Algebraic View Specification*

Barbara Paech

Institut für Informatik, Technische Universität München
Arcisstr.21, D-80290 München
paech@informatik.tu-muenchen.de

Abstract. The application of algebraic specification techniques in the early phases of software development requires a means for specifying *views*. In this paper we argue for algebraic view specification based on an *algebraic concept model*. The concept model consists of two parts: a meta model defining the concepts of different views and the relationships between them, and a system model defining the system behaviour. We show how to derive an algebraic concept model from a semi-formal one given usually as an entity relationship diagram. This gives the rigour of formality to pragmatic view specifications and allows for an easy translation between formal and pragmatic specifications.

1 Introduction

Using algebraic methods in industrial software development means introducing an algebraic specification at some point of the development process and exploiting the mathematical semantics for code generation, refinement, verification and the like. There is some evidence that introducing formal specifications very early in the development process is most profitable [SH95]. This is due to the fact that errors in requirement analysis and design are the most costly, making the expenses for precise specification and thorough validation worthwile. In the following, *algebraic view specifications* based on an *algebraic concept model* are introduced as a means of making algebraic specifications better applicable to requirements analysis and definition.

Views

In the process of requirements analysis *several views* of the required software system and its environment are specified. The reason is that in the early stages there is not enough information to describe the system as a whole. Instead, seperated in several views, information is gathered which later on must be integrated in a design satisfying all the views. Pragmatic development methods like FUSION [CAB+94], OMT [RBP+91], OOSE [Jac92], SSADM [DCC92] offer description techniques for theses views. Mostly, only the notation is defined together with an informal semantics. Consequently, CASE tools often only support editing and syntactic checks of these description.

* This work was carried out within the project SysLab, supported by Siemens Nixdorf and by the Deutsche Forschungsgemeinschaft under the Leibniz program

Meta Model

More powerful CASE tools are based on a *repository* in which information about all the objects relevant to the software development is stored [HL93]. Following the ANSI standard for *information resource dictionary systems* [ANS89] there are four levels of such objects:

1. the real world objects relevant to the software system (e.g. `Miss Marple`),
2. the types and relationships of the real world objects (e.g. `detective, crime, worksOn`), often called *model level*,
3. the concepts used to describe these types and relationships (e.g. `entity, process`), often called *meta model level*, and
4. the concepts used to describe the meta model.

From the point of view of a method designer the meta model level is particularly interesting. On this level the general concepts used to model the application and the software system are fixed. Interestingly, most books on software engineering methods do not make this meta model explicit. It is, however, becoming increasingly popular for method comparison (e.g. [Gil93]). In the tradition of semantic data modelling usually entity relationship or object diagrams are used to define the meta model. Thus the modelling concepts are characterized through their attributes and relationships (and operations - in case of an object diagram). Each view corresponds to a certain part of the meta model. Relationships between concepts of different views determine *consistency conditions* between different views. These consistency condition can be enforced by a CASE tool to give support to the *integration* of the views.

Concept Model

To allow for *algebraic* view specification the meta model can be formalized in an algebraic specification language. This idea is worked out in the first part of the paper. However, from the point of view of algebraic specification, traditional meta models are not sufficient. They do not provide a semantics of the modelling concepts in terms of *system behaviour*. Therefore, in the second part of the paper the *system model* is introduced and combined with the meta model. The combined model is called *concept model*. Different parts of this model determine different views together with their mathematical semantics. The relationships between the concepts give rise to very powerful consistency conditions.

This paper is structured as follows. As an example we introduce part of the meta model of the method FUSION and the corresponding views. Then we discuss their formalization. In section 4 we define a system model for FUSION and show how to combine the two models to give a precise semantics to the views. Related work is discussed in the conclusions.

2 Views and the Meta Model - An Example

As an example we discuss the simplified meta model of the analysis phase of FUSION (see figure 1). The notation used for this and the following models is explained in the legend. We only show the most important attributes.

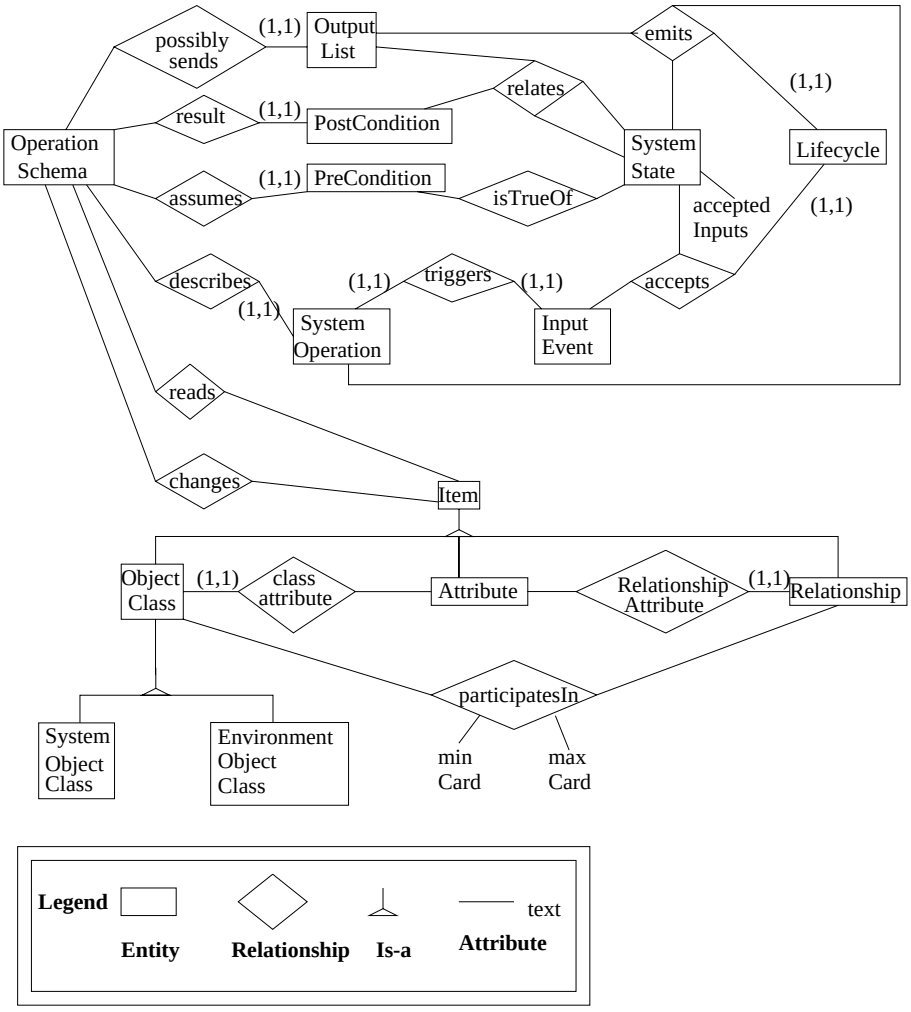

Fig. 1. Meta Model

This meta model is not given explicitly in [CAB+94]. The model given here covers all concepts necessary to explain the description techniques for the views

developed in the analysis phase. After explanation of the model along with the description techniques for the different views some modelling alternatives are discussed.

FUSION offers three views: the object model, the operation model and the lifecycle model.

Object Model

The *objects* of the application, together with their *attributes* and *relationships*, are described in the object model[2]. In a second step the objects included in the software system are distinguished from the environment objects. The former consitute the *system object model*. In figure 2 part of the object model for the well-known automatic teller machine example is shown. Figure 3 gives the relevant part of the meta model.

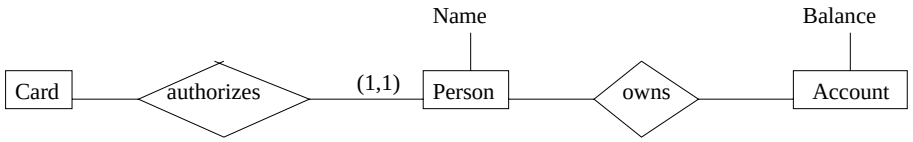

Fig. 2. Object Model Example

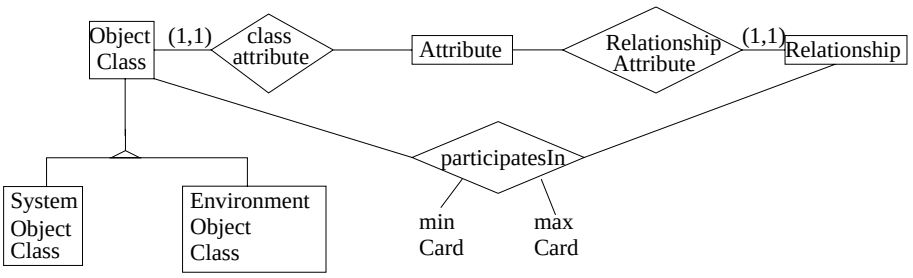

Fig. 3. Object Model

[2] For the sake of simplicity we omit further structuring facilities like generalization and aggregation here.

Operation Model

System operations are *described* in one or more *operation schemata*. Each schema covers the items *read* or *changed*, the output events *possibly sent*, the *precondition* and *the postcondition*. The precondition characterizes the set of *system states* enabling the operation. The postcondition characterizes a set of pairs of system states and a *list of output events* such that the application of the operation in the first state yields the second state together with the list of output events. As example the system operation `dispenseCash` of the automatic teller machine is given in figure 4[3]. The relevant part of the meta model is shown in figure 5.

Operation	`dispenseCash(amount)`
reads	`Account`
changes	`Account`
sends	`Person`: {`overdraw, newAmount?, cash, ejectCard`}
assumes	`Card` inserted and `Account owned by Person` who is `authorized` through `Card`
result	If `amount` available on the `Account`, subtract `amount` from `Account` balance, send `cash` and `ejectCard`. Otherwise send `overdraw` and `newAmount?`.

Fig. 4. Operation Model Example

Lifecycle Model

The *lifecycle* model determines when input events are *accepted* (depending on the history of accepted events) and what output events may be *emitted*. In FUSION lifecycles are denoted as regular expressions, where output events are prefixed by #. An example describing the lifeycycle of the automatic teller machine is given in figure 6. Figure 7 shows the corresponding part of the meta model.

As is most evident in the lifecycle meta model, the concrete syntax of the description technique and the concepts defined in the meta model may be quite different. This only mirrors the fact that different description techniques can be used for the same view. Therefore the meta model is much more adequate for method comparison than the description techniques themselves. Also, the meta model makes it easier to understand the description techniques.

[3] We have simplified the parameterization mechanism of FUSION here.

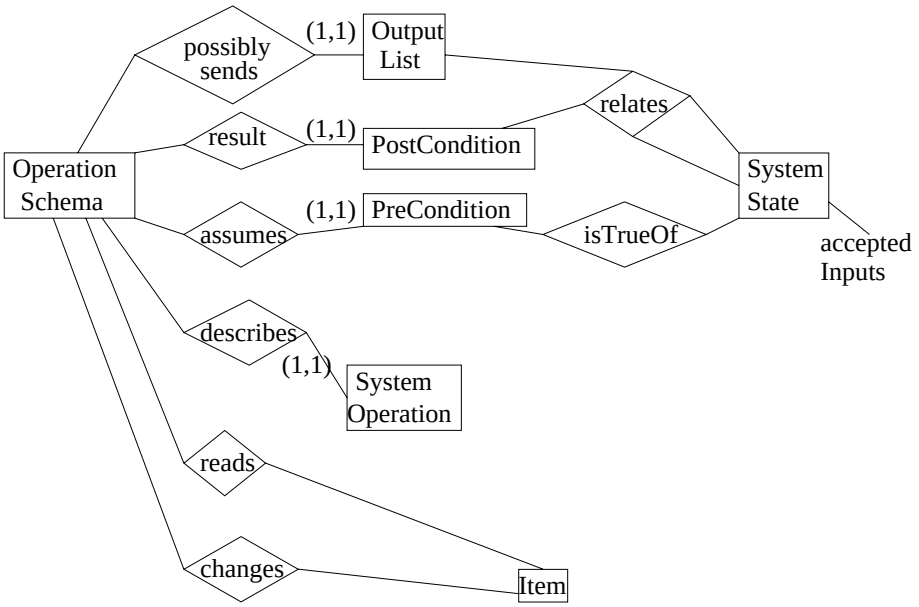

Fig. 5. Operation Model

```
(insertCard.(#rejectCard |
            #amount?. (dispenseCash.#overdraw.#newAmount? |
            dispenseCash.#cash.#newAmount?)*. eject.#ejectCard))*
```

Fig. 6. Lifecycle Example

This finishes the explanation of the meta model. As mentioned before, this model is not unique. It is adequate for method description. For implementation in a CASE repository it is too general. For example, it is not possible to keep a list of all system states. Therefore only a simplified version of the relationships involving `SystemState` will be supported. E.g. instead of the `accepts` relationship all input events referred to in the lifecycle could be recorded.

3 Algebraic Meta Model and Algebraic Views

As shown in [Het95] entity relationship diagrams can be formalized in an algebraic specification language. For each entity a sort is introduced, for each

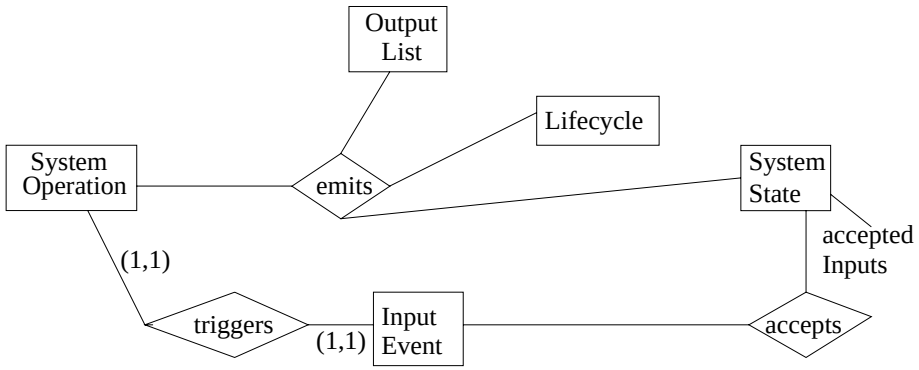

Fig. 7. Lifecycle Model

attribute an operation and a predicate for each relationship. Cardinalities are expressed as constraints. In the following the meta model of figure 1 is translated into the algebraic specification language SPECTRUM [BFG+93]. The translation is shown in figure 8. The numbers (i) are reference points for the following explanation.

(1) This specification is based on the specification of natural numbers, of sets, of the attribute sorts and polymorphic lists. All functions are strict and total.
(2) For each entity of the meta model a sort is introduced. `EQ` means that for all the introduced sorts equality is decidable.
(3) The attributes are mapped into strict and total functions from the entity sorts to the attribute sorts:
(4) The relationships are mirrored in general by predicates. One-one relationships are translated into functions (see for example the `triggers` relationship).
(5) The cardinality constraints are captured by axioms. These axioms use the function `card` defined in the specification of sets. For a relationship `rel: Sort1` $\times$ `...` $\times$ `Sortn` a cardinality constraint for `Sorti` determines the number of instances of the relationship where the entities of the other sorts are fixed. As an example consider the relationship `accepts` between `Lifecycle`, `SystemState` and `InputEvent`.
(6) As discussed in [Het95] there are many constraints not expressible with entity relationship diagrams. They can be expressed in SPECTRUM and added as further axioms to the algebraic meta model. One example is a stronger cardinality constraint for `Lifecycle` in `accepts`. There is just one lifecycle for the whole system. Another example is the consistency between the output list characterized by `relates` and the one characterized by `emits`.

Figure 10 gives an example of an algebraic view specification. It specifies the equivalent of the lifecycle of figure 6 based on the concepts defined in

```
META MODEL = {

(1) enriches Nat + Set + AttrSort + List;
    strict total;
(2) sort InputEvent, SystemOp, OpSchema, OutputEvent, SystemState,
    Precondition, Postcondition, Lifecycle, Attribut, ObjectClass,
    SystemObjectClass, EnvironmentObjectClass, Relationship;
    sort Item = Attribute ⊎ Relationship ⊎ Object;
    EQ;
(3) acceptedInputs        : SystemState → List InputEvent;
    minCard               : Relationship × ObjectClass → Nat;
    maxCard               : Relationship × ObjectClass → Nat;

(4) classAttribute        : Attribute× ObjectClass → Bool;
    relationshipAttribute : Attribute × Relationship → Bool;
    participatesIn        : ObjectClass × Relationship → Bool;
    describes             : OpSchema → SystemOp;
    possiblySends         : OpSchema → List OutputEvent;
    triggers              : InputEvent → SystemOp;
    reads                 : OpSchema × Item → Bool;
    changes               : OpSchema × Item → Bool;
    assumes               : OpSchema → Precondition;
    result                : OpSchema → PostCondition;
    isTrueOf              : Precondition × SystemState → Bool;
    relates               : Postcondition × SystemState × SystemState
                          × List OutputEvent → Bool;
    accepts               : Lifecycle × SystemState × InputEvent → Bool;
    emits                 : Lifecycle × SystemState × SystemOp
                          × List OutputEvent → Bool;

(5) axioms
    ...
    ∀ l: LifeCycle. ∀ st: SystemState. ∃ acceptsSet: Set (LifeCycle ×
    SystemState × InputEvent). ∀ in: InputEvent.
    ((l,st,in) ∈ acceptsSet ⇔ accepts(l,st,in)) ∧
    1 ≤ card(acceptsSet);
    ∀ l : LifeCycle. ∀ in:InputEvent. ∃ acceptsSet: Set (LifeCycle ×
    SystemState × InputEvent). ∀ st: SystemState.
    ((l,st,in) ∈ acceptsSet ⇔ accepts(l,st,in)) ∧
    1 ≤ card(acceptsSet);
    ∀ in: InputEvent. ∀ st: SystemState. ∃ acceptsSet: Set (LifeCycle ×
    SystemState × InputEvent). ∀ l: Lifecycle.
    ((l,st,in) ∈ acceptsSet ⇔ accepts(l,st,in)) ∧
    1 ≤ card(acceptsSet) ∧ card(acceptsSet) ≤ 1);
(6) ........
    ∀ l,l': lifecycle. ∀ st,st': SystemState. ∀ in,in': InputEvent.
    (accepts(l,st,in) ∧ accepts(l',st',in')) ⇒ l = l';
    ∀ st: SystemState. ∀ op: SystemOp. ∀ out: List OutputEvent.
    (∃ l: Lifecycle. emits(l,st,op,out)) ⇔
    (∃ st': SystemState. ∀ sch: OpSchema . ∀ cond1: Precondition. ∀
    cond2: Postcondition. (describes (sch) = op ∧ assumes(sch)= cond1 ∧
    isTrueOf(cond1,st) ∧ result(sch) = cond2) →
    relates(cond2,st,st',out));
    endaxioms;

}
```

Fig. 8. Formal Meta Model

the algebraic meta model. First, for the sorts `InputEvent, OutputEvent` and `Lifecycle` the relevant elements are introduced. Then the relationships `accepts` and `emits` are specified by axioms. Essentially, these axioms characterize an Mealy-automaton corresponding to the lifecycle expression. This automaton is shown in figure 9.

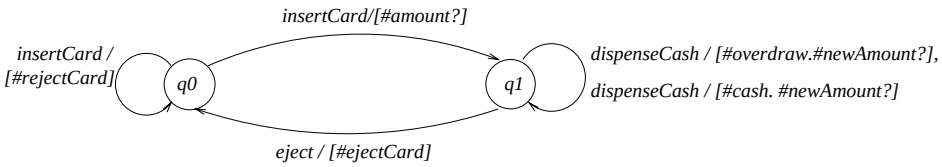

Fig. 9. Lifecycle Automaton

Automaton state q_0 characterizes all system states where the sequence of `acceptedInputs` is empty or ends with *eject*. Automaton state q_1 characterizes system states reached by accepting *insertCard* or *dispenseCash*. So the axioms for `accepts` first describe the set of accepted `InputEvents`. *insertCard* is accepted in system states characterized by q_0 and *dispenseCash* and *eject* are accepted in system states characterized by q_1. The axioms for `emits` first describe the set of `OutputEvents`. Then the `OutputEvent` is specified in dependency of the `InputEvent`. This corresponds to the transition labels of the automaton of figure 9.

Discussion

The algebraic meta model and view specification combines elements of pragmatic and formal software engineering methods.

From the point of view of pragmatic software engineering methods, formally specified views are not of *direct* use because of the skills required to apply mathematical specifications in general. However, mirroring the meta model in the algebraic meta model allows for an easy translation between the pragmatic view description and the algebraic view description. This translation supports the *indirect use* of formal methods as characterized in [Hußß95]. The algebraic specification can be used for an analysis of the corresponding views yielding precise rules for transformation and consistency checks to be used by the software developer. If required, e.g. in the case of safety-critical system properties, a large part of the algebraic specification can be generated from the informal one.

From the point of view of algebraic specifications, views introduce a new structuring principle. The meta model introduces a new level of indirection into the specifications making the modelling concepts explicit. While this lengthens the specifications, it makes the process of specifying more flexible. Because of the

```
ATM LIFE = {
enriches META MODEL;
insertCard, dispenseCash, eject : InputEvent;
#rejectCard, #amount?, #overdraw, #newAmount?, #cash,
#ejectCard : OutputEvent;
atmLife : Lifecycle;

axioms
∀ st: SystemState. ∀ in: InputEvent.
accepts(atmLife,st,in) ⇒ (in = insertCard ∨ in = dispenseCash ∨ in = eject) ∧
accepts(atmLife,st,insertCard) ⇔ (acceptedInputs(st) = [] ∨
(δlast(acceptedInputs(st)) ∧ last(acceptedInputs(st)) = eject)) ∧
(accepts(atmLife,st,dispenseCash) ∨ accepts(atmLife, st, eject)) ⇔
(δlast(acceptedInputs(st)) ∧
( last(acceptedInputs(st)) = insertCard ∨ last(acceptedInputs (st)) =
dispenseCash));

∀ st:SystemState. ∀ out: List OutputEvent. ∀ x: OutputEvent.
∀ op: SystemOp.
(emits(atmLife,st,op,out) ∧ isEl(x,out)) ⇒ (x = #rejectCard
∨ x = #amount? ∨ x = #overdraw ∨ x = newAmount? ∨ x = #cash
∨ x = #ejectCard) ∧
emits(atmLife,st,op,out) ⇒ (op = triggers(insertCard)
∨ op = triggers(dispenseCash) ∨ op = triggers(eject)) ∧
emits(atmLife,st,triggers(insertCard),out) ⇔
(out = [#rejectCard] ∨ out = [#amount?]) ∧
emits(atmLife,st,triggers(dispenseCash),out) ⇔
(out = [#overdraw. #newAmount?] ∨ out = [#cash.#newAmount?]) ∧
emits(atmLife,st,triggers(eject),out) ⇔ (out = [#ejectCard]);
endaxioms
```

Fig. 10. Algebraic Lifecycle Specification

common vocabulary different views can be specified and refined by different people at different times. The consistency constraints between the views are made explicit through the axioms relating concepts of different views. One example of such a consistency condition is the second formular in (6) of figure 8. It relates the lifecycle and the operation view.

However, the semantics of the concepts given in the algebraic meta model is not complete. Nothing is said about the relationship between system states and objects. Nothing is said about the transition between two system states in general. The meta model is lacking the concepts necessary to define *system behaviour*. Of course, we could have included them in the meta model from the beginning. However, this is not necessary, if the meta model is used to charcterize

the views. Also, meta models of pragmatic methods do not cover these concepts. Thus we prefer to introduce another model collecting all the concepts relating to system behaviour in one place: the *system model*. The system model of FUSION, its integration with the meta model, and its formalization is discussed in the following sections.

4 System Model and Concept Model

In this section the system model of FUSION is given. For reasons of space, we only give the semi-formal version denoted as an entity relationship-diagram. The derivation of the algebraic version follows the approach discussed above.

FUSION only considers *sequential systems*. These can be modelled as state transition systems, where the state determines the set of existing objects, the values of attributes and of relationships. Since FUSION distinguishes between input events received and the ones accepted, in each state the list of accepted input events leading to this state is included. Also, for each transition the list of emitted output events is recorded. Altogether, figure 11 shows the system model. Entities depicted as dashed boxes (`SystemObjectClass` in the figure) are just introduced for layout reasons to avoid crossing lines.

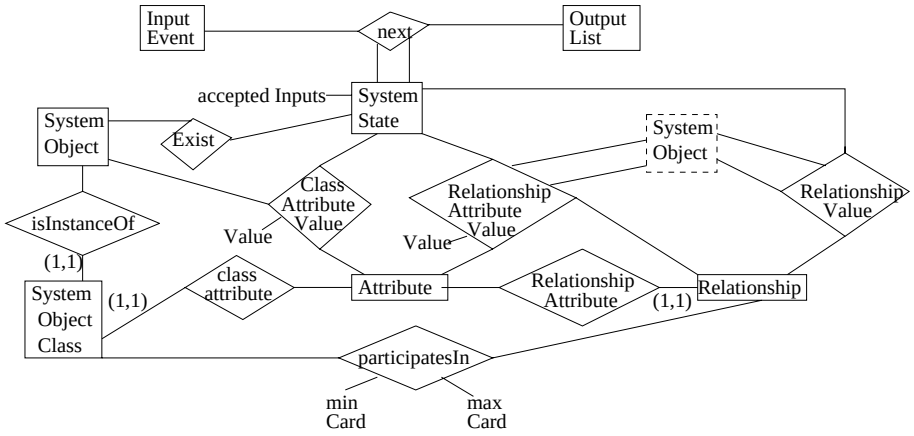

Fig. 11. System Model

The meta model and the system model can be joined by identifying common concepts. We call the combined model the *concept model*. On the algebraic side this corresponds to a specification combination using +. Again, there are a lot of consistency constraints between the concepts of the joined models which are not expressed in the entity relationship diagram. In the algebraic concept model these can (and have to) be expressed with further axioms. The most important

one is the connection between **next** and the concepts of the operation schema (see figure 12).

```
∀ st1,st2: SystemState. ∀ in : InputEvent. ∀ out: List OutputEvent.
next( st1,in,st2,out) ⇔
∃ life: Lifecycle. ∃ op: SystemOp. ∀ sch: OpSchema.
∀ cond1 : PreCondition. ∀ cond2: PostCondition.
(¬ accepts( st1,in,life) ∧ st1 = st2 ∧ out = []) ∨
(accepts(st1,in,life) ∧ triggers( in) = op ∧
((assumes (sch) = cond1 ∧ isTrueOf( cond1,st1) ∧ result( sch) = cond2)
⇒ relates( cond2,st1,st2,out)) )
```

Fig. 12. Concept Model Axiom

The concept model gives a semantics to all modelling concepts in terms of the system behaviour. Some of the concepts are directly reflected in the syntax of the description techniques (e.g. the concepts of the object model), while others are only indirectly reflected (e.g. the concepts of the lifecycle model). Also, some concepts are exclusively related to views or the system model, while others are common to some view and the system model. Thus the purpose of the concepts within the method is made explicit in the concept model. Structuring formal specifications along these lines makes them easier to comprehend.

5 Conclusions

Altogether we advocate the following combination of algebraic specification and pragmatic software development methods. The views supported by the pragmatic method are captured in the meta model. Together with the system model underlying the pragmatic method it constitutes the pragmatic concept model. The concept model is directly translated to an algebraic concept model using the ideas of [Het95]. This specification has to be completed with axioms for the constraints not expressible in the entity-relationship-diagram. The resulting specification can be structured as follows:

Basic View Specification All sorts, functions and axioms which refer to concepts of one view or the system model specify the semantics of that view.

Consistency All axioms which refer to concepts of different views specify the consistency constraints between the views..

Based on this specification the concrete syntax of the description techniques supported in the pragmatic method is given a semantics in terms of the algebraic concept model. Thus every pragmatic view specification can directly be translated to an algebraic one. Following the paradigma of indirect use, the algebraic

view specification is hidden to the everyday system developer. It is exploited as much as possible within CASE-tools for the pragmatic method. One example is the automatic check of consistency constraints between views. Another example is support of a refinement notion for views based on the algebraic semantics. An expert system developer might use the algebraic semantics for semi-automatic proofs of complex and critical properties.

Related Work

There are two sides two our approach. On one hand it is related to the efforts to give a formal foundation to pragmatic software development methods. In this respect our work is most influenced by [Huß94] where an algebraic semantics to SSADM is given. There also an algebraic system model is used to give semantics to the description techniques of SSADM. However, this is not related to an explicit meta model.

On the other hand our approach extends the work on algebraic system specification. Approaches for the algebraic specification of concurrent systems, for example SMoLCS [AMRW85], also include a system model. However, this work is not extended to views. In [GH95] the language TROLL *light* is given an algebraic semantics. TROLL *light* offers a template to specify different aspects of the system. However, this template does not give the full flexibility of views, since all aspects have to be specified at the same time. So this language is more suited for the design than for requirements analysis.

Future Work

In this paper we have argued for an algebraic concept model allowing for the algebraic specification and integration of views. On one hand this introduces the flexibility of views into algebraic specifications making them better applicable to the early phases of software development. On the other hand this gives the rigour of formality to pragmatic view specification. A great challenge is to extend these ideas to the specification of distributed systems. Within the SYSLAB-project a powerful system model for distributed systems has been given [RKB95] . In a next step this will be related to a meta model supporting an adequate set of views. Because of the additional complexity of distributed systems it is even more important to integrate the views using an algebraic concept model.

Acknowledgement

Thanks are due to the SYSLAB-group for stimulating discussions and to the anonymous referees for many helpful comments.

References

[AMRW85] E. Astesiano, G.F. Mascari, R. Reggio, and M. Wirsing. On the parameterized algebraic specification of concurrent systems. In *TAPSOFT,LNCS 185*, pages 342–358. Springer Verlag, 1985.

[ANS89] ANSI. *American National Standard X3.138-1988: Information Resource Dictionary System (IRDS).* American National Standard Institute, 1989.

[BFG+93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM, An Informal Introduction, Version 1.0. Technical Report TUM-I9312, Technische Universität München, 1993.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development - The FUSION method.* Prentice Hall, 1994.

[DCC92] E. Downs, P. Clare, and I. Coe. *Structured systems analysis and design method: application and context.* Prentice-Hall, 1992.

[GH95] M. Gogolla and R. Herzig. An algebtraic development technique for information systems. In *AMAST,LNCS 936*, pages 446–460. Springer Verlag, 1995.

[Gil93] M. Gilpin. A comparison of object-oriented analysis and design methods. Technical report, INTERSOLV at CASE World, 1993.

[Het95] R. Hettler. *Entity/Relationship-Datenmodellierung in axiomatischen Spezifikationssprachen, Ph.D. thesis.* Reihe Softwaretechnik, FAST. Tectum Verlag, 1995.

[HL93] H. Habermann and F. Leymann. *Repository.* R. Oldenbourg Verlag, 1993.

[Huß94] H. Hußmann. Formal foundation for pragmatic software engineering methods. In B. Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen*, pages 27–34, 1994.

[Huß95] H. Hußmann. Indirect use of formal methods in software engineering. In M.Wirsing, editor, *ICSE-17 Workshop on Formal Methods Applications in Software Engineering Practice*, pages 126–133, 1995.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design.* Prentice-Hall, 1991.

[RKB95] B. Rumpe, C. Klein, and M. Broy. Ein strombasiertes mathematisches modell verteilter informationsverarbeitender systeme - SYSLAB Systemmodell. Technical Report TUM-I9510, Technische Universität München, 1995.

[SH95] H. Saiedian and M. Hinchey. Issues surrounding the transfer of formal methods technology into the actual workplace. In M.Wirsing, editor, *ICSE-17 Workshop on Formal Methods Applications in Software Engineering Practice*, pages 69–76, 1995.

This article was processed using the LaTeX macro package with LLNCS style