XIX. LCM and MCM

Specification of a Control System using Dynamic Logic and Process Algebra

Roel Wieringa

Vrije Universiteit Amsterdam

Abstract

LCM 3.0 is a specification language based on dynamic logic and process algebra, and can be used to specify systems of dynamic objects that communicate synchronously. LCM 3.0 was developed for the specification of object-oriented information systems, but contains sufficient facilities for the specification of control to apply it to the specification of control-intensive systems as well. In this paper, the results of such an application are reported. The paper concludes with a discussion of the need for theorem-proving support and of the extensions that would be needed to be able to specify real-time properties.

19.1 Introduction

LCM 3.0 (Language for Conceptual Modeling version 3.0) is a formal language developed for the specification of the external behavior of data-intensive systems [9]. Examples of data-intensive systems are business information systems and database systems. LCM is based on a combination of dynamic logic and process algebra, and contains features to specify control structures as well as data structures. LCM comes with a method for conceptual modeling (MCM), which provides a set of heuristics to find models of external system behavior, and for validating the quality of these models [16][17]. Methodologically, MCM is closely related to the Jackson System Development (JSD), which has been applied to the specification of control-intensive systems [12]. Examples of control-intensive systems are in-

dustrial process control systems and robot control systems. This paper is intended to show that LCM/MCM is suitable for the specification of control-intensive systems as well. However, it will become clear from this paper that the utility of LCM/MCM would be enhanced if automated support would be provided for reachability analysis of systems specified in LCM, and if LCM would be extended with constructs to deal with real time and with exceptions such as device failure. It will be argued in this paper that safety analysis is a special case of reachability analysis.

In section 19.2, MCM is explained by applying it to the development of a model for the production cell control system. Section 19.3 introduces LCM 3.0, again using the production cell control system as running example. Section 19.5 discusses the lessons learned from this application, and section 19.6 concludes the paper with a list of topics for further work.

19.2 Method for Conceptual Modeling (MCM)

MCM is a method to produce formal and informal conceptual models of observable system behavior. An important reason why MCM produces both formal and informal models is that MCM is designed with the aim of allowing both the possibility of formal *verification* of an implementation against a conceptual model, and of *validation* of the conceptual model against informal requirements that arise from discussions with the customer. The goal of verification is to show that a system specified in one way has the same behavior as a system specified in another way. The goal of validation is to show that a specification conforms to the intentions of the domain specialists. In other words, verification is concerned with the question whether we implement a system right, whereas validation is concerned with the question whether we model the right system.

Verification can proceed by formal proof or by testing; validation is essentially an informal affair, because the intentions of the domain specialists are themselves informal. In MCM in order to make formal verification possible, the conceptual model is specified in a formal language (LCM 3.0). In addition, in order to make validation possible, the model is specified informally, by means of diagrams and structured text. The informal presentation techniques have a precise correspondence with the formal specification.

Another design aim of MCM is to integrate useful elements of existing methods, ranging from Entity-Relationship modeling and Data Flow modeling to object-oriented modeling techniques. One reason for this is that there should be

progress in system development methods. Inventing something new every five years does not constitute progress. Instead, we should take whatever is good from existing methods and try to improve on it. Another reason for this approach is that the acceptance of a method is likely to be increased when it uses techniques that are familiar to practitioners. This is incidentally an additional reason why formal and informal methods are combined in MCM: it enhances the usability of the formal method.

MCM models a system as consisting of a collection of communicating objects. Each object has a local state and a local behavior. Objects communicate with each other by means of synchronous communication events. Objects are subject to integrity constraints, which are static or dynamic constraints on allowable object states or behavior. Constraints may be local to one object or global to the entire system. In a data-intensive system, constraints represent business rules. In a control-intensive system, they can be used to express safety constraints.

When we model a data-intensive system in MCM, we first make a model of the *universe of discourse* (UoD) of the system, which is the part of the world about which the system registers data. For example, in a model of a library database system, we would model the UoD as a set of objects such as DOCUMENT, MEMBER, RESERVATION and LOAN instances. A borrow event in the UoD would be a synchronous communication event that in one atomic state transition deletes a RESERVATION object and creates a LOAN object. UoD objects would be represented in the system by records, that act as *surrogates* for the corresponding UoD objects. The state of the surrogates is updated whenever the corresponding UoD objects change state. For example, the borrow event in the UoD would be registered by a database transaction, that deletes a RESERVATION record and creates a LOAN record in one atomic state transition. Thus, each database system transaction corresponds to an event in the UoD, in which one or more UoD objects participate. Database systems are essentially registration systems.

The basic idea to transfer and extend this to a method for modeling control systems is to model the UoD as a set of communicating devices, each of which is represented in the system by a device surrogate. In real-time methods, such a surrogate is often called a *virtual device* [11]. Devices communicate by participating in a synchronous event. The control function of the system is provided by defining control objects in addition to the device objects. A control object does not correspond to a UoD object. It encapsulates device communications and enforces the required behavior of the devices. The problem of controlling devices in the UoD is thus reduced to the problem of defining the required synchronizations be-

tween the devices. This idea will become clear by the illustrations in the following paragraphs, taken from the production cell control system.

19.2.1 The class model

As explained above, one class is defined for each device in the UoD. It turned out to be convenient to define one class for each sensor device and for each actuator device. Examples of sensor object classes are TABLE_SWITCH, TABLE_POTMETER, and ARM1_POTMETER. Examples of actuator object classes are TABLE_H_MOTOR, TABLE_V_MOTOR and ARM1_MOTOR. Each object class has only one existing instance in the system.

In addition to the device objects, control objects are defined, that enforce the required behavior of the devices. Corresponding to the modular structure of the production cell system, the control system contains the following control objects: TABLE_CONTROL, ARM1_CONTROL, ARM2_CONTROL, ROBOT_CONTROL, PRESS_CONTROL and CRANE_CONTROL. Again, these are objects classes that each have exactly one existing instance. There are no control objects for the conveyor belts, because it is assumed that they move continuously. This means that it is also assumed that the blanks are spaced on the feed belt with sufficient distance so that the robot and rotary table have the time to return to the positions in which they are ready to receive the next blank from the feed belt.

The central component of models of data-intensive systems is a class diagram, which usually is some form of enhanced Entity-Relationship diagram. In the case of control-intensive systems, such a diagram is usually quite simple. Figure 1 shows a fragment of the class diagram representing the table switch and and the table control object classes.

A class is represented in Coad & Yourdon style [6] by a rectangle partitioned into three areas, listing, from top to bottom, the class name, names of attributes and predicates applicable to class instances, and names of events applicable to class instances. By convention, a predicate name starts with an upper-case letter and an attribute name consists only of lower-case letters. The TABLE_SWITCH object has three predicates, Exists, Table_lower_position and Table_upper_position, and two events, table_lower_position and table_upper_position. No object in the model has attributes.

Formally, the extension of a class is the set of all possible identifiers (oids) of the class instances. All predicates, attributes and events declared in the class are applied to oids. Each class has at least the Exists predicate, which is set to true when the object starts its existence and set to false when it ceases to exist. All in-

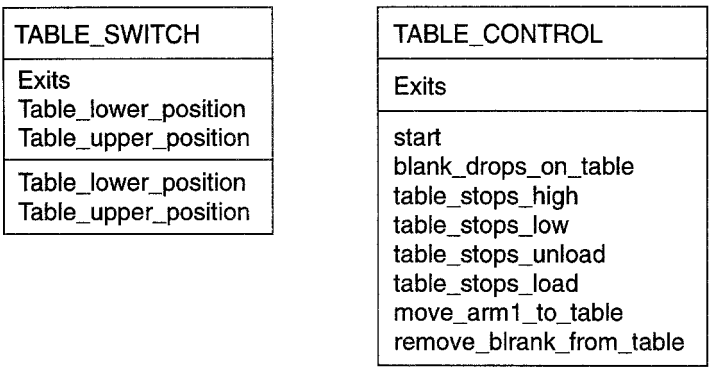

Figure 1 Class diagram of the table switch and the table control

tegrity constraints only concern existing class instances. The ability to create and delete objects is essential in data-intensive systems, but tends to be less important in control-intensive systems. In the production cell example, all classes have only one instance that eternally exists, and no object is ever created or deleted. One way to introduce objects that are dynamically created and deleted would be to model each incoming blank as an object. However, to express the parallelism between the different device and control objects, we must model these objects as separate processes anyway. Defining a BLANK object class then does not add any information, even in the case that different blanks would require a different treatment by the production cell. (Different treatments of different kinds of blanks could be specified by adding tests and choices to the specification of control object behavior.)

Returning to the class specifications in Figure 1, the Table_lower_position predicate in the TABLE_SWITCH class is needed to be able to specify a safety constraint.

MCM allows the specification of attributes of objects and of relationships between them. A relationship can itself have attributes and behavior. In addition, there is a special *is_a* relationship, that expresses that one class is a subclass of another one, and that defines inheritance of attributes, behavior and constraints from the superclass to the subclass. In our production cell model, there are no attributes and no relationships and there is no taxonomic structure — or more accurately, the attributes, relationships and taxonomic structure that the actual devices and control objects have in the real world, are not represented in the model. Attributes and relationships are typically needed in data-intensive systems to be able to store the necessary data to answer queries. In control-intensive systems, all data needed to be able to perform the control function is usually present in the state of the object life cycles.

The events in the TABLE_SWITCH class are events generated by this sensor. As explained below, these events are forced by the TABLE_CONTROL object to synchronize with other events in the UoD, such as table_stop_v. All events in the TABLE_CONTROL class are synchronization events between events in the UoD.

The life cycle model

The behavior of an object is called an *object life cycle* in MCM and we will follow this practice here. Figure 2 shows two equivalent representations the life cycle of the table control object. The start event is a synchronization event between control objects that is needed because of different speeds with which the parts of the system move. When a blank drops on the table, the control object tells the table motors to move upward and right, until the table reaches the top position and has the direction needed to be unloaded by arm1. It then tells arm1 to move to the table. When arm1 picks up the blank from the table, the table is moved downward and returned to its starting position.

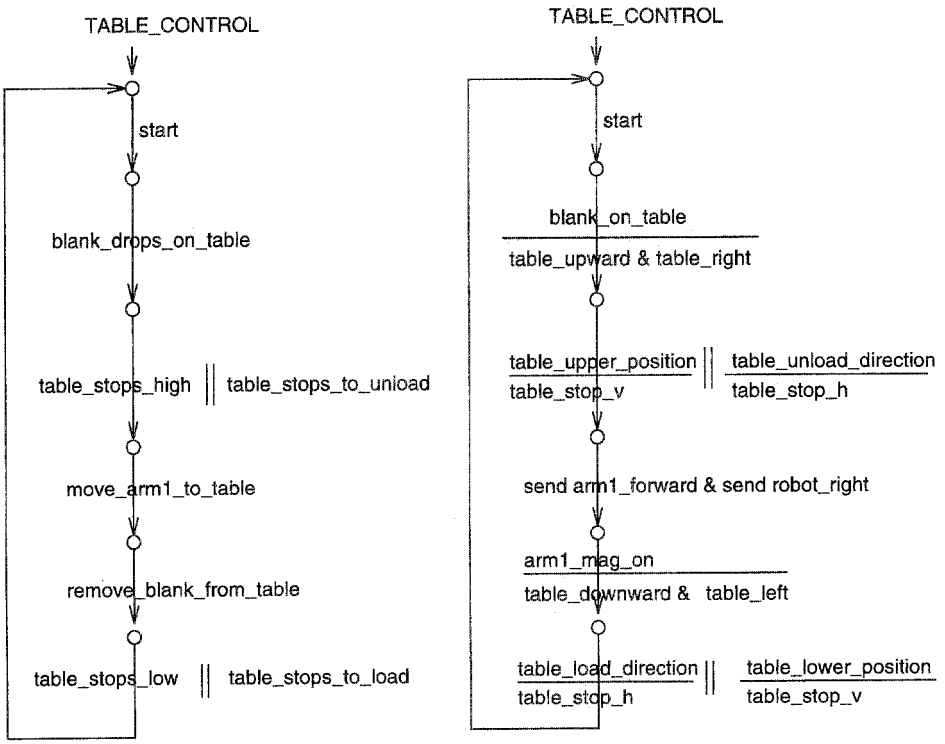

Figure 2 Life cycle of table control

The notation a ‖ b for two events a and b stands for the process ab + ba (a choice between the sequential processes ab and ba). More in general, we can label an arrow in a life cycle diagram with a multiset of processes. A transition along that arrow is then equivalent to a parallel execution of the processes in this multiset. These life cycle diagrams are also called *recursive process graphs* and are defined formally in [13].

On the right-hand side in figure 2, each event in the control life cycle has been replaced by the events that it synchronizes. The notation

$$\frac{\text{blank\_on\_table}}{\text{table\_upward \& table\_right}}$$

is a fancy way of writing blank_on_table & table_upward & table_right, used to express informally that blank_on_table triggers the other two events. This notation is similar to the well-known stimulus/response notation in Mealy machines [14]. The difference between stimulus and response is not represented in the formal specification; all that is represented is that certain events occur synchronously. Stimulus/response pairs have thus been modeled using Esterel's *synchrony hypothesis*, which says that the response to a stimulus occurs simultaneously with the stimulus [2][5].

The communication structure of the model can be shown by means of context diagrams and by a transaction decomposition table. For each control object, we can draw a *context diagram* such as the one shown in Figure 3.

Just like in data flow (DF) diagrams [22], the system of interest is drawn as a circle, and the systems with which it communicates are drawn as rectangles. These external objects are all devices (sensors or actuators). Note that the table control communicates with devices that are not part of the table, such as ARM1_MOTOR and ROBOT_MOTOR. This is to enforce synchronization between different parts of the production cell. Unlike DF diagram conventions, an arrow represents a synchronous communication rather than a flow of data, control, energy or material. No distinction is made between these different kinds of communications in the diagram. The direction of the arrow suggests initiative. A double-headed arrow represents a continuous communication. Continuous communications must be translated by an event recognizer into the relevant discrete events. The behavior of the event recognizer can easily be specified by a life cycle (e.g. as a polling algorithm).

A *transaction decomposition table* is a simple way to represent the decomposition of communications (called transactions) into their component events. Figure 4 shows a fragment of the transaction decomposition table of the table control.

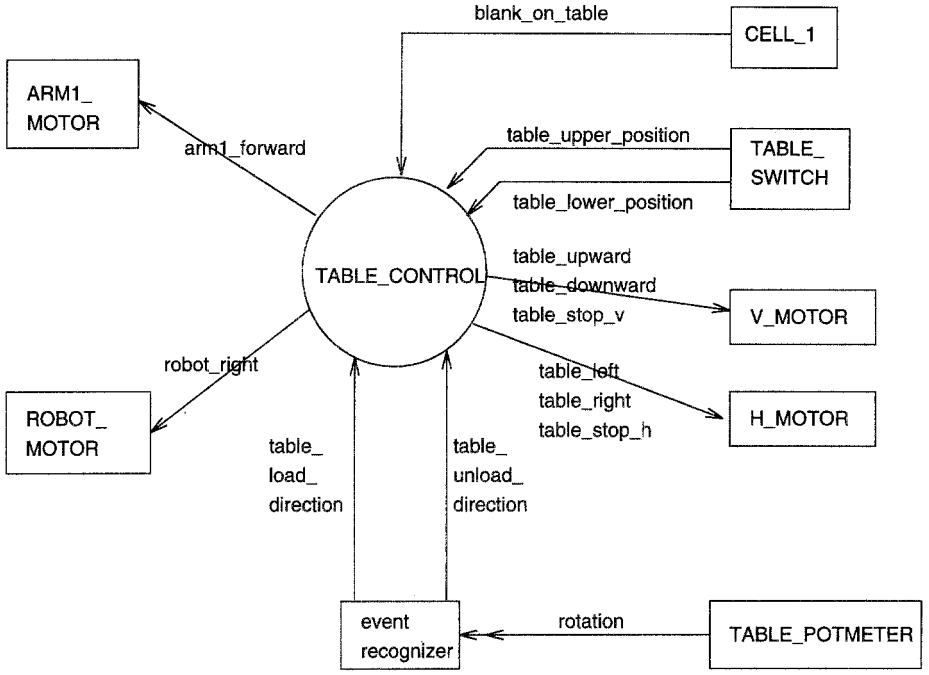

Figure 3 Context diagram of the table control

TABLE_CONTROL	start	blank_drops_ on_table	table_stops_ high	table_stops_ to_unload	move_arm1_ to_table	remove_blank_ from_table
CELL1		blank_on_table				
TABLE_SWITCH			table_upper_ position			
TABLE_V_MOTOR		table_upward	table_stop_v			table_downward
TABLE_H_MOTOR		table_right		table_stop_h		table_left
TABLE_POTMETER				table_unload_ direction		
ARM1_MOTOR					arm1_forward	
ARM1_MAG						arm1_mag_on
ROBOT_MOTOR					robot_right	

Figure 4 Part of the transaction decomposition table containing transactions of the table control object. The table control transaction omitted are table_stops_low and table_stops_to_load

The leftmost column contains object classes, and the top row shows all transactions of the control object. Each transaction is a communication event involving two or more events in the life of device objects. Each column thus shows in the context of which other events a local event is executing, and each row shows what the local events of the objects of a class are. One local event may be part of more than one communication event. The empty column below the *start* event is explained by the fact that the start event is a synchronization event between the different control objects and not between the devices of the system.

The control objects not only synchronize events in the life of sensors and actuators, they also synchronize events among themselves. in other words, global control is *distributed* over the control objects. For example, the move_arm1_to_table event in TABLE_CONTROL synchronizes the rotary table with ARM1 and ROBOT. Distributing these synchronization events over the different control objects makes the specification easier to understand but it makes it less reusable. As an alternative, one could define a PRODUCTION_CELL_CONTROL object, to which global control events are allocated. The lower-level control objects such as TABLE_CONTROL would thereby make less assumptions about the context in which they are employed, and would therefore become more reusable in other contexts.

19.3 Language for Conceptual Modeling (LCM 3.0)

19.3.1 Syntax and intuitive semantics

An LCM 3.0 specification consists of three components:

- A value type specification that defines all abstract data types needed for the model, such as natural numbers or rationals, in an order-sorted conditional equational specification. The intended semantics of the value type specifications is initial.
- A class specification that defines all object– and relationship classes in the model.
- A service specification that defines the system transactions. In a data-intensive system, these are registration events in which one or more system surrogates are updated because their corresponding UoD objects changed state.

In our model of the production cell control system, it turned out that the service specifications had to be enhanced to control object specifications, which have a life cycle that consists of system transactions. This does not involve a change in the underlying logic of the language, but it does require an extension of the syntactic sugar in which this logic is presented to the user.

For each object class to be defined in the class specification, the value type specifications must define an *identifier sort*, that provides all oids of the objects of that class. The identifier sort has the same name as the corresponding class. In the production cell example, the identifier sorts are the simplest sorts possible: they each contain only one constant as element. The identifiers of the table switch and table control objects are defined as shown in Figure 5. In data-intensive systems, there will in general by infinitely many identifiers of a class.

```
begin value type TABLE_SWITCH
    functions
        ts : TABLE_SWITCH;
end value type TABLE_SWITCH;

begin value type TABLE_CONTROL
    functions
        tc : TABLE_CONTROL;
end value type TABLE_CONTROL;
```

Figure 5 Specification of two identifier sorts

There are two kinds of class specifications in LCM 3.0, *object class* and *relationship class* specifications. For each object class, there is a corresponding identifier sort declaration of the same name. For each relationship class, the identifier sort is defined to be the cartesian product of the component identifier sorts. For example, if LOAN is a relationship class between DOCUMENT and MEMBER, then LOAN identifiers have the form $\langle d, m \rangle$, where d is a DOCUMENT identifier and m a MEMBER identifier. There are no relationship classes in the production cell example.

The *is_a* relationship between classes can be defined by defining a partial order on identifier sorts. For example, in a library database, we may want to declare the subclass relationship BOOK $\leq$ DOCUMENT. Semantically, this means that the class of book identifiers is a subset of the class of document identifiers. There is no significant taxonomic structure in the production cell (this is discussed in more detail in Section 19.5.)

The TABLE_SWITCH and TABLE_CONTROL classes are specified in Figures 6 and 7.

```
begin object class TABLE_SWITCH
    predicates
        Exists                          initially true;
        Table_upper_position;
        Table_lower_position            initially true;
    events
        table_lower_position;
        table_upper_position;
    life cycle
        TABLE_SWITCH = table_upper_position . table_lower_position . TABLE_SWITCH;
    axioms
        [table_upper_position(ts)] not Table_lower_position(ts);
        [table_lower_position(ts)] not Table_upper_position(ts);
end object class TABLE_SWITCH;
```

Figure 6 Specification of the TABLE_SWITCH object class

For each class, attributes, predicates and events are declared. The TABLE_SWITCH object contains two predicates, which are both initialized to true. In general, the Exists predicate will be set to true for an identifier by a creation event and set to false by a deletion event.

The production cell specification does not contain any attribute declarations. As an example of what an attribute declaration looks like, the following attribute of TABLE_CONTROL would count the number of blanks that has dropped on the table:

```
attributes
    nr_of_blanks : NATURAL              initially 0;
```

An attribute is a unary function on the identifier sort of the class. Only the codomain of the function is shown in the specification. Thus, nr_of_blanks is a function TABLE_CONTROL → NATURAL.

The events applicable to the instances of the class are declared in the events section. Each event is a function with codomain EVENT and may have several argument sorts. The first argument sort is not shown in the declaration, because it is always the identifier sort of the class. Thus, table_upper_position is a function TABLE_SWITCH → EVENT and blank_drops_on_table is a function TABLE_CONTROL x CELL1 x TABLE_V_MOTOR x TABLE_H_MOTOR → EVENT.

All communication events are *transactions* of the control system with its environment, and they are declared as such in TABLE_CONTROL. Their decomposition into local events is defined in the transaction decomposition section of the class specification where the transactions are specified.

```
begin object class TABLE_CONTROL
    predicates
        Exists                    initially true;
    transactions
        start;
        blank_drops_on_table(CELL1, TABLE_V_MOTOR, TABLE_H_MOTOR);
        table_stops_high(TABLE_SWITCH, TABLE_V_MOTOR);
        table_stops_low(TABLE_SWITCH, TABLE_V_MOTOR);
        table_stops_to_load(TABLE_POTMETER, TABLE_H_MOTOR);
        table_stops_to_unload(TABLE_POTMETER, TABLE_H_MOTOR);
        move_arm1_to_table(ARM1_MOTOR, ROBOT_MOTOR);
        remove_blank_from_table(TABLE_H_MOTOR, TABLE_V_MOTOR);
    transaction decompositions
        blank_drops_on_table(c1, tv, th) = CELL1.blank_on_table(c1) &
                    TABLE_V_MOTOR.table_upward(tv) &
                    TABLE_H_MOTOR.table_right(th);
        table_stops_high(ts, tv) = TABLE_SWITCH.table_upper_position(ts) &
                    TABLE_V_MOTOR.table_stop_v(tv);
        table_stops_low(ts, tv) = TABLE_SWITCH.table_lower_position(ts) &
                    TABLE_V_MOTOR.table_stop_v(tv);
        table_stops_to_load(tp, th) = TABLE_POTMETER.table_load_direction(tp) &
                    TABLE_H_MOTOR.table_stop_h(th);
        table_stops_to_unload(tp, th) = TABLE_POTMETER.table_unload_direction(tp) &
                      TABLE_H_MOTOR.table_stop_h(th);
        move_arm1_to_table(a1m, rm) = ARM1_MOTOR.arm1_forward(a1m) &
                    ROBOT_MOTOR.robot_right(rm);
        remove_blank_from_table(th, tv) = TABLE_H_MOTOR.table_left(th) &
                    TABLE_V_MOTOR.table_downward(tv);
    life cycle
        TABLE_CONTROL = start .
                    blank_drops_on_table .
                    (table_stops_high || table_stops_to_unload) .
                    move_arm1_to_table .
                    remove_blank_from_table .
                    (table_stops_low || table_stops_to_load) .
                    TABLE_CONTROL;
end object class TABLE_CONTROL;
```

Figure 7 Specification of the TABLE_CONTROL object class

The life cycle of the class instances is defined in a recursive process specification in the style of ACP [1]. The class name is used as main variable of this specification. The & operator in the transaction decomposition specification is the communication operator from ACP. It is commutative and associative. If $t = e_1$ & e_2, then t is considered to be different from e_1 and e_2, and the effect of t is the joint effect of that of e_1 and e_2. The process executed by the specified system is the parallel composition of all object life cycles, in which all local events that are not

transactions are encapsulated (renamed to deadlock). The only events that can occur, are transactions, whose effect is the same as the joint effect of the component events.

A class specification may contain axioms in order-sorted dynamic logic with equality, that constrain the values of attributes, define the effect of the events on the attributes, and define event preconditions. All axioms are universally quantified, but the quantifications are not shown in the example. Axioms must either be static integrity constraints, effect axioms, or precondition axioms.

A *static integrity constraint* is a formula without modal operators. It is an invariant of the state space of the system. An example of a static integrity constraint would be

nr_of_blanks(t) < 1000;

Safety constraints are all specified as static integrity constraints.

An *effect axiom* has the form $\phi \rightarrow [\alpha]\psi$, where ϕ and ψ are conjunctions of atoms of the form $a = c$ or literals, and α is an event. The meaning of $[\alpha]\psi$ is that after all possible executions of α, ψ is true.

A *precondition axiom* defines a necessary precondition of success for an event and has the form $\langle\alpha\rangle true \rightarrow \psi$. The meaning of $\langle\alpha\rangle\phi$ is that there is a possible execution of α that terminates and after which ϕ is true. The meaning of $\langle\alpha\rangle true$ is that there is a possible execution of α that terminates. The meaning of the precondition axiom $\langle\alpha\rangle true \rightarrow \psi$ is therefore that if we are in a state where there is a possible execution of α that terminates, then currently, ψ is true. For all non-creation events, there are implicit preconditions of the form

<table_lower_position(ts)>true → Exists(ts);

That is, only existing objects can execute (non-creation) events.

There are implicit *frame axioms* that define the non-effects of an event, i.e. they say what attributes and predicates do not change during the event.

19.3.2 Outline of declarative semantics

The declarative semantics of a class specification consists of three parts, an abstract data type, process algebra and a Kripke structure (Fig. 8).

Details on the declarative semantics of LCM specifications are given elsewhere [15][20]. Here, a brief outline is given of the basic ideas.

The value specification part of an LCM specification consists of a data type specification, which is interpreted in an *abstract data type A*, and a process type

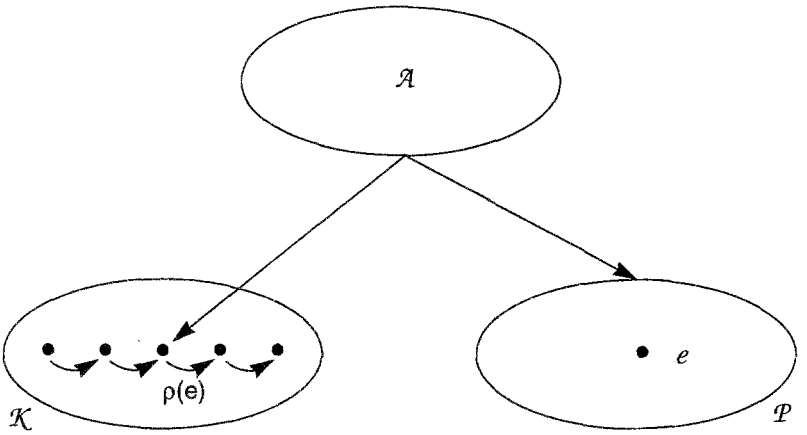

Figure 8 The structure of a model for LCM specifications

specification, explained below. As stated before, the intended semantics of the value type specification is the initial algebra semantics. This means that the abstract data type contains only data elements that can be named by closed terms in the specification, and that data elements are identified if and only if the closed terms denoting them can be proven equal in the value specification [7][10].

In addition to a value type specification, an LCM specification consists of class and transaction specifications. The class specification declares unary functions (called attributes) and unary predicates, both of which can be updated. The axioms of the specifications give static constraints on the attributes and predicates, and give effect and precondition axioms, that state how they are updated. These are interpreted in a *Kripke structure K* that consists of a set of possible worlds. All the possible worlds share the same domain, which is the abstract data type defined by the value specification. Different possible worlds may however assign a different interpretation to the attributes and predicates.

The class and transaction specifications in an LCM specification also declare atomic events, that are combined into object life cycles. Formally, we have sorts EVENT and PROCESS with EVENT ≤ PROCESS. The value specification defines process combinators as functions on PROCESS. For example, choice is a function declared with infix notation as

$$\_ + \_ \ : \ \mathsf{PROCESS} \times \mathsf{PROCESS} \rightarrow \mathsf{PROCESS}.$$

The axioms for the process combinators are taken from ACP [1]. These declarations and axioms jointly form a process theory, which can be viewed as a specification of a process type. The process type specification is interpreted in the

process algebra P of the model. The intended semantics is here the standard graph model of processes, slightly enhanced to include recursive process graphs [13]. The process algebra gives a meaning to processes independently of their effect on attributes and predicates. This means basically that *P* defines an equivalence relation on processes, that formalizes an observational equivalence notion. For example, the terms $e_1 + e_2$ and $e_2 + e_1$ are interpreted as the same process, because choice is commutative.

To define the effect of events and terminating processes on the attributes and predicates in the Kripke structure, a function ρ is defined that for each event *e* in *A*, defines an accessibility relation

$$\rho(e) \subseteq PW \times PW,$$

where *PW* is the set of possible worlds in *K*. ρ is extended by structural induction to terminating processes. There are many such functions ρ compatible with the axioms in the LCM specification. The intended semantics is that ρ assigns a minimal accessibility relation to *e*. In particular,

- ρ assumes that the conjunction of the completed preconditions of each event is necessary and sufficient for the event to lead to a next world. The completed precondition of an event *e* is the conjunction of all preconditions of *e* listed in the specification, all static integrity constraints (i.e. nonmodal axioms) and a nonmodal formula that guarantees the *e* leads to a next possible state.
- ρ(*e*) leads only to worlds that differ minimally from the current world.

There are still many different formalizations of this minimal change semantics, some of which are computationally tractable. This is subject of current study [18].

19.3.3 Axioms and inference rules

An axiom system that is complete for the loose semantics is given in [20]. A complete axiom system for the intended semantics given above is not yet known but in [19], we give a system that contains some of the needed frame axioms. To give an impression of the system, I list the modal logic axioms in Figure 9.

The R axiom says that if two actions are equal in the process algebra, then they have the same effect on the attributes and predicates of objects. It corresponds with the congruence condition on the function ρ in the semantics and it makes the axiomatization (without the frame axioms PosFr and NegFr) complete with respect to the loose semantics [20]. Axioms not listed above include first-order logic axioms,

(K) $[e]\,(\phi \rightarrow \psi) \rightarrow ([e]\,\phi \rightarrow [e]\psi)$

(R) $$\frac{\forall D :: e_1 = e_2}{\forall D :: [e_1]\phi \leftrightarrow [e_2]\,\phi}$$

(Barcan) $\forall x{:}\, s :: [e]\phi \rightarrow [e]\forall x{:}s :: \phi$ for $\notin Var(e)$

(PosFr) $\forall D :: P(t_1, ..., t_n) \rightarrow [e]P(t_1, ..., t_n)$
where P is nonupdatable and t_i contains only nonupdatable function symbols ($i = 1, ..., n$).

(NegFr) $\forall D :: \neg P(t_1, ..., t_n) \rightarrow [e] \neg P(t_1, ..., t_n)$
where P is nonupdatable and t_i contains only nonupdatable function symbols ($i = 1, ..., n$).

(N) $$\frac{\phi}{[e]\phi}$$

Figure 9 Modal logic axioms

axioms for substitution and congruence, equality axioms and the usual inference rules for first-order logic.

19.4 Verification of Safety and Liveness

Safety constraints are easily formalized as static integrity constraints in LCM. An example of a safety constraint is

```
   not Robot_zero_angle(rm)                    and
   not Arm1_zero_extension(a1m)                and
   not Table_lower_position(ts)
-> not (Blank_on_table(c1) and Arm1_holds_blank(a1))
```

Here, rm identifies the robot motor, a1m the motor of arm1, ts the table switch, c1 the photoelectric cell that signals whether a blank has arrived at the table, and a1 the magnet of arm1. The constraint says that if arm1 comes close to the table, then there must not be both a blank on the table and a blank in arm1. Specification of this constraint requires the addition of a number of predicates to the specification, and a number of effect axioms that update these predicates at the appropriate events. In order to prove that this constraint is respected during any possible behavior of the system, it was necessary to add some synchronization points. This is the reason why the events start and move_arm1_to_table are present in the table control life cycle. Since there is currently no automatic theorem-prover for LCM,

the constraints were proven manually. This could not be done as rigorous and detailed as would have been possible with the aid of a theorem-prover.

The idea of the proof of safety is very simple. The safety constraint is an axiom that should be true in all states of the model (i.e. in all possible worlds of the Kripke structure). The system is a parallel composition of a number of cyclic processes that in each of their transactions should respect the safety constraints. This is a classic integrity constraint verification problem known from databases: For each possible transaction, we should be able to prove that if the system currently satisfies the constraint, then it satisfies the constraint after executing the transaction. For any given constraint, only a few transactions are able to violate a constraint, viz. those transactions that update a predicate or attribute that occurs in the constraint. For example, only a few transactions update the safety constraint listed above, such as move_arm1_to_table. The proof that these transactions do not cause violations of the safety constraints is an elementary (but tedious) application of the axioms outlined above.

The specification makes a number of assumptions about the environment, that cannot currently be expressed in LCM. In general, it is assumed that the control system is ready to receive an event from its environment when it occurs. For example, it is assumed that the elevating rotary table is ready to receive the next blank when it arrives. This assumption cannot be expressed without constructs to specify and reason about real-time properties.

The safety constraints are currently formulated in an overly restrictive way, and as a consequence, the synchronization points introduced in the current specification cause an unnecessary reduction in concurrency between the movement of the parts of the production cell. This makes the system less efficient than it could be.

One of the results of this experiment in formal specification is that it became clear that the designer needs a tool to explore the design space effectively and efficiently. The tool should allow the designer to search for a specification that is optimal according to a set of criteria. One such criterion is that the parts of the production cell should not collide. Another is that the speed with which a blank is put through the system is as high as is possible, given the constraints on the system. Putting this differently, what we need is a tool that allows the designer to translate assumptions about the speed with which blanks arrive at the elevating rotary table and the speed with which the parts of the production cell moves into an optimal design of the control system. Safety analysis is only one part of the capability of

such a tool. Real time analysis and, possibly, real space analysis (to reason about locations and speed) is another.

We are currently designing a tool for *reachability analysis*, that allows the designer to evaluate the reachability properties of the specified system [8]. In general, the system will be able to answer questions of the form "Starting from a state satisfying ϕ_1, is there a sequence of transactions to a state satisfying ϕ_2 such that the path satisfies constraints Φ?". If there are such paths, the system should exhibit one of these and be able to show why it is compatible with the constraints. If there is no such path, the system should be able to explain why. Answering these queries requires theorem-proving as well as planning capability. We believe that such a system would aid the designer in the exploration of the design space in search of a system design that is safe as well as efficient. If a path of transactions is found that leads from an initial state to some desired state, it would allow the designer to present a constructive proof that a liveness property is satisfied by the system.

In general, reachability queries are unsolvable, but since our specifications have a simple form, there is hope that we can solve some interesting subcases using theorem-proving techniques. The techniques for finding a reachability (dis)proof borrows ideas from plan generation and theorem-proving in AI. To make the specification more amenable to these techniques, it is first translated into situation calculus, where for each updatable predicate, the events that can lead to its update, and the necessary and sufficient preconditions for each event, are listed. This is used to reason from the desired final state ϕ_2 to conditions ψ on an initial state that is compatible with the given conditions ϕ_1 on the initial state. A tableaux technique is used to find a model of the formula $\psi \wedge \phi_1$. A path from this model to the final state is then generated by applying the events in forward direction. There is a prototype implementation of this procedure in Prolog, using techniques from the Satchmo theorem prover [4][8].

19.5 Discussion of the Specification

19.5.1 Extending MCM to control-intensive systems

The application of MCM to control-intensive systems required one change in the method. In data-intensive systems, all control is present in the UoD. Most data-intensive systems are *registration systems*: they merely register the events that occur in the UoD (and answer queries about the registered data). Consequently, the structure of a model of system behavior of data-intensive systems is very simple: We

define a number of classes, one for each UoD object. Each UoD object may perform events, and some of them perform communications. These events and communications are registered by the system. The system model is therefore a copy of the UoD model, with the difference that the events and communications of system objects are initiated by the events and communications of the corresponding UoD objects.

Just as for registration systems, the transactions of the production cell control system correspond with transactions in the UoD. There are however two differences between control systems and registration systems:

- The initiative of the transaction may be with the control system as well as with objects in the UoD. Usually, a transaction is initiated by a device in the UoD. Because the control system enforces a synchronization with another device, the second device is then also forced to perform an event.
- The transactions of the control system are encapsulated in a control object, which has its own life cycle, by which it enforces meaningful behavior on the objects in the UoD.

No assumption is made about how many sensors or actuators participate in one transaction. In addition, we may specify different transactions to occur synchronously, so that several stimulus/response pairs may occur at the same time.

These extensions to MCM do not require a change in the logic of LCM. They merely reflect a different use of this logic. Note that control objects are very similar to the interactive function processes of JSD [12].

19.5.2 Extending LCM to control-intensive systems

LCM was found to be suitable for the specification of control-intensive systems. Some parts of the language were not used at all in this example. Thus, the specification uses mainly the constructs from ACP. Attributes, relationships between objects and taxonomic structures were not defined. In addition, object classification is not an issue in the production cell control system. However, the facilities of the language to specify these structures did not decrease the ease of use of the language for the specification of control structures.

There are some obvious extensions to the language, which would make it more useful for the specification and validation of such systems:

- *Real time* constructs that allow one to specify timing properties of events and states of the system. For example, in the spirit of time Petri nets [3], we could label each transition e with an interval $[t_1, t_2]$ that expresses that

e cannot be performed before time t_1 has elapsed from the moment that e is enabled, and must be performed before time t_2 has elapsed. This allows one to specify the time that a system must wait in a state as well as a time-out before which a certain event must occur. It also allows us to derive, by means of a reachability analysis, the maximum time needed for the production cell before it is ready to receive the next blank from the feed belt.

- *Exceptions* such as device failures have not been expressed at all in the current model. This requires the specification of time-outs as well as, more generally, the occurrence of abnormal events and recovery from abnormal behavior.

As a first move towards the realization of these extensions to LCM, we have started a project that extends dynamic logic with real time and with deontic logic (the logic of actual and ideal behavior) [21].

19.5.3 Implementation

The system has not been implemented in executable code. In the past, students have manually translated LCM specifications of database system behavior into SQL database schemas embedded into C, and into a persistent version of C++. The goal of these projects was to find out whether these translations could be done at all, and what could be preserved of the structure of the LCM specification. Verification of the implementation has not been performed, but it is clear that the use of dynamic logic offers the possibility to verify whether transactions have been implemented correctly. Each transaction is a terminating program that should realize the effects as specified in the effect axioms of the LCM specification. This remains a topic for further research.

19.5.4 Specification and verification effort

The specification was found by first searching for an informal model, represented in a number of diagrams and accompanying (unstructured) notes and comments. The major tool to come to grips with the informal model turned out to be the event trace diagram (also called message sequence diagram in some methods), showing all local events in the objects of the system as well as the synchronizations between the processes. It took me three iterations to arrive at the current architecture of the control system (i.e. this is the fourth version of the system). Altogether, this took me about 12 hours, distributed over one week. By dry-running the system, I convinced myself that I had modeled a system that should be able to respect the safety

constraints. To increase my confidence, I specified a number of safety constraints formally and proved them manually.

Having satisfied myself that I had found a stable model of a safe system, I wrote the formal specification, including the predicates that I discovered would be necessary for the formal proof of system safety. Excluding the crane specification, the result is about 8 pages and took about a day to write. Detailed proof of the safety constraints, down to the most detailed propositional logic manipulations, was not performed because this requires a theorem-prover.

Writing down the formal specification did not involve much creative work (other than inventing informative names for parts of the specification). What is sorely needed in this kind of clerical work is a workbench with intelligent text editing facilities for the specification as well as graph editing facilities for the diagrams, and the ability to cross-check the different parts of the formal and informal specification. Current research includes the incremental specification of such a workbench in LCM, and an implementation in C++.

19.6 Conclusions

With the minor extension of a construct to define control objects, LCM 3.0 is suitable for the specification of control-intensive systems. However, it does not contain any facilities for the specification of real-time properties that are usually important for control-intensive systems. Future work will therefore include extension of the language and its logic with real-time constructs. Current work includes the design and implementation of workbench for building integrated formal and informal specifications and of a tool for reachability analysis (without real time) of systems specified in LCM 3.0.

Acknowledgements

This paper benefited from comments made by Remco Feenstra and Claus Lewerentz on an earlier version.

References

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

[2] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[3] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, SE–17:259–273, March 1991.

[4] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 488–505, Venice, 1988. Springer-Verlag.

[5] R. Budde. ESTEREL Applied to the case study production cell. In *Case Study "Production Cell"*, FZI-Publication 1/94, pages 51–75. Forschungszentrum Informatik an der Universität Karlsruhe, 1994.

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

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

[8] R.B. Feenstra and R.J. Wieringa. Validating database constraints and updates using automated reasoning techniques. Submitted for publication.

[9] 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.

[10] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R.T. Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall, 1978. Volume IV: Data Structuring.

[11] H. Gomaa. *Software Design Methods for Concurrent and Real-Time Systems.* Addison-Wesley, 1993.

[12] M. Jackson. *System Development.* Prentice-Hall, 1983.

[13] P.A. Spruit and R.J. Wieringa. Some finite-graph models for process algebra. In J.C.M. Baeten and J.F. Groote, editors, *2nd International Conference on Concurrency Theory (CONCUR'91)*, pages 495–509, 1991.

[14] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems.* Prentice-Hall/Yourdon Press, 1985. Three volumes.

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

[16] 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.

[17] R.J. Wieringa and R.B. Feenstra. The university library document circulation system specified in LCM 3.0. Technical Report IR-343, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1993.

[18] R.J. Wieringa and W. de Jonge. Object identifiers, keys, and surrogates. *Theoretical and Practical Aspects of Object Systems*, To be published.

[19] R.J. Wieringa, W. de Jonge, and P.A. Spruit. Roles and dynamic subclasses: a modal logic approach. In M. Tokoro and R. Pareschi, editors, *Object-Oriented Programming, 8th European Conference (ECOOP'94)*, pages 32–59. Springer, 1994. Lecture Notes in Computer Science 821. Extended version to be published in *Theory and Practice of Object Systems (TAPOS)*.

[20] 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.

[21] R.J. Wieringa and J.-J.Ch. Meyer. DEIRDRE (deontic integrity rules, deadlines and real time in databases). Faculty of Mathematics and Computer Science, Vrije Universiteit., 1993.

[22] E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1989.