrease_balance(a; m);

    forall a: ACCOUNT, m : MONEY::
      submit_check(a, m) = accept_check(a; m) + refuse_check(a; m);

end system function AccountTransactions
```

Figure 4.14: Transactions of the account system.

- The `submit_check` transaction is nondeterministic. The choice between `accept_check` and `refuse_check` is made by the preconditions of the transitions in the state model of `ACCOUNT`. Furthermore, the nondeterminism of the two possible local effects of `refuse_check` is note visible in the transaction definition. This choice is made by the preconditions as well.

4.3 Analysis of the Temperature Ramp example

4.3.1 Analysis of the state model

Figure 6.2.2 from [39, page 114] contains a state model of a temperature ramp object class. This is part of a model of a juice plant controller, which controls the processing of batches of fruit juice. The juice is mixed in a cooking tank and put into cans. The entire model specifies the controller for the cooking tank and the canning operation and is specified more fully in [38]. The temperature ramp state model of figure 6.2.2 contains the following actions:

- **Created**. Upon creation of a cooking ramp, a timer is created for this ramp, the current time and the current cooking tank temperature are stored in the attributes of the temperature ramp, and the ramp sends an event to itself to transition to the controlling state.
- **Controlling**. Upon entry in the controlling state, and henceforth every time the timer expires, the ramp compares the actual cooking tank temperature with the desired temperature and turns the heater on or off accordingly. If the temperature is not high enough, the time is set to 10 seconds. If the temperature is high enough, the ramp sends an event to itself to transition to the complete state.

- **Complete**. Upon entry in this state, the ramp sends an event to its environment that it is complete, turns off the heater, deletes the timer and deletes itself.

The actions to be performed in the controlling state thus depend upon the current state.

Note that the heater is turned on every time that the time expires, which is redundant (since it is turned on already upon entry in the controlling state). Note also that the start temperature, which is an attribute of Temperature Ramp, is never used.

Figure 4.15 shows a Mealy automaton that corresponds to the Moore automaton in figure [39, page 114]. The term `actual_temperature(tank(bid))` represents the actual temperature of the tank of the current batch, where `bid` identifies the batch to be mixed. The values `bid` and `desired_temperature` are given as actual parameters of the event `do_temperature_ramp`. Note that the term `actual_temperature(tank(bid))` can only be written down after we declare the attributes of `HEATING_TANK` and `BATCH`, as shown in figure 4.16. When those attribute are not yet declared, we should use an informal description of the test.

Upon creation, the Mealy automaton in figure 4.15 enters a transitory state that it leaves immediately (in the same object transaction). Depending upon the actual temperature, the heater is turned on or off and the Temperature Ramp continues monitoring the temperature or goes to the completion state. The transitory state allows us to enforce a sequence between a single triggering event (Do Temperature Ramp) and a test that determines which action is generated by the event. It also allows is to hide the nondeterminism of the automaton: there are external events (`do_temperature_ramp` and `timer_expires`) each of which causes a transition to one out of two possible next states. This is made more visible if we eliminate the transitory state, as shown in figure 4.17. In this Mealy automaton, each transition represents a single transaction of a Temperature Ramp object. The nondeterministic representation in figure 4.17 is simpler than the representation with the transitory state. In particular, some of the actions generated when `do_temperature_ramp` enters the state `CONTROLLING` need not be done when `do_temperature_ramp` enters the state `COMPLETE`.

As in the case of the account life cycle, removal of transitory states increases the nondeterminism of the state model. For example, the event `do_temperature_ramp` is nondeterministic, since it may lead to the state `CONTROLLING` or to `COMPLETE`. The choice depends upon a test on the state of the environment of the object. If `actual_temperature(tank(bid)) < desired_temperature` then the temperature ramp is created, a timer is created, the heater is turned on and the ramp periodically checks whether this condition still holds. Again, the nondeterminism concerns two things:

- which the state transition is triggered by the nondeterministic event and
- which action is generated by the transition.

There is are two conceptual problems with the automaton in figure 4.17, which are related:

- First, the Mealy automaton has two initial states, as shown by the two arrows entering from nowhere.
- If `actual_temperature(tank(bid)) >= desired_temperature` upon entry of the diagram, then the temperature ramp need not be created. But then there is no object to send the response `temperature_ramp_complete!`. It is not possible to create the ramp, send the `temperature_ramp_complete` message, and delete the ramp in one transition.

These problems should have been avoided by testing the creation condition for the temperature ramp *before* the temperature ramp state model is entered. Thus, the batch that sends the `do temperature ramp` message should not do this when the temperature in the tank is already high enough. This would also reduce the number of initial states of the temperature ramp to one, which is as it should be. We assume henceforth that the batch checks the precondition before sending a `do temperature ramp`

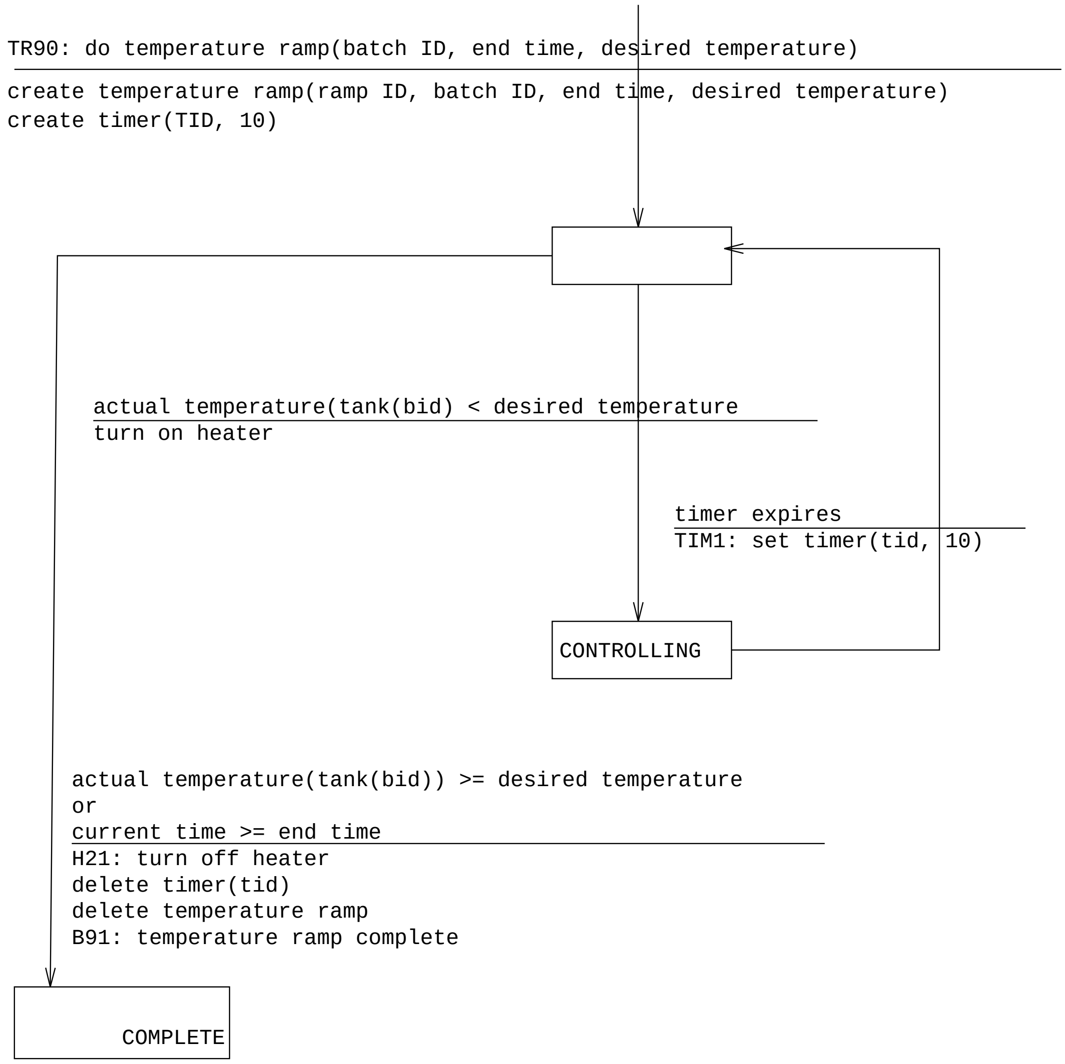


Figure 4.15: Nondeterministic Mealy automaton with a transitory state.

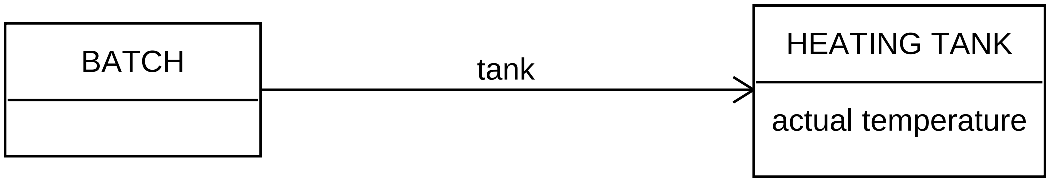


Figure 4.16: Part of a class-relationship diagram of the juice plant controller.

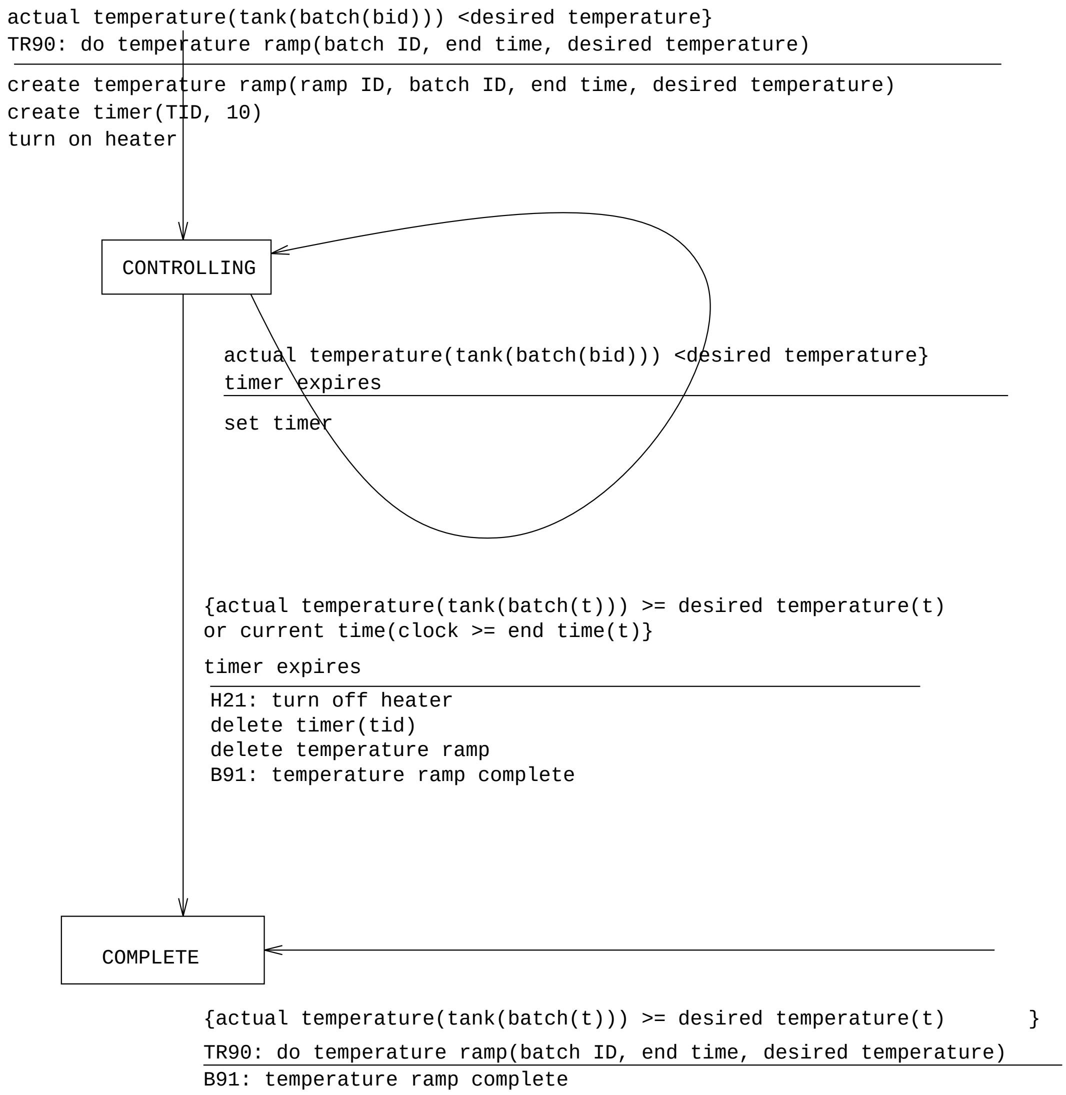


Figure 4.17: Nondeterministic Mealy automaton.

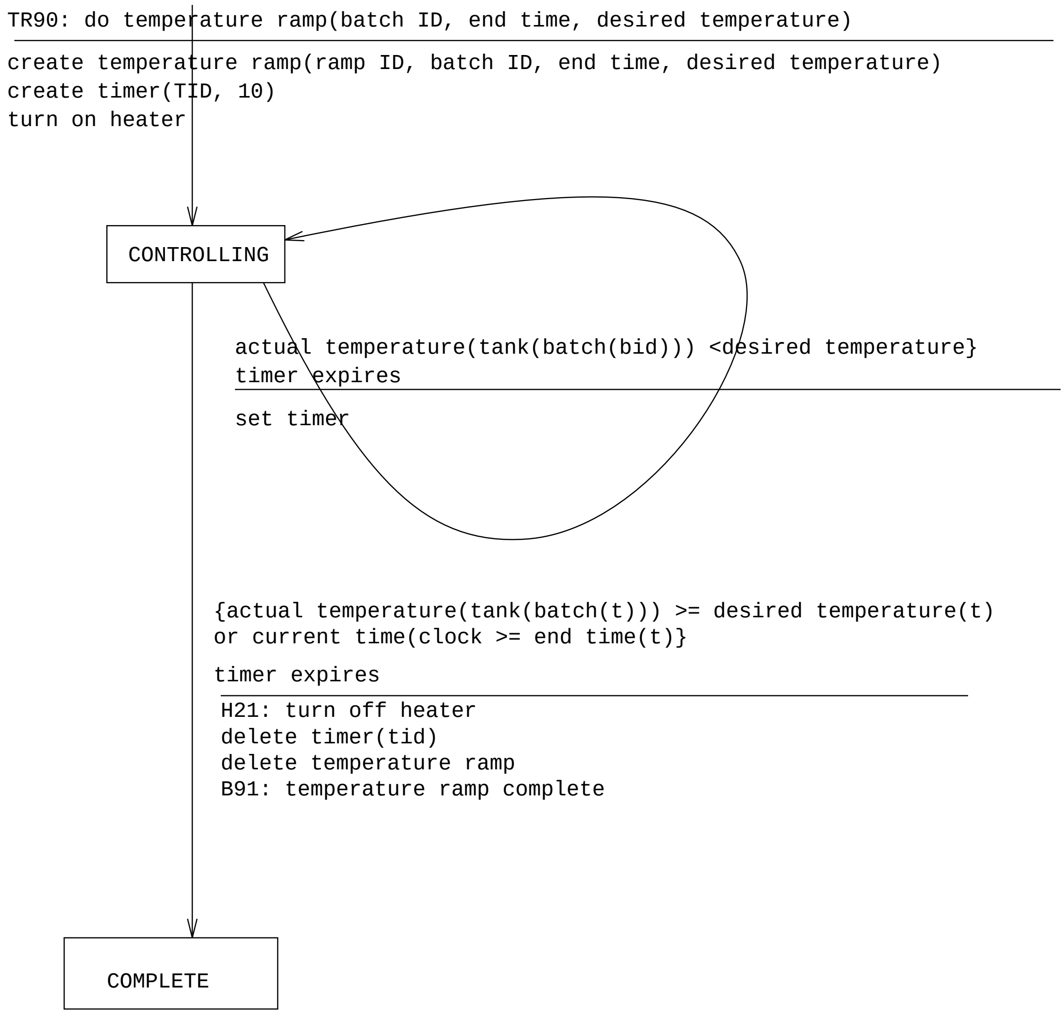


Figure 4.18: State model of the temperature ramp, assuming that the precondition for creation has already been checked when do_temperature_ramp is sent.

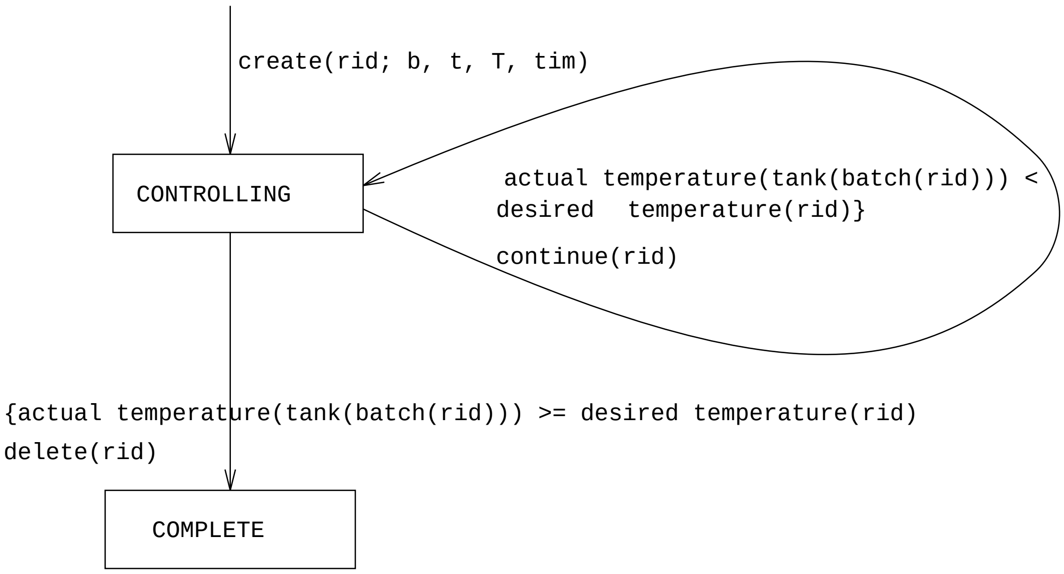


Figure 4.19: Preconditions and local state transitions of the temperature ramp.

message to create a temperature ramp. This further simplifies the temperature ramp state model to the one shown in figure 4.18.

We finally separate communication information (events and actions) from local information (preconditions and local state change). Figure 4.19 shows the resulting life cycle diagram of the temperature ramp state model. Figures 4.20 and 4.21 give a fragment of the relevant class-relationship diagram and of the data dictionary, respectively. We assume that there is one heater for the entire system. This probably wrong (more likely, there is one heater for each heating tank), but there is nothing in the Shlaer-Mellor case study that indicates how many heaters there are.

4.3.2 System transactions

Figure 4.22 gives the decomposition table of some of the juice plant controller transactions. To find a description of the transactions in informal terms, that can be understood by domain specialists, one can simply conjoin the descriptions of the preconditions and local effects of the local events in each transaction.

Figure 4.23 contains a transaction CFD for the `start_controlling_temperature` transaction and figure 4.24 contains a transaction DFD. The DFD raises a methodological issue, i.e. when a class appears in a DFD as data store and when it appears as external entity. The rule is simple: If the instances of the class are created, manipulated and deleted in the store of the computer system, then they should appear as data stores; if they are entities in the environment of the controller, that must communicate with the controller, then they should appear as external entities. The issue is nevertheless subtle and can give rise to misunderstandings:

- `ACCOUNT` instances may be records in a database or objects in a UoD. As objects in the UoD, they are invisible, for a bank account is a social construction, a legal entity that fixes an agreement between a bank and a customer, consisting of a system of mutual rights and obligations. In the bank account example, we only represent the data store and not the external entity, because the only way that the bank and its customers interact with the UoD bank account object is by manipulating the data store.

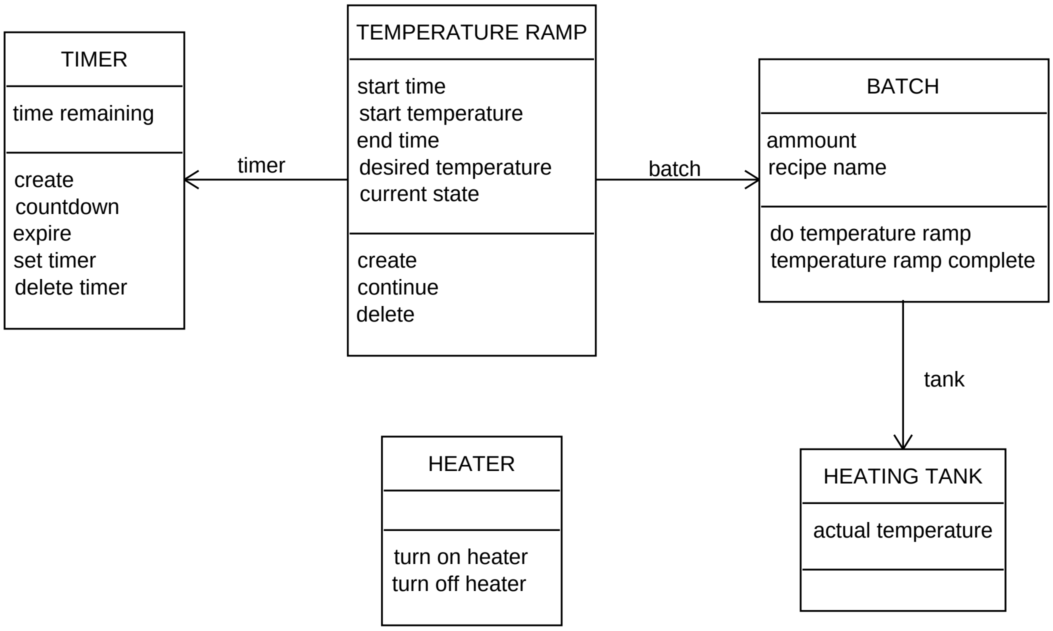


Figure 4.20: Fragment of a class-relationship diagram containing the TEMPERATURE_RAMP object class.

- A temperature ramp is an object created in the store of the controller and used for controlling the period of time that a heater must be turned on.
- A heater is an object in the environment of the controller. The controller does not need to represent a heater as a data store; it merely communicates with it by sending `turn_on` and `turn_off` messages to it.
- A timer is an object created by the controller and updated periodically.
- The clock is an entity in the environment of the controller. The controller does not need remember the state of a clock; when it needs to know the state of the clock, it interrogates it by asking its current time.
- Batches exist in the environment of the controller, but this is not relevant for the controller. For each batch, the controller must remember that it exists and to which tank it is assigned. Hence, it maintains a data store of batch surrogates.

Following Fusion, we may want to indicate on the CRD the separation of classes whose instances exist in the UoD and classes whose instances exist in the system. Note however that some objects may exist in both (e.g. `CUSTOMER` in the bank account system). It may be wiser to show this separation as the difference between data stores and external entities in a transaction DFD (the customer class would appear as external entity and as a data store).

4.3.3 Specification in LCM

Figures 4.25 and 4.26 give an LCM specification of the relevant classes. Some remarks about this formalization are needed:

TEMPERATURE_RAMP object class.
Attributes

- batch. The batch of cans to be filled.
- start_time. The time at which heating starts.
- start_temperature. Temperature of the fluid in the heating tank assigned to the batch at the start time.
- end_time. maximum heating time of the fluid.
- desired_temperature. Desired temperature of the fluid.
- timer. Timer object that counts the time remaining to the next instant that the fluid temperature must be checked.
- current_state. The temperature ramp can be in states CONTROLLING or COMPLETE.

Transitions.

- create_temperature_ramp(r; b, t, T, tim) creates a fresh temperature ramp r, sets the start time of r to the current time of the clock, sets the start temperature to the current temperature of the heating tank assigned to the batch b, sets the end time to t and desired temperature to T. The timer identifier tim is remembered in attribute tt timer and the temperature ramp goes into state CONTROLLING.
- continue checks the current temperature of the heating tank assigned to the batch. It only occurs if the current temperature is lower than the desired temperature and the end time has not yet past.
- delete kills the temperature ramp. It occurs only if the desired temperature has been reached or the end time has past.

Figure 4.21: Sample data dictionary entries of the TEMPERATURE_RAMP.

	start_controlling_ temperature	continue_heating	stop_heating
BATCH	do_temperature_ramp		temperature_ramp_ complete
TEMPERATURE_ RAMP	create	continue	delete
TIMER	create_timer	timer_expires & set_timer	timer_expires & delete_timer
HEATER	turn_on		

Figure 4.22: Transaction decomposition table of some juice controller transactions.

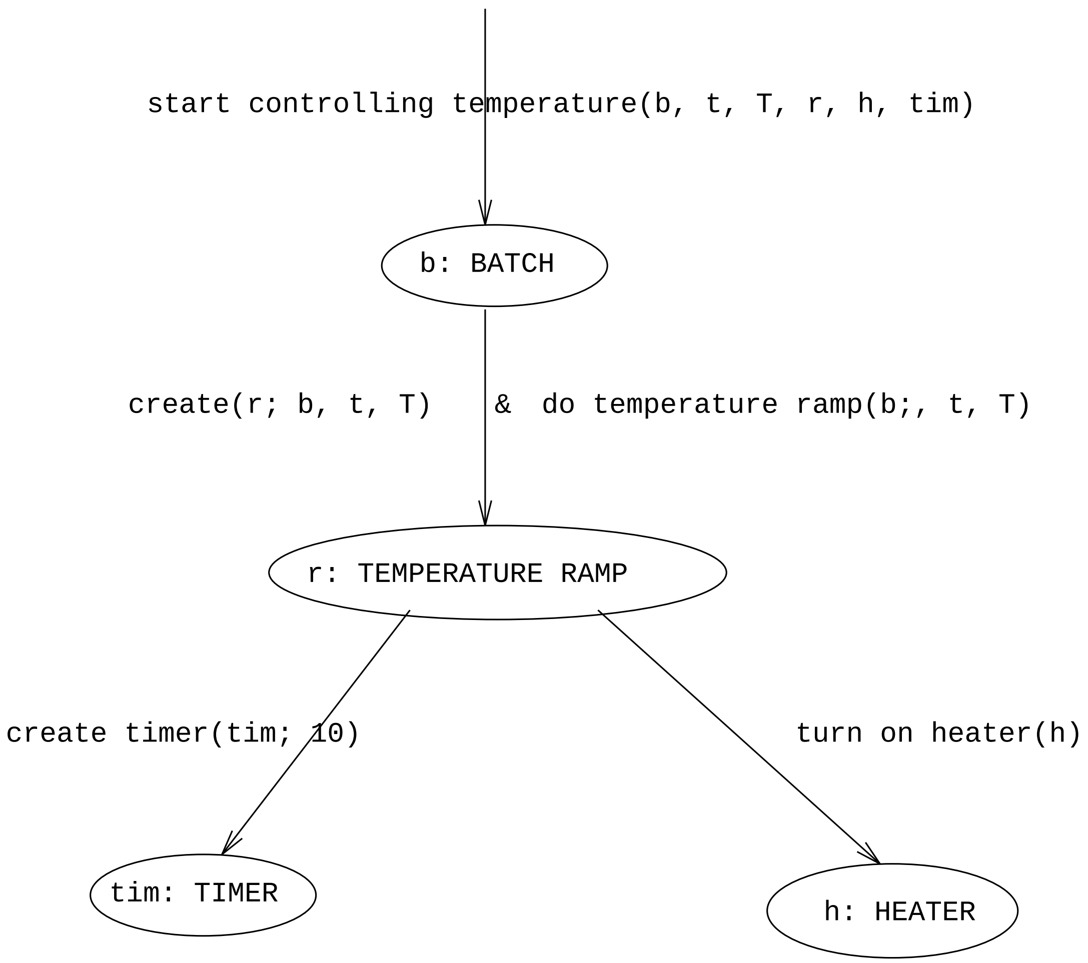
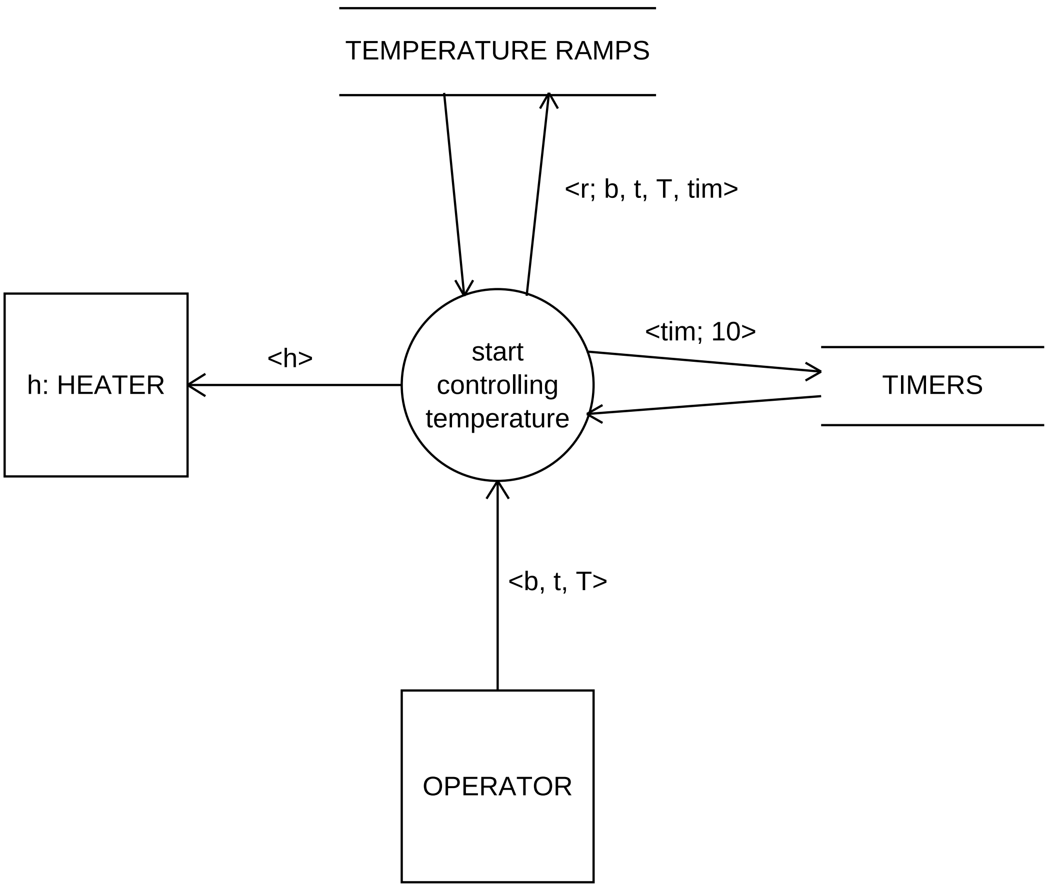


Figure 4.23: Communication flow diagram of the `start_controlling_temperature` transaction of the juice plant controller.



- `do temperature ramp(b; t, T)` is a transition of batch `b` to create a temperature ramp with end time `t` and desired temperature `T`.
- `create(r; b, t, T, tim)` creates a fresh temperature ramp `r`, sets the `start time` of `r` to the current time of the clock, sets the `start temperature` to the current temperature of the heating tank assigned to the batch `b`, sets the `end time` to `t` and `desired temperature` to `T`. The timer identifier `tim` is remembered in attribute tt timer and the temperature ramp goes into state `CONTROLLING`.
- `create timer(tim, 10)` creates a timer with a fresh identifier `tim` and sets the deadline to 10 time units.
- `turn on heater(h)`

Figure 4.24: Data flow diagram of the `start_controlling_temperature` transaction of the juice plant controller.

```
begin object class BATCH
  attributes
    tank : HEATING_TANK;
    amount : NATURAL;
    recipe_name : STRING;
  actions
    do_temperature_ramp(BATCH; TIME, RATIONAL);
    temperature_ramp_complete(BATCH);
  ....
end object class BATCH;

begin object class HEATING_TANK
  specialization of TANK;
  attributes
    actual_temperature : RATIONAL;
  ....
end object class;

begin object class HEATER
  ....
  transitions
    turn_on_heater(HEATER);
    turn_off_heater(HEATER);
  ....
end object class HEATER;

begin object class TIMER
  attributes
    time_remaining : TIME;
  transitions
    create(TIMER, TIME)                    creation;
    countdown(TIMER);
    expires(TIMER);
    set_timer(TIMER);
    delete_timer(TIMER)                    deletion;
  axioms
    forall tim: TIMER, t : TIME::
      [create(tim, t] deadline(tim) = t;

    forall tim: TIMER, t : TIME::
      time_remaining(tim) = t ->
      [countdown(tim)] time_remaining = t-1;

    forall tim: TIMER::
      <expires(tim)> time_remaining(tim) = 0;

    forall tim: TIMER, t: TIME::
      [set_timer(tim, t)] time_remaining(tim) = t;

end object class TIMER;
```

Figure 4.25: Auxiliary specifications for the TEMPERATURE_RAMP object class specification in LCM.

```
begin object class TEMPERATURE_RAMP
  attributes
    batch : BATCH;
    start_time :TIME;
    start_temperature : RATIONAL;
    end_time : TIME;
    desired_temperature : RATIONAL;
    timer : TIMER;
    current_state : STATES;                   --- this is superfluous
  transitions
    create(TEMPERATURE_RAMP; BATCH, TIME, RATIONAL, TIMER)    creation;
    continue(TEMPERATURE_RAMP);
    delete(TEMPERATURE_RAMP)                                  deletion;
  axioms
    forall r : TEMPERATURE_RAMP, b : BATCH, t : TIME, T: RATIONAL,
    tim : TIMER::
      [create_temperature_ramp(r; b, t, T, tim)]
         start_time(r) = current_time(clock) and
         start_temperature(r) = actual_temperature(tank(b)) and
         end_time(r) = t and
         desired_temperature = T and
         timer(r) = tim and
         current_state = CONTROLLING;

    forall r : TEMPERATURE_RAMP ::
         <continue(r)>true ->
           current_state(r) = CONTROLLING and
           actual_temperature(tank(batch(r))) < desired_temperature(r)
           and timer(r) - 10 > 0;

--- The time is checked before it is decremented
    forall r : TEMPERATURE_RAMP, t: TIME ::
         end_time(r) = t -> [continue(r)] end_time(r) = t-10;


    forall r : TEMPERATURE_RAMP, b : BATCH ::
         <delete(r; b)>true ->
           current_state(r) = CONTROLLING and
           b = batch(r) and
           (actual_temperature(tank(batch(r))) >= desired_temperature(r)
            or timer(r) - 10 <= 0);

end object class TEMPERATURE_RAMP;
```

Figure 4.26: Specification of the TEMPERATURE_RAMP object class in LCM.

- Some names have been changed compared. Shlaer and Mellor use alternatively `desired_temperature` and `end_temperature` for the same attribute; we use `desired_temperature` at all places. Shlaer and Mellor use `COOKING_TANK` and `HEATING_TANK` for the same object class at different places in their book; we use `HEATING_TANK` for this class.
- The `tank` attribute of `BATCH` has `HEATING_TANK` as codomain instead of `TANK`, as it has in the example given by Shlaer and Mellor. The reason for this is that `actual_temperature` is not an attribute of `TANK` in general but of `HEATING_TANK`.
- The attributes ending in `_id` have been dropped, because, as explained earlier, in LCM a variable ranging over a class really ranges over the internal identifiers of the class. If needed, the external identifiers can be added though.
- We assumed an object with identifier `clock` and attribute `current_time`. It has not been specified that the `countdown` transition in instances of the `TIMER` class occurs at every tick of the clock. We could do this, but then it cannot be specified that every tick of the clock actually really happens every time step.
- The value types `RATIONAL` and `TIME` are assumed. The value type `STATES` is not declared, but assumed to contain the values `CONTROLLING` and `COMPLETE`.
- The precondition `b = batch(r)` is needed to show the batch identifier `b` to the system transaction of which `delete` is a part.

Figure 4.27 shows some transaction specifications of the juice plant control system.

4.4 Intended semantics

The intended semantics of an LCM specification with action declarations and dynamic logic axioms consists of a process algebra and a Kripke structure. The **process algebra** P is a set of processes that serves as interpretation structure for **process terms**. A process term is a term built from the following ingredients:

- Closed or open action terms.
- Process operators for sequence (.), choice (+) and synchronization (&).

Only finite action terms are interpreted in the process algebra. These represent terminating processes. Variables in open action terms are treated as constants. We assume the graph model of processes, but other models could also be used. The advantage of the graph model is that Mealy machines can be used as graphical representations of the processes. The process algebra interprets equality relations between process terms that hold regardless of the assignment of values to variables. For example, $a_1 \& a_2$ and $a_2 \& a_1$ are interpreted as the same process, because synchronous composition is commutative.

The **Kripke structure** K is a set of possible states, where each state has the same domain, viz. the abstract data type(s) defined in the value specifications. Let Σ be the set of all possible value assignments to variables in the abstract data type(s). There is a function $\rho : \Sigma \to \mathcal{A} \to \wp(\mathcal{K} \times \mathcal{K})$ that for each value assignment assigns a binary relation on K to each process in P, called the **accessibility relation** of the process. The accessibility relation must respect the dynamic logic axioms, and the axioms in the specification, in the usual way. This is worked out in detail in [49]. In addition, there are two minimality properties in the intended semantics, that correspond to the frame and qualification assumptions.

```
begin system function TemperatureRampTransactions
  transactions
      start_controlling_temperature(BATCH, TIME, RATIONAL,
                                    TEMPERATURE_RAMP,
                                    HEATING_TANK, TIMER);
      continue_heating(TIMER, TEMPERATURE_RAMP)
      stop_heating(TIMER, TEMPERATURE_RAMP)
  decompositions

    forall b : BATCH,
           t : time,
           T: RATIONAL,
           r : TEMPERATURE_RAMP,
           h : HEATER,
           tim : TIMER ::
      start_controlling_temperature(b, t, T, r, h, tim) =
        do_temperature_ramp(b; t, T) &
        create(r; b, t, T) &
        create_timer(tim; 10) &
        turn_on_heater(h);

    forall r : TEMPERATURE_RAMP,
           tim : TIMER ::
      continue_heating(tim, r) =
        continue(r) &
        timer_expires(tim) &
        set_timer(tim, 10);

    forall r : TEMPERATURE_RAMP,
           tim : TIMER ::
      stop_heating(tim, r) =
        delete(r; b) &
        timer_expires(tim) &
        delete_timer(tim) &
        temperature_ramp_complete(b)

end system function TemperatureRampTransactions
```

Figure 4.27: Specification of two transactions of the juice controller.

4.5 Discussion

4.5.1 Extensions and variations of dynamic logic

LCM 3.0 only allows actions explicitly declared in some class specification. The utility of the language would be enhanced if we would add the following actions:

- For each formula ϕ we allow the test $\phi?$. This is already done to a limited extent in the account example. Full dynamic logic allows modal tests such as $([\alpha]\phi)?$, which tests if after α, ϕ would be true. In most specifications, non-modal tests would probably suffice.
- For each predicate, the actions $IP(t)$ and $DP(t)$ could be defined to stand for "insert/delete tuple t in/from the extension of P". This has exactly the same formal semantics as the actions $@T(P(t))$ and $@F(P(t))$ in Parnas' Software Cost Reduction (CSR) method [11]. These actions are used in real time system specification to stand for the events that $P(t)$ becomes true/false. Where IP and DP are appropriate for updates to a data store, $@TP(t)$ and $@FP(t)$ are appropriate for events in external entities. The axiomatization in both cases is the same. We can use the axiomatization by Spruit [40]. These axioms could be added to the next version of LCM without problems.
- In addition, Spruit defined a bulk update action $\&X : T(\phi \rightarrow \alpha)$, where $X : T$ is a set of variable declarations. The meaning of the action is that α is performed for all bindings of X for which currently ϕ is true. This is one atomic state transition. Spruit formalized this for the actions IP and DP but we would like to generalize this to arbitrary atomic actions.

Another addition worth considering is the ability to define actions in terms of simpler actions. This has the advantage that frame axioms need not be generated: an operational definition of the meaning of an action does not need frame assumptions over and above those for the elementary actions. Ultimately, only Spruit's elementary actions are needed and the frame assumption for these has been axiomatized. A difficult point here is the process definition language. Spruit uses a regular process definition language, but this may not be powerful enough. Unrestricted use of recursion in process terms however is notoriously difficult. If we would use a transaction definition language that allows recursion, then we come close to the way transactions are defined as macrosteps in Statemate [24, 20, 30, 31]. This deserves further investigation.

4.5.2 Life cycle inheritance

Since object life cycles can be defined for object classes as well as mode classes, each mode can have its own behavior. However, there are consistency constraints on life cycles of a class and its mode classes. If a supertype and a subtype both have significant state models, then OOA distinguishes the static subclass from the mode case.

- If instances cannot migrate between subtypes, then OOA recommends defining state models for each subtype and factoring out the common part of these state models as state model of the supertype.
- If instances can migrate between subtypes, then OOA suggests two ways to specify this.
  - Define one state model for the supertype, that contains the state models of all subtypes and shows how an object moves from one subtype to another subtype.
  - Define state models for each of the subtypes, that start and end with type migration events.

This is not illustrated by Shlaer and Mellor. These heuristics agree with the suggestion given elsewhere that life cycle inheritance for static subclasses is downward, and that it is upward for modes [47]. Life cycle inheritance is an active research topic that deserves extra investigation in the context of TCM.

4.5.3 The synchronicity assumption

The **synchronicity assumption** in TCM/LCM consists of two elements:

- system transactions take no time and
- the system produces the response to an event instantly (in no time).

Because a system transaction consists of object transactions, these satisfy the synchronicity assumption as well.

The synchronicity assumption is called the *synchrony hypothesis* by Berry [4] and Harel [19]. We will call it the synchronicity *assumption*, because it is not a hypothesis to be tested but a simplifying assumption about the system being specified. The assumption is justified as long as the implemented system responds sufficiently fast to incoming events as compared to its environment, that we can neglect the reaction speed. The synchronicity assumption is satisfied by any abstract system whose actions take no time. Assuming that events and responses are incidents that take no time, we can maintain the statement that the event causes the response but that yet the response occurs simultaneously with the response. The synchronicity assumption is adopted in the semantics of formal languages ESTEREL [4] and Statemate [24, 20], as well as by semiformal methods such as the Ward-Mellor method [41, 42, 29] and the Hatley-Pirbhai method [21].

4.5.4 Formal specification in LCM — dynamic logic and process algebra

An alternative formalization in LCM is to use process algebra to define the state model of a class by means of a recursive process specification. The `transitions` section of a specification declares constants of sort `ACTION`, which denote elementary actions. A **process term** is a term built from action constants using operators for sequence (`.`), choice (`+`) and parallel composition (`&`). A term without parallel composition is called a **basic process term**. A **recursive process specification** is a set of equations that must be solved in a particular model. Under certain conditions and in certain models, recursive process specifications have a unique solution [2]. Following the synchronicity assumption, we formalize an object transaction as a synchronous composition:

$$\phi?\&e\&a.$$

Again, causality is *not* represented here. This gives us the specification in figure 4.28.

It is still unclear in this representation how the variables in the process equations are bound. If all variables in one equation are bound by a universal quantifier, then every occurrence of each variable in that equation refers to the same value. If there is one quantifier for all equations, this should hold for all equations. This is correct in the case of `a`, which should refer to the same account identifier through the equations. However, it is incorrect in the case of the variable `b`, which can be changed by the actions. It is also incorrect for the variables `aid` and `cid`, which may be different at every occurrence of the events. However, once these two variables are bound in a transaction, they should have the same value in the test and the action of this transaction. This is not represented in the process equations. Until these problems are solved, the dynamic logic representation seems the better option to pursue.

```
begin object class ACCOUNT
  functions
    fee : MONEY;
  attributes
    account_ID : NATURAL:
    balance : MONEY                              initially 0;
    customer_ID :NATURAL;
  identifier
    account_ID;
  transitions
    create(ACCOUNT; NATURAL, NATURAL)            creation;
    increase_balance(ACCOUNT; MONEY);
    decrease_balance(ACCOUNT; MONEY);
    accept_check(ACCOUNT; MONEY);
    refuse_check(ACCOUNT; MONEY);
  life cycle
    forall a : ACCOUNT ::
    ACCOUNT(a) = open_account(a; cid, aid) & create(a; aid, cid) .
                 OK(a; aid, cid, b);
    OK(a; aid, cid, b) =
      deposit(a; m) & increase_balance(a; m) . OK(a; aid, cid, b) +
      withdraw(a; m) & (m <= balance(a))? & decrease_balance(a, m) .
      OK(a; aid, cid, b) +
      submit_check(a; m) & (m <= balance(a))? & accept_check(a, m) .
      OK(a; aid, cid, b) +
      submit_check(a; m) & (m > balance(a) and fee <= balance(a))? & refuse_check .
      OK(a; aid, cid, b) +
      submit_check(a; m) & (m > balance(a) and fee < balance(a))? & refuse_check .
      KO(a; aid, cid, b);
    KO(a; aid, cid, b) =
         deposit(a; m) & (m + balance(a) >=0)? & increase_balance(a, m) .
         OK(a; aid, cid, b) +
         deposit(a; m) & (m + balance(a) < 0))? & increase_balance(a, m) .
         KO(a; aid, cid, m) +
         submit_check(a; m) & refuse_check(a, m) . KO(a; aid, cid, b);

  axioms
    -- effect axioms
    forall a : ACCOUNT , aid, cid : NATURAL ::
      [create(a; aid, cid)] customer_ID(a) = cid and
                            account_ID(a) = aid and
                            balance(a) = 0;
    forall a : ACCOUNT , b, m : MONEY ::
      balance(a) = b -> [increase_balance(a; m)] balance(a) = b+m;
    forall a : ACCOUNT , b, m : MONEY ::
      balance(a) = b -> [decrease_balance(a; m)] balance(a) = b-m;
    forall a : ACCOUNT , b, m : MONEY ::
      balance(a) = b -> [accept_check(a; m)] balance(a) = b-m;
    forall a : ACCOUNT , b, m : MONEY ::
      balance(a) = b -> [refuse_check(a; m)] balance(a) = b-fee;

end object class ACCOUNT;
```

Figure 4.28: Specification of the ACCOUNT object class in LCM, using process algebra extended with tests.

Chapter 5

The process model

5.1 The OOA notation

An OOA process model consists of the following components:

- **Action Data Flow diagrams** (ADFDs) that represent the processing performed during one action.
- For each process in an ADFD, a **process descriptions** must be produced.
- From all ADFDs, an **object access model** (OAM) can be constructed. This does not add new information, but it explicitly shows the accesses made by an ADFD to data stores of other object types.
- More summary information can be produced: **State process tables** show which processes are used in which actions. This does not contain new information.

We discuss ADFDs and the OAM in more detail.

5.1.1 Action data flow diagrams

An ADFD is a network of data stores and processes connected by directed arcs. It specifies the work done by the system when the action is executed. There is one ADFD for each action (and hence for each state in the Moore convention). Figure 6.2.4 [39, page 116] contains an ADFD for the `Created` state of figure 6.2.2 from [39, page 114]. An ADFD has the following components:

- **Data stores**, represented by two parallel lines. There are three kinds of data stores.
  - An **object data store** contains all (currently existing) instances of one object class.
  - A **timer data store** contains the current state of a timer object.
  - The **current time data store** contains the current time.

  For typographical convenience, a data store may be repeated at different places in an ADFD. Figure 6.2.4 [39, page 116] contains object data stores for three object classes.
- **Processes** represent separate units of computation. They are represented by bubbles. Each process is assigned to an object class. This assignment is reflected in the process identifier. In figure 6.2.4 [39, page 116], the following identifiers have been chosen': TR for Temperature ramp, CT for Cooking Tank, and B for Batch. There are four types of processes.

- An **accessor process** is a process whose sole purpose is to create, read, update or delete an instance of an object in an object data store. TR.1, TR.4, B.1, CT.1 and TIM.3 in figure 6.2.4 [39, page 116] are accessor processes.
- An **event generator** is a process that produces exactly one event as output. TR.3 in figure 6.2.4 [39, page 116] is an event generator.
- A **transformation** is a process whose purpose is one of communication or transfer of data.
- A **test** is a process that performs a test and makes the result available to a conditional control flow.

- **Data flows** represent the flow of information in the action, and are represented by arrows. There are three kinds of data flows.
  - An **event data flow** points into a process from nowhere or points from a process to nowhere. In the first case, it is an input event data flow and it must be labeled by attributes of the event that triggered the action and needed by the process. For example, the event data flow into process TR.1 is labeled by all event attributes of the triggering event (those attributes are shown in the state model of figure 6.2.2 from [39, page 114] The event data flow into process B.1 is labeled with only one attribute. If an event data flow points out of a process to nowhere, it is an event generated by the action corresponding to the entire ADFD, and it must be labeled by the same event as the generated event in the state model. For example, the event data flow out of process TR.3 is labeled by the generated event of the state model in figure 6.2.2 from [39, page 114].
  - A data flow between two processes is labeled by the names of the attributes that it carries. An attribute may have two names, one that takes the perspective of the sending process and one that takes the perspective of the receiving process (e.g. the flow from process CT.1 to TR.4). If only one name is given, as in the flow from B.1 to CT.1, then the attribute is shared between the two objects. This, the flow from B.1 to CT.1 is equivalent to the annotation Batch.Tank ID = Cooking Tank . Tank ID
  - **Control flows** are represented by dashed lines and are used to specified precedence of process execution, and choice between process executions. Control flows that indicate choice are called *conditional.* Control flows are not shown in figure 6.2.4 [39, page 116].

5.1.2 Object access model

The **object access model** shows accesses between objects performed by accessor processes. If the ADFD of an action in the state model of one class accesses the object data store of another class, then the object access model shows an arrow from the first to the second class. For example, figure 6.8.1 [39, page 131] shows an object access model generated from the ADFD of figure 6.2.4 [39, page 116]. Object classes are represented by ovals. An arrow points from the accessing class to the accessed class. each arrow is labeled by the accessor process. Note that the arrow does not say anything about the direction of the flow of information. Rather, it indicates the direction of initiative, i.e. it points from the initiator of the access to the class who suffers the access.

5.2 Specifying the process model in TCM

The process model specifies the preconditions and effects of actions and specifies events generated by the action. These are already specified in TCM semi-formally by the data dictionary entries and transaction decomposition tables and formally by the LCM specification. We have seen that it is instructive to add *transaction data flow diagram* to the semi-formal documentation, which shows the data flows in an entire system transaction. One such transaction DFD can be drawn for each transaction.

5.3 Discussion

The process model can be dispensed with because it gives an operational way to perform the computations necessary to perform a system transaction. The reasons for the simplification as we move from ADFDs to TDFDs are the following:

- The Shlaer-Mellor method does not use the transaction concept as a structuring principle.
- Transactions are modeled in TCM using the synchronicity assumption.
- The effect of a transaction is defined by defining the effect of the local events within the transaction. Since these local effects must be local, and different local events in one transaction belong to different objects, these can be conjoined without problems. Note however that local events may have global preconditions. This is not a problem, for all these local preconditions are tested in the initial state of the transaction, and testing them does not update the state of any object.

Note that the representation of synchronous communication by the OAM in OOA is incomplete:

- Shlaer and Mellor ignore the possibility of communicating with another instance of the *same* object class. For them, communication is communication with an instance of a different object class. For example, the OAM only represents accesses to data stores of *other* classes.
- Each process in an ADFD is allocated to an object. This means that an action executed by an instance of C may involve processes allocated to objects of different instances of C, or to instances of different classes. Shlaer and Mellor forget to mention that data flows between processes in one ADFD that are allocated to different classes are synchronous communications too.

Chapter 6

The object communication model

6.1 The OOA notation

The **object communication model** (OCM) shows message sending between objects (figure 5.1.1 from [39, page 86]). The OCM is represented as a directed graph in which the nodes are external entities or individual objects (instances), and the arrows are events sent by one node to another. The graph can be constructed from the state models and summarizes information already present in the state models.

In OOA, events are assumed to arrive at the object to which they are directed at the instant in which they are generated [39, page 107]. Actions are atomic in the sense that they cannot be broken down in subactions [39, pages 45, 105], but they are not atomic in time. That is, actions are assumed to take time. Now, if the receiver is ready to process the event, the event will be processed immediately; but as we have just seen, this takes time, and during this time, the receiver cannot process other events. If the receiver is not ready to process an event, the event will wait. Events do not get lost [39, page 107], i.e. they are always received. However, there is no explicit mention of a *fairness assumption*: an event may wait to be processed for an indefinitely long time. OOA assumes *partial arrival order nondeterminism*: multiple events generated by the same object will be processed in the order in which they are sent, but events sent by different objects may be processed in any order.

6.2 Specifying the communication model in TCM

The communication model in TCM is simpler, because only synchronous communication is assumed, and because system and object transactions are modeled using the synchronicity assumption: transitions take no time, and actions occur simultaneous with the events that trigger them. The event/action pairs that occur in the Mealy state models, and that give rise to the OCM, thus appear as part of the transaction decomposition tables. We have seen that these can also be depicted graphically in a kind of simplified object communication diagram, that we called a *transaction communication flow diagram*, that shows the flow of initiative during a transaction. Figure 4.23 gives a transaction communication flow diagram of the `start_controlling_temperature` transaction of the juice plant controller. A transaction CFD does not add any formal information to the specification, but is a useful supplement to the transaction decomposition table.

Note that the synchronous communications shown in the object access model (OAM) appear as either read actions in the transaction DFD or must appear as local events inside a transaction, in which case they will appear in the transaction CFD.

6.3 Discussion

Asynchronous communication can be modeled in TCM by introducing a postbox object for each asynchronous communication. The postbox process communicates synchronously with senders and receivers, and it should implement arrival order nondeterminism. However, the assumption of partial arrival order nondeterminism causes problems: If two timers send a timeout to an object simultaneously, one of these must wait, which defeats the purpose of the timeout. Timeouts have the essential and annoying property that you cannot postpone them. Furthermore, if two timers send timouts to the same object at different moments, then these may be processed in a reverse order! This seems fundamentally wrong too. These problems can be avoided by adopting the synchronicity assumption and doing away with postboxes altogether.

Chapter 7

Results and conclusions

7.1 Summary of results

Our comparison of OOA with TCM has yielded the following results:

- The information model has about the same modeling power as the class model in TCM, except that mode classes and roles are not represented in the information model, and the cardinality constraints are very limited in the information model. OOA does not allow relationships to be specialized, as TCM does.
- OOA uses the Moore convention to represent state models. This can be transformed into a Mealy representation. All transitory states can be removed from the Mealy representation, at the price of introducing nondeterminism. The Mealy machines can be represented in TCM by separating the local behavior (preconditions and local transitions), which are represented by life cycles, from communication information, which is represented by transaction decomposition. In this move, asynchronous communication is replaced by the synchronicity assumption.
- The process model can be simplified to transaction DFDs. These give some useful information about transactions.
- The communication model can be simplified to transaction CFDs. These give some useful information about transactions. TCM uses the synchronous model of communication, where OOA uses asynchronous communication.

7.2 Extensions of TCM

Our analysis of OOA has led to a number of possible extensions of TCM:

- Complexity reduction by means of domains and subsystem is useful. TCM should contain some such technique, although it is still too early to decide upon a particular technique.
- Documenting transactions by means of transaction DFDs and CFDs is a useful extension of TCM.

7.3 Changes to LCM

Our analysis has also suggested a number of changes to LCM, that should make the gap between LCM and traditional, well-known methods less wide:

- Terminological changes:

old	new
surjection	mandatory
injection	unique
relationship	link
relationship class	link class
relationship class	relationship
events	transitions
service	system functions
static constraints	state axioms
dynamic constraints	transition axioms

- There are no life cycle definitions in the suggested new version of LCM. Life cycles are defined diagrammatically and by means of dynamic logic axioms using the `current_state` attribute.

7.4 Further research

Our analysis has also yielded some topics for further research:

- Transaction definition could be made more powerful by introducing tests $\phi?$, choice, bulk updates $X : T :: \phi \to \alpha$ and defined actions, using atomic inserts and deletes.
- The inheritance of life cycles for static partitions and for modes should be investigated. An intriguing possibility is to use the substates of state charts to represent modes. Mode changing transitions would then be transitions among superstates, that each have their own life cycle. There is existing research in this area on which we can build [10, 28, 33, 35, 47].
- The specification of actions to be performed upon entering or leaving a state should be investigated.
- The semantics of state charts should be raided for proposals for macrostep semantics, to see if these can be used for transaction semantics in LCM. The assumption that each microstep in a macrostep is executed by a different object could make the macrostep semantics trivial.

We have not discussed methodological issues in this report. The major methodological issue is the concept of thread of control and its use in the simulation of system behavior in OOA [39, pages 94–104]. Some of the representations used in OOA simulations could be useful for the representation of the results of reachability analysis of TCM specifications.

Bibliography

[1] S. Austin and G.I. Parkin. Formal methods: a survey. Technical report, Division of Information Technology and Computing, National Physics Laboratory, Teddington, Middlesex United Kingdom, 31 March 1993.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[3] S. Bear, P. Allen, D. Coleman, and F. Hayes. Graphical specification of object-oriented systems. In N. Meyrowitz, editor, *Object-Oriented Programming: Systems, Languages and Applications/European Conference on Object-Oriented Programming*, pages 28–37, Ottawa, October 1990. ACM. Sigplan Notices 25, number 10.

[4] 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.

[5] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press/Prentice-Hall, 1990.

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

[7] J.F. Costa, A. Sernadas, and C. Sernadas. *OBL-89 User's Manual, version 2.3*. Instituto Superior Técnico, Lisbon, May 1989.

[8] D. Craigen, S. Gerhart, and T. Ralston. An international survey of formal methods. volume 1: Purpose, approach, analysis, and conclusions. Technical Report NISTGCR 93/626, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, 1993.

[9] D. Craigen, S. Gerhart, and T. Ralston. An international survey of formal methods. volume 2: Case studies. Technical report, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, 1993.

[10] J. Ebert and G. Engels. Structural and behavioural views on omt-classes. In S. Urban E. Bertino, editor, *Proceedings of the International Conference on Object-Oriented Systems, Methodologies, and Applications (ISOOMS'94)*, pages 142–157, Palermo (Italy), September 21-23 1994. Lecture Notes in Computer Science 858.

[11] S.R. Faulk and P.C. Clements. The NRL software cost reduction (SCR) requirements specification methodology. In *Fourth International Workshop on Software Specification and Design*, pages 102–107. Computer Society Press, April 3–4 1987.

[12] R.B. Feenstra and R.J. Wieringa. Translating LCM specifications into reachability calculus. In preparation.

[13] R.B. Feenstra and R.J. Wieringa. LCM 3.0: a language for describing conceptual models. Technical Report IR-344, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1993.

[14] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, pages 21–28, January 1994.

[15] J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Operational semantics of order-sorted algebra. In W. Brauer, editor, *Proceedings, 1985 International Conference on Automata, Languages and Programming*, pages 221–231. Springer, 1985. Lecture Notes in Computer Science, Volume 194.

[16] J.A. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[17] 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.

[18] J.A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Laboratory, 333 Ravenswood Ave., Menlo Park, CA 94025, U.S.A., 1988.

[19] D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer, 1985. NATO ASI Series.

[20] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings, Symposium on Logic in Computer Science*, pages 54–64. Computer Science Press, June 22–25 1987.

[21] D. Hatley and I. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House, 1987.

[22] F. Hayes and D. Coleman. Coherent models for object-oriented analysis. In A. Paepcke, editor, *Object-Oriented Programming: Systems, Languages and Applications/European Conference on Object-Oriented Programming*, pages 171–183. ACM, 1991. Sigplan Notices 25, number 11.

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

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

[25] I. Jacobson, M. Christerson, P. Johnsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Prentice-Hall, 1992.

[26] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The Troll Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.

[27] R. Jungclaus, R.J. Wieringa, P. Hartel, G. Saake, and Thorsten Hartmann. Combining Troll with the Object Modeling Techique. In B. Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen. GI-Fachgespräch FG 1: Integration von semi-formalen und formalen Methoden für die Spezifikation von Software*, pages 35–42. Springer, Informatik aktuell, 1994.

[28] A. Lopes and F. Costa. Rewriting for reuse. In *Proceedings ERCIM Workshop, Nancy, November 2-4*, pages 43–55. INRIA, 1993.

[29] S.J. Mellor and P.T. Ward. *Structured Development for Real-Time Systems.* Prentice-Hall/Yourdon Press, 1985. Volume 3: Implementation Modeling Techniques.

[30] A. Pnueli and M. Shalev. What is in a step. In J.W. Klop, J.-J.Ch. Meyer, and J.J.M.M. Rutten, editors, *J.W. de Bakker, 25 Jaar Semantiek. Liber Amicorum*, pages 373–399. Stichting Mathematisch Centrum, 1989.

[31] 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.

[32] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design.* Prentice-Hall, 1991.

[33] G. Saake, P. Hartel, R. Jungclaus, R.J. Wieringa, and R.B. Feenstra. Inheritance conditions for object life cycle diagrams. In U.W. Lipeck and G. Vossen, editors, *Formale Grundlagen für den Entwurf von Informationsystemen*, pages 79–88. Institut für Informatik, Universität Hannover, Postfach 6009, D-30060, Hannover, May 1994. Informatik-Berichte Nr. 03/94.

[34] J. Scheerder, P.A. Spruit, and R.J. Wieringa. Built-in value type specifications in LCM 3.1. Technical Report IR-386, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, July 1995.

[35] M. Schrefl. Behavior modeling by stepwise refining behavior diagrams. In H. Kangassolo, editor, *Entity-Relationship Approach: The Core of Conceptual Modelling*, pages 119–134. North-Holland, 1991.

[36] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[37] S. Shlaer and S.J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice-Hall, 1988.

[38] S. Shlaer and S.J. Mellor. An object-oriented approach to domain analysis. *ACM SIGSOFT Software Engineering Notes*, 14(5):66–77, July 1989.

[39] S. Shlaer and S.J. Mellor. *Object Lifecycles: Modeling the World in States*. Prentice-Hall, 1992.

[40] P.A. Spruit. *Logics of Database Updates*. PhD thesis, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1994.

[41] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems*. Prentice-Hall/Yourdon Press, 1985. Volume 1: Introduction and Tools.

[42] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems*. Prentice-Hall/Yourdon Press, 1985. Volume 2: Essential Modeling Techniques.

[43] R.J. Wieringa. Requirements engineering: Semantic, real-time, and object-oriented methods. Course notes.

[44] R.J. Wieringa. A method for building and evaluating formal specifications of object-oriented conceptual models of database systems (MCM). Technical Report IR-340, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1993.

[45] R.J. Wieringa. *Requirements Engineering: Frameworks for Understanding*. Wiley, 1996. To be published.

[46] R.J. Wieringa and W. de Jonge. Object identifiers, keys, and surrogates. *Theory and Practice of Object Systems*, 1993. To be published.

[47] 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.

[48] R.J. Wieringa, R. Jungclaus, P. Hartel, G. Saake, and T. Hartmann. OMTROLL — Object Modeling in Troll. Proceedings of the International Worskshop on Information Systems — Correctness and Reusability (IS-CORE'93), Udo W. Lipeck and G. Koschorrek (eds), pages 267–283. Institut für Informatik, Universität Hannover, Postfach 6009, D-30060, Hannover., September 1993.

[49] 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.