Proc. 4th Int. Conf. on Algebraic Methodology and Software Technology (AMAST'95), V.S. Alagar (Ed.), Springer, Berlin, LNCS (1995)

An Algebraic Development Technique for Information Systems*

Martin Gogolla & Rudolf Herzig

Bremen University
Department for Mathematics and Informatics
Database Systems Group
Postfach 330440, D-28334 Bremen, GERMANY
e-mail: gogolla@informatik.uni-bremen.de

Abstract. This paper reports on successful application of algebraic ideas to the formal development of software systems, in particular information systems. It describes (1) a formalism, i.e., a language, for the specification of information systems, (2) a method for the construction of specifications in this language, and (3) implemented and planned parts of a specification environment covering important phases of the software development process.

Keywords: *Algebraic and Logical Foundations* – Algebraic Methodologies for Languages and Systems; *Concurrent and Reactive Systems* – Algebraic Approaches, Object Oriented Models; *Software Technology* – Specification Languages and Tools, Integration of Pragmatic and Formal Methods.

1 Introduction

Although theory and practice of algebraic specifications have been studied by many people for about twenty years, there are opinions doubting that this line of research is well-suited for practical software development:

> *Algebraic specification techniques have been widely applied to small examples but there is little evidence yet that they are suitable for specifying large systems. It is worth trying to explain the problems of algebraic specifications with a small example. Imagine a system containing a database which we wish to update. If we model the database directly we can simply specify validity (the object to be updated exists, the type is correct, etc.). In an algebraic approach we would have to establish existence of an object by reasoning about the sequence of inputs to the system and determining whether the object had been created (successfully) since it was last deleted. This would make the specification obscure and cumbersome.*
>
> *John McDermid and Paul Rook [MR91]*

* Work reported here has been partially supported by the CEC under Grant No. 6112 (COMPASS).

We do not follow the above arguments, but propose an algebraic approach to the specification of information systems dealing with the notion of state in an implicit manner. Thus the above criticism does not hold for our approach (and other recent algebraic proposals like [GM87, DG94, EO94, GD94, MQ93]), because the notion of state for a system is (at least for our application area, namely databases and information systems) so important that we decided to incorporate it directly into our framework. The basic idea is to represent system states as well as system transitions as algebras. Therefore, although we stay on algebraic grounds, we give up the idea (present in classical algebraic specifications) that the complete system is specified as one algebra.

Thus, the main contribution of this paper is *one* attempt to explain why and how algebraic techniques can be applied successfully for practical software development. Following [ESD93], our approach to software engineering [GJM91], in particular to software development, can be visualized in Fig. 1 by a triangle picturing the tension between a formalism, a method, and an environment.

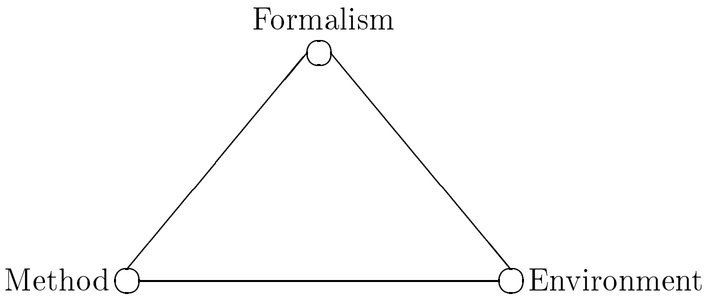

Fig. 1. Ingredients of a Software Development Approach

By *formalism* we mean a specification or programming language with well-defined syntax and semantics. However having an expressive language is not enough. Developers must have some guidelines how to proceed when they write specifications or programs. Thus, a *method* supporting the construction of software should be at hand. In order to round off the scenario, we do not expect that software development is carried out in a paper-and-pencil fashion but is assisted by a computerized development *environment* tailored to the respective formalism and method.

Let us now come back to our application area, namely information systems. Here, the aim of the conceptual modeling process is to provide a first precise description of the part of the world which is to be modeled, the so-called Universe of Discourse (UoD). By observing a certain UoD one realizes that it consists of complex structured entities with time-varying behavior. These entities exist concurrently and interact with each other. Therefore the idea of object-oriented specification is to represent the real-world entities as formal objects in a specification language. The main advantage of following the object paradigm is the fact that all relevant information concerning one object can be found within one single unit and is not distributed over a variety of locations.

Recently, a number of object specification languages like ABEL [DO91], CMSL [Wie91], GLIDER [CJO94], MONDEL [BBE+90], OCS [AAR94], OS [Bre91], and Π [Gab93] have been proposed. However, we work with the language TROLL *light*, a dialect of OBLOG [SSE87] and TROLL [JSHS91]. In TROLL *light* structure as well as behavior of real-world entities can be characterized in a direct way.
The structure of the rest of this paper mainly follows the triangle depicted in Fig. 1. In Sect. 2 we sketch the formalism we use, namely the language TROLL *light*. Section 3 explains our method for the development of information systems. In Sect. 3 we describe implemented and planed tools for supporting the method mentioned before. The last section gives some concluding remarks.

2 Formalism

2.1 Concepts and Syntax of TROLL *light*

We explain the essentials of TROLL *light* by means of an example. In Fig. 2 binary trees, where the nodes carry boolean information, are specified as object types or templates as object types are called in TROLL *light*.

```
TEMPLATE Node
  DATA TYPES  Bool;
  ATTRIBUTES  Content:bool;
  SUBOBJECTS  Left, Right:node;
  EVENTS      BIRTH create;
                    createLeft;
                    createRight;
                    update(Content:bool);
              DEATH destroy;
  VALUATION   [ create ] Content=true;
              [ update(B) ] Content=B;
  INTERACTION createLeft >> Left.create;
              createRight >> Right.create;
  BEHAVIOR    PROCESS Node =
                ( create -> NodeLife );
              PROCESS NodeLife =
                ( createLeft, createRight, update -> NodeLife |
                  { UNDEF(Left) AND UNDEF(Right) }
                    destroy -> Node );
END TEMPLATE
```

Fig. 2. TROLL *light* example template

`DATA TYPES`**:** In TROLL *light*, data types are employed for various purposes. Data types are assumed to be specified with a data type specification language [Wir90]. Their signature is made known in a template with the `DATA`

`TYPES` section. The basic difference between data values as instances of data types and objects as instances of object types is that values are regarded as stateless items while objects are associated with an inherent notion of state.

`ATTRIBUTES:` The `Node` template uses the data type `Bool` in order to define one attribute `Content` of data sort `bool`.

`SUBOBJECTS:` For templates (or object types) we employ the same naming convention as for data types. Template names are written capitalized, and each template (`Node`) induces a corresponding object sort written exactly as the template but with a starting lower case letter (`node`). A `Node` object is allowed to have two local sub-objects, namely `Left` and `Right`, of sort `node`.

`EVENTS:` The things which can happen to `Node` objects are called events. `Node` objects can be created, their left and right sub-nodes can be created, they can be updated, and they can be destroyed.

`VALUATION:` Valuation rules serve to define the effect of events on attributes. They specify the value an attribute has after the occurrence of an event.

`INTERACTION:` Interaction rules are used to describe the synchronization of events. For instance, whenever the `createLeft` event occurs in the parent object, the `create` event must occur for the left sub-object. New objects can only be created as sub-objects of existing objects by this mechanism which is called birth event calling.

`BEHAVIOR:` Possible life cycles, i.e., sequences of allowed events, are specified in the behavior part. For a given template there is always a process with the same name which is the initial process for objects belonging to the template. An objects' life starts with a `BIRTH` event and possibly ends with a `DEATH` event, which in this case is only allowed to occur when there are no sub-objects. In between, the events `createLeft`, `createRight`, and `update` can occur, but due to the interaction rule, for instance, `createLeft` can only occur if the called event `create` fits into the sub-objects' life cycle.

Above, we have shown a very simple example of a TROLL *light* specification for introductory purposes. For more sophisticated applications, we have to refer to [CGH92, GCH93, GHC$^+$94]. There, you find for instance parameterized attributes and sub-object symbols, complex attributes, and non-trivial event synchronization. Several case studies demonstrating that TROLL *light* is suitable for larger specifications as well have been carried out. We mention a specification environment [GC93], a medical information system [Bec93], a production cell [HV94], and a conference information system [HH94] among other smaller TROLL *light* specifications [Ehr93].

2.2 Basic Ideas of an Algebraic Semantics

The details of the algebraic semantics of TROLL *light* can only be sketched here. The full treatment is given in [GH95]. Speaking in technical terms, object signatures consists of a data signature part (with data sorts and corresponding data operators) and object specific items like object (identifier) sorts, attribute symbols, and event symbols.

Definition 1. An **object signature** $O\Sigma = (DS, \Omega, OS, A, E)$ consists of

- a set DS of data sorts,
- a family of sets of operation symbols $\Omega = \langle \Omega_{dw,ds} \rangle_{dw \in DS^*, ds \in DS}$,
- a set OS of object sorts,
- a family of sets of attribute symbols $A = \langle A_{os,w,s} \rangle_{os \in OS, w \in S^*, s \in S}$, and
- a family of sets of event symbols $E = \langle E_{os,w} \rangle_{os \in OS, w \in S^*}$

where S refers to the union of data and object sorts: $S = DS \cup OS$.

Example 2. The `Node` template induces the following object signature.

data sorts	$bool$	
operations	$false, true, bottom :\to bool$	$(f, t, \perp)$
object sorts	$node$	
attributes	$left, right : node \to node$	(l, r)
	$content : node \to bool$	(ct)
events	$create : node$	(c)
	$createLeft, createRight : node$	(cL, cR)
	$update : node \times bool$	(u)
	$destroy : node$	(d)

We abbreviate the operation names as indicated on the right hand side above. With respect to sub-object symbols as found in `Node` we assume that these are treated just like attribute symbols. The subtle distinction is discussed in [GH95].

The interpretation of data sorts, data operations, object identifier sorts, and attribute symbols characterize the states of an object community. Event symbols are associated with state transitions. Hence we define the notions of data, attribute, and event signatures as follows.

Definition 3. Let an object signature $O\Sigma$ be given. The **data signature** induced by $O\Sigma$ consists of the data sorts and the data operations of the object signature, i.e., $D\Sigma = (DS, \Omega)$.
The **attribute signature** $A\Sigma$ induced by $O\Sigma$ consists of S, i.e., the union of data and object sorts, and the union of operation and attribute symbols, i.e., $A\Sigma = (DS \cup OS, \Omega \cup A)$.
The **event signature** $E\Sigma$ induced by $O\Sigma$ consists of the attribute signature extended by one special sort symbol $\hat{e}$ for events and an appropriate family of event symbols $\hat{E}$, i.e., $E\Sigma = (DS \cup OS \cup \{\hat{e}\}, \Omega \cup A \cup \hat{E})$ with $\hat{E} = \langle \hat{E}_{os,w,\hat{e}} \rangle_{os \in OS, w \in S^*}$ and $\hat{E}_{os,w,\hat{e}} := E_{os,w}$.

The above signatures are signatures in the classical sense. Thus, they induce corresponding classes of algebras. We will use the notions data, attribute, and event algebra to refer to them. All these algebras are partial; however, we assume that the carrier sets are explicitly completed by bottom elements ($\perp$) to express that an operation is undefined.
Besides the signature part, TROLL *light* templates further include certain axioms restricting the interpretation of object signatures.

Definition 4. Let an object signature $O\Sigma$ be given. The **axioms** for object community specification are divided into constraints, derivation rules, valuation rules, interaction rules, and behavior definitions.

- A **constraint** φ is a formula restricting the class of possible attribute algebras $ALG_{A\Sigma}$.
- A **derivation rule** is of the form $\alpha = t_\alpha$ where α is an attribute and t_α is a term with the same sort as the attribute.
- A **valuation rule** is of the form $\{\varphi\}[t_{\hat{e}}]\alpha = t_\alpha$, where φ is a formula, $t_{\hat{e}}$ is an event term, α is an attribute, and t_α is a term with the same sort as the attribute α.
- An **interaction rule** is of the form $\{\varphi\}\ t_{\hat{e}} \gg t'_{\hat{e}}$ where φ is a formula, and $t_{\hat{e}}$ and $t'_{\hat{e}}$ are event terms.
- A **behavior definition** for a template T consists of a set of process definitions of the following form:
  $\Pi_0 = (\{\varphi_1\}t_{\hat{e}_1} \rightarrow \Pi_1 \mid \{\varphi_2\}t_{\hat{e}_2} \rightarrow \Pi_2 \mid \ldots \mid \{\varphi_n\}t_{\hat{e}_n} \rightarrow \Pi_n)$
  Here the Π_i's are process names, the φ_i's formulas, and the $t_{\hat{e}_i}$'s event terms.

The details of the logic for formulas φ – a variation of first-order predicate calculus formulas tailored for information system purposes — can be found in [HG94].

Valuation rules, interaction rules, and behavior definitions have been mentioned before. With constraints it is possible to specify restricting conditions on object states, and with derivation rules one describes derived attributes.
Object signatures are interpreted by so-called object communities. This name was chosen because in object communities different objects co-exist and communicate with each other.

Definition 5. Let an object signature $O\Sigma$ be given. $MOD_{O\Sigma}$ denotes the **class** of all **$O\Sigma$-models** or **$O\Sigma$-object communities**:

$$MOD_{O\Sigma} = \{M_{O\Sigma} \mid M_{O\Sigma} \subseteq (ALG_{A\Sigma} \times ALG_{E\Sigma} \times ALG_{A\Sigma})\}$$

We may assume the interpretation of the data signature to be fixed. It is also possible to restrict the interpretation of object identifier sorts in the sense that object identifiers will be chosen from a term algebra induced by the attribute signature. Then, only attributes contribute to states. For details we have to refer to [GH95] again.
Intuitively, triples $\langle A_L, \hat{A}, A_R\rangle$ in $M_{O\Sigma}$ express that there is a state transition from attribute algebra A_L to attribute algebra A_R via the occurence of the events in $\hat{A}$. A step in an object's life can be traced along such triples: For example, an object $obj \in A_{L,os}$ for some $os \in OS$ can have for attribute α the value c_1 in A_L $[\alpha_{A_L}(obj, \ldots) = c_1]$; this can change in A_R to c_2 $[\alpha_{A_R}(obj, \ldots) = c_2]$ due to the occurrence of event e in $\hat{A}$ $[e_{\hat{A}}(obj, \ldots) \neq \bot]$.

Example 6 (Object community). In Fig. 3 we have sketched part of an object community for our running example. Each arrow stands for a triple $\langle A_L, \hat{A}, A_R\rangle$. Event algebras are depicted next to arrows by giving only the elements of $\hat{e}$

different from $\perp$. Let us explain some details a bit more. (1) The transition along $\{cR(i), c(r(i))\}$ stands for the event $cR(i)$ occurring in i and the create event $c(r(i))$ occurring in $r(i)$. (2) The transition along $\{d(l(i)), d(r(i))\}$ stands for the simultaneous occurence of two delete event in the respective objects. Of course much more transitions than the depicted ones are possible.

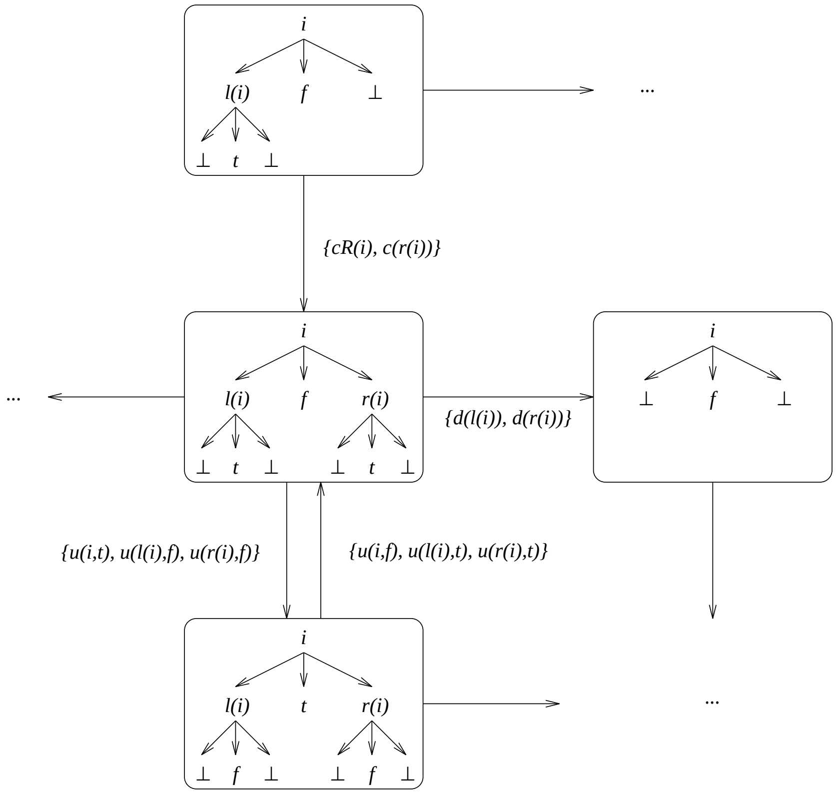

Fig. 3. Part of an example for a simple object community

In the graphical representation, left arrows represent the function $left_A$, middle arrows the function $content_A$, right arrows the function $right_A$, and the undefined object $\perp$ is duplicated as it is needed.

The possible state transition relation must obey the specified axioms. Therefore one defines the validity of axioms as follows.

Definition 7. Let an object signature $O\Sigma$, an $O\Sigma$-model $M_{O\Sigma}$ and axioms over $O\Sigma$ be given. The **validity** of axioms is defined as follows.

- A constraint φ is **valid** in $M_{O\Sigma}$ iff for all triples $\langle A_L, \hat{A}, A_R \rangle$ in $M_{O\Sigma}$ the formula φ holds in A_L and A_R.
- A derivation rule $\alpha = t_\alpha$ is **valid** in $M_{O\Sigma}$ iff for all triples $\langle A_L, \hat{A}, A_R \rangle$ in $M_{O\Sigma}$ attribute α evaluates in A_L and A_R to the value of t_α.

- A valuation rule $\{\varphi\}[t_{\hat{e}}]\alpha = t_\alpha$ is **valid** in $M_{O\Sigma}$ iff for all triples $\langle A_L, \hat{A}, A_R\rangle$ in $M_{O\Sigma}$ the following is true: If the formula φ holds in A_L and the event $t_{\hat{e}}$ occurs[2] in $\hat{A}$, then the attribute α evaluates in A_R to the value which t_α had in A_L.
- An interaction rule $\{\varphi\}\ t_{\hat{e}} \gg t'_{\hat{e}}$ is **valid** in $M_{O\Sigma}$ iff for all triples $\langle A_L, \hat{A}, A_R\rangle$ in $M_{O\Sigma}$ the following holds: If φ is true in A_L and $t_{\hat{e}}$ occurs in $\hat{A}$, then $t'_{\hat{e}}$ must also occur in $\hat{A}$.
- A behavior definition
  $\Pi_0 = (\{\varphi_1\}t_{\hat{e}_1} \to \Pi_1 \mid \{\varphi_2\}t_{\hat{e}_2} \to \Pi_2 \mid ... \mid \{\varphi_n\}t_{\hat{e}_n} \to \Pi_n)$
  for template T is **valid** in $M_{O\Sigma}$ iff for all triples $\langle A_L, \hat{A}, A_R\rangle$ in $M_{O\Sigma}$ the following holds: If an event e of template T occurs in $\hat{A}$ for an object o of object sort t, then there is a process definition and an index j such that $t_{\hat{e}_j}$ evaluates in $\hat{A}$ to e, φ_j is true in A_L, $\mathit{process\_state}(o) = \Pi_0$ in A_L, and $\mathit{process\_state}(o) = \Pi_j$ in A_R. In order to handle behavior definitions in a correct way, the object signature $O\Sigma$ has to be extended by attributes $\mathit{process\_state} : os \to \mathit{process\_state\_sort}$ for every object sort $os \in OS$.

TROLL *light* does not have a loose semantics in the sense that a specification is associated with a class of models, but the semantics of a TROLL *light* text is one distinguished object community. Again, the details can be found in [GH95].

3 Method

3.1 Development of an Initial TROLL *light* Specification

The entire design of an information system requires some separate but dependent design steps. The first one is the requirements analysis in which the UoD must be thoroughly analyzed. After having fixed *what* is to be modeled we must proceed *how* it has to be modeled. For convenience, the model used for conceptual design should be as close to the UoD as possible.
The step from an informal description of the UoD to a first formal specification is difficult. Therefore we are interested in some guidelines how to obtain object descriptions which reflect the UoD as adequately as possible. We propose the following sequence of design steps.

1. **Separation of data and objects:** We will often have the choice to model a real-world entity either as a complex value, i.e., data, or as an object. In general, there are some indicators for the modeling as an object, such as object sharing, updates on objects, and cyclic object modeling [Hul89, LR89]. In summary, the first step determines what templates we have to write.
2. **Building an object hierarchy:** Objects must be organized into an object hierarchy. The topmost object of this hierarchy is a schema object. To fix sub-object relationships we can proceed either top-down, bottom-up or in yoyo-like way.

[2] We say an event e occurs in $\hat{A}$ if e evaluates to something different from $\perp$ in $\hat{A}$.

3. **Modeling object properties:** Observable properties of objects are modeled by attributes. Here a number of choices can be made: Data-valued attributes describe printable information determined by the data types; relationships between objects can be given by object-valued attributes or by object sorts as parameters for attributes.
4. **Modeling of events:** Events are abstractions of state-modifying operations on objects. Events can have data- or object-valued parameters.
5. **Specification of the effect of events:** The effect of event occurrences on attributes is specified by valuation rules. Result terms, i.e., right-hand sides of valuation rules, are evaluated in the state before the event occurs.
6. **Synchronization of events:** The synchronization of events in different objects is specified by interaction rules. This is the most important relationship between objects as far as dynamic behavior is concerned. The synchronization rules fix the event sets that cause system transitions.
7. **Determining object integrity:** Object integrity is addressed by specifying possible event sequences, by imposing integrity conditions on object states, and by giving preconditions for the application of valuation or interaction rules. The possible event sequences in an object's life starting with a `BIRTH` and possibly ending with a `DEATH` event are given as CSP-like processes. The specified valuations and interactions are only applied if their preconditions evaluate to true in the actual state; otherwise the event has no updating or calling effect.

We cannot expect that one single pass through all these steps yields a satisfactory result. Sometimes we have to go back and repeat some steps until a certain degree of correspondence to the requirements is achieved. This is supported by animating templates as described below.

3.2 Transforming Descriptive TROLL *light* Specifications

The general idea of this part of our development method is to perform several transformation steps on the initial TROLL *light* specification in order to end up with a specification which is as close as possible to an executable program. Thus, we want to achieve transformations $T_1, ..., T_n$ related to the initial TROLL *light* specification as follows: $spec_0 \stackrel{T_1}{\to} spec_1 \stackrel{T_2}{\to} ... \stackrel{T_n}{\to} spec_n$. In each step T_i, more and more "descriptive" TROLL *light* features should be replaced by more "operational" ones. For the concrete language elements of TROLL *light* this means the following:

1. **Constraints:** The general idea is to check the constraints as locally as possible. Thus one does not want all constraints to be checked globally for one object community state, but one should transform constraints equivalently by moving them into pre-conditions of behavior patterns. By doing so, one avoids unneccesary computations in the case that a follow-up state of an (executed) event or event set does not satisfy the constraints and, what is even more important, the applicability of events can be checked effciently.

2. **Valuations:** For the valuation rules, the pre-conditions should be transformed in order to achieve a deterministic object community so that one gets closer to conventional programs. This means that each two pre-conditions of the same event have to be checked whether they exclude each other, and if not so, they have to be transformed equivalently so that they do not hold for the same cases.
3. **Interaction and behavior:** In principle, the same strategy as for valuation rules should be applied here. Non-overlapping interaction rules and behavior patterns (non-overlapping with respect to the pre-conditions) guarantee an object community with a smaller degree of non-determinism.

We have already started to incorporate classical, semi-formal approaches to software engineering like the Entity-Relationship approach [GHC+94] into our method. But more such approaches should be covered, especially Petri Nets [Rei85] seem to be a good candidate to extend structural modelling techniques with dynamic concepts.

4 Tools

We shortly mention three tools for the development environment of TROLL *light* specifications: The animator, the proof support system, and a transformation tool. The first two (together with a context-free and -sensitive syntax checker) have been implemented, and the third one is planed.

4.1 Animating Specifications

A template describes structural and dynamic aspects of a prototypical object. Structural properties are centered around the specification of possible attribute states, dynamic properties around the specification of possible event sequences. Looking at a template with sub-object slots the prototypical object described by this template is in fact an object community where events in different objects may be synchronized by interaction rules. TROLL *light* object descriptions abstract from the causality or initiative of event occurrences. Hence making our algebraic transition system to move means to indicate certain event occurrences from the outside of the system. This is what we call *animation* of templates on the basis of their algebraic semantics. By this, a specification of a formal conceptual schema can be validated against the informal system requirements.
Animating templates. Animation of templates may help to assure that the specified behavior of objects or object communities matches the required behavior. Of course, animation of a conceptual model shows the same problems like testing an implementation. From observations that the observed behavior agrees with the requirements we would like to draw the conclusion that a conceptual schema is correct with respect to the requirements. This, however, cannot be done, because there still may be some traces in the animation which have not been tested and which may show the opposite effect. Hence animation like testing an implementation is useful to falsify informal correctness.

Requirements for a tool supporting animation. A software system with the aim to support the animation of templates should meet the following requirements. It should support (1) the exploration of actual states of an object community, (2) the specification of event occurrences for initiating state transitions, and (3) the visualization of state changes.

To be more precise exploration of states should be assisted by means to illustrate the actual global structure of an object community, to show the attribute state (and optionally the behavior state) of a single object, to traverse sub-object relationships or object-valued attributes, and eventually to formulate ad hoc queries against object states.

With respect to the specification of event occurrences it must be possible to indicate event occurrences for possibly more than one object at a time. The system should help to find event sets being closed against synchronization. Hence, when a specified event occurrence provokes a second event occurrence in another object, this event occurrence should be added automatically to the current event set.

State changes provoked by a given closed event set, as there might be insertions and deletions of objects or attribute updates, should be made visible to the user, for example by appropriate messages in a specific window. When a desired state transition cannot be carried out, for instance because the resulting state violates some integrity constraints, these conditions should also be reported to the user.

Architecture of the animation system. First of all the animation system consists of a *template dictionary* which is a persistent store for object descriptions. The template dictionary is not an exclusive part of the animation system but an integral part of the whole development environment, for instance shared by the parser and the proof support system. The second basic component of the animation system is the *object base* which is a persistent store to hold object states. In principal, the object base could also be a transient store but in the system used for implementation there was only a slight distance between transient and persistent structures so that we preferred persistent structures. Hence it is possible to stop a current animation session at one time and start it again later on.

Object descriptions contain terms and formulas of the TROLL *light* query calculus which are evaluated by the third component of the animation system, the *term evaluator*. The evaluation of terms and formulas generally depends on the current state of an object community so that the term evaluator must be able to access the object base. The *execution module* is the heart of the animation system. Its task is to compute for a given set of event occurrences a successor state or to report errors if such a state cannot be determined for different reasons. Finally the *user interface* establishes communication with the human operator.

4.2 The Proof Support System

The proof support system has the aim to verify properties of specified objects on the level of specification. This seems to be useful in order to check whether the abstract specification meets the intended properties. Hopefully, misdevelopments based on inadequate specifications are avoided by verifying such properties directly from the specification.

Verifying properties. Our verification calculus is embedded into the development environment by the following steps depicted in Fig. 4:

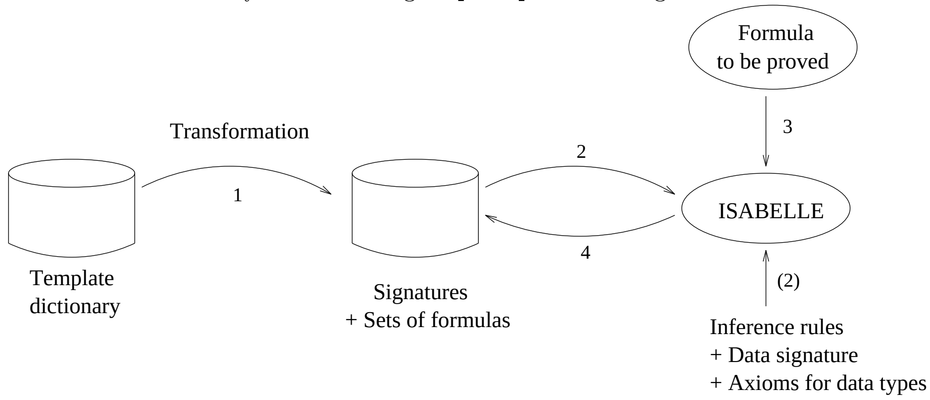

Fig. 4. Working with the proof support system.

1. Automatically transformating a TROLL *light* specification yields a corresponding template signature together with a set of formulas giving a description of the specified objects by means of the verification calculus and therefore in terms of logic. Results of these transformations are stored in a database.
2. After the transformation of a TROLL *light* specification has been completed, a generic theorem prover, in our case ISABELLE [Pau90], can be activated. In order to use this theorem prover for the verification calculus its syntax and the appropriate inference rules must be loaded into the prover (in fact we can generate an instance of ISABELLE in which these parts are already included). Next, we have to include the signature and the formulas obtained from the TROLL *light* specification to be investigated.
3. Then we can specify a property to be proven as a formula of the verification calculus. For such a formula the prover can be employed. Typically, the proof can be controlled in an interactive way, i.e., we can carry out the proof step by step by applying single inference rules or we can cause ISABELLE to automatically run parts of the proof by choosing suitable proof tactics.
4. The corresponding formula can be added to the database, if the proof of a property was successful. Proceeding this way, the formula can be reused later in proving further properties of the same object for which is was shown to be true originally.

4.3 Supporting Specification Transformation

The method described above for the translation of more "descriptive" TROLL *light* specifications into more "operational" ones can be supported by an appropriate tool. This tool should assist the developers especially in modifying formulas and process definitions. We can identify two important sub-tasks.

1. **Transformation of constraints:** In order to transform constraints into pre-conditions we must again distinguish between two cases: Treatment of attributes and treatment of sub-objects.
   Attributes: For each attribute appearing in a constraint, the tool must discover the valuation rules where the attribute is manipulated. From there one can fix the events manipulating the attribute, and the tool can point to the pre-conditions of interaction formulas and behavior definitions for the respective events. Possibly, the tool can try to integrate the constraint into these pre-conditions.
   Sub-objects: For a sub-object appearing in a constraint the tool can detect the interaction rule where the sub-object is created and fix from this point the event responsible for creation of the sub-object. The original constraint should be integrated into the pre-condition of the responsible event.
2. **Transformation of process definitions:** The tool should try to make process definitions as deterministic as possible. For example in the simple case that we have as part of a process definition $\{\varphi_1\}e \to \Pi \mid \{\varphi_2\}e \to \Pi$ this can be transformed equivalently into $\{\varphi_1 \vee \varphi_2\}e \to \Pi$. The tool should also give hints to apply rules like the above one to more involved cases.

5 Conclusion

We have explained an algebraic development technique for information systems. The technique consists of three parts: a formalism, a method, and an environment providing tools for the formalism and method. We have indicated how the relationship and dependencies between these three parts in principle look like. Of central concern was the algebraic semantics of our formalism. This semantics took deep influence on the creation of the method and the implemented and planned tools.
However, our approach is far from being perfect. Further case studies have to be carried out in order to get more insight of and criticism for our method. The existing tools can certainly be improved and new ones can be added. From such experience further improvement of our specification language TROLL *light* is expected.

References

[AAR94] R. Achuthan, V.S. Alagar, and T. Radhakrishnan. An Object-Oriented Framework for Specifying Reactive Systems. In V.S. Alagar and R. Missaoui, editors, *Proc. Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94)*, pages 18–30. Université du Quebéc à Montréal, 1994.

[BBE+90] G. v. Bochmann, M. Barbeau, M. Erradi, L. Lecomte, P. Mondain-Monval, and N. Williams. Mondel: An Object-Oriented Specification Language. Département d'Informatique et de Recherche Opérationnelle, Publication 748, Université de Montréal, 1990.

[Bec93] E. Becker. The Electronic HDMS-A Patient's Record – A TROLL *light* Case Study. Project Thesis, TU Braunschweig, 1993.

[Bre91] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*. Springer, Berlin, LNCS 562, 1991.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, TU Braunschweig, 1992.

[CJO94] S. Clerici, R. Jimenez, and F. Orejas. Semantic Constructions in the Specification Language GLIDER. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification (WADT'92)*, pages 144–157. Springer, Berlin, LNCS 785, 1994.

[DG94] P. Dauchy and M.-C. Gaudel. Algebraic Specifications with Implicit State. Technical Report 887, Université de Paris-Sud, 1994.

[DO91] O.-J. Dahl and O. Owe. Formal Development with ABEL. Technical Report 159, University of Oslo, 1991.

[Ehr93] H.-D. Ehrich, editor. *Beiträge zu* KorSo*– und TROLL light–Fallstudien*. Technische Universität Braunschweig, Informatik-Bericht, 93–11, 1993.

[EO94] H. Ehrig and F. Orejas. Dynamic Abstract Data Types: An Informal Proposal. *EATCS Bulletin*, 53:162–169, 1994.

[ESD93] ESDI. *OBLOG CASE Version 1.0*. Espirito Santo Data Informatica, Lisbon, Portugal, 1993.

[Gab93] P. Gabriel. The Object-Based Specification Language Π: Concepts, Syntax, and Semantics. In M. Bidoit and C. Choppy, editors, *Proc. 8th Workshop on Abstract Data Types (ADT'91)*, pages 254–270. Springer, Berlin, LNCS 655, 1993.

[GC93] M. Gogolla and I. Claßen. An Object-Oriented Design for the ACT ONE Environment. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Proc. 3rd Int. Conf. on Algebraic Methodology and Software Technology (AMAST'93)*, pages 361–368. Springer, London, Workshops in Computing, 1993.

[GCH93] M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL *light*. In A. Miola, editor, *Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93)*, pages 17–32. Springer, Berlin, LNCS 722, 1993.

[GD94] J.A. Goguen and R. Diaconescu. Towards an Algebraic Semantics for the Object Paradigm. In H. Ehrig and F. Orejas, editors, *Proc. 9th Workshop on Abstract Data Types (ADT'92)*, pages 1–29. Springer, Berlin, LNCS 785, 1994.

[GH95] M. Gogolla and R. Herzig. An Algebraic Semantics for the Object Specification Language TROLL *light*. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Proc. 10th Workshop on Abstract Data Types (ADT'94)*, pages 288–304. Springer, Berlin, LNCS 906, 1995.

[GHC+94] M. Gogolla, R. Herzig, S. Conrad, G. Denker, and N. Vlachantonis. Integrating the ER Approach in an OO Environment. In R. Elmasri, V. Kouramajian, and B. Thalheim, editors, *Proc. 12th Int. Conf. on the Entity-Relationship Approach (ER'93)*, pages 376–389. Springer, Berlin, LNCS 823, 1994.

[GJM91] G. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall (NJ), 1991.

[GM87] 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.

[HG94] R. Herzig and M. Gogolla. A SQL-like Query Calculus for Object-Oriented Database Systems. In E. Bertino and S. Urban, editors, *Proc. Int. Symp. on Object-Oriented Methodologies and Systems (ISOOMS'94)*, pages 20–39. Springer, Berlin, LNCS 858, 1994.

[HH94] J. Hackauf and N. Hartmann. Specification of an Information System for Organizing Conferences – A TROLL *light* Case Study. Project Thesis, TU Braunschweig, 1994.

[Hul89] R. Hull. Four Views of Complex Objects: A Sophisticate's Introduction. In S. Abiteboul, P.C. Fischer, and H.J. Schek, editors, *Nested Relations and Complex Objects in Databases*, Springer, Berlin, LNCS 361, pages 87–116, 1989.

[HV94] R. Herzig and N. Vlachantonis. TROLL light — Specification with a Language for the Conceptual Modelling of Information Systems. In C. Lewerentz and T. Lindner, editors, *Case Study "Production Cell": A Comparative Study in Formal Specification and Verification*, pages 231–239. FZI-Publication 1/94, FZI, Karlsruhe (Germany), 1994.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91–04, Technische Universität Braunschweig, 1991.

[LR89] C. Lécluse and P. Richard. Modeling Complex Structures in Object-Oriented Databases. In *Proc. 8th ACM Symp. Principles of Database Systems*, pages 360–368, 1989.

[MQ93] J. Meseguer and X. Qian. Logic-Based Modeling of Dynamic Object Systems. In P. Bunemann and S. Jajodia, editors, *Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of Data*, pages 89–98. SIGMOD Record Vol. 22, Number 2, June 1993.

[MR91] J. McDermid and P. Rook. Software Development Process Models. In J. McDermid, editor, *Software Engineer's Reference Book*, chapter 15, pages 15/1–15/36. Butterworth-Heinemann, Oxford, 1991.

[Pau90] L.C. Paulson. Isabelle: The Next 700 Theorem Provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* EATCS Monographs on Theoretical Computer Science. Springer, Berlin, 1985.

[SSE87] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P.M. Stoecker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, pages 107–116. VLDB Endowment Press, Saratoga (CA), 1987.

[Wie91] R. Wieringa. Equational Specification of Dynamic Objects. In R.A. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design & Construction (DS-4), Proc. IFIP WG 2.6 Working Conference, Windermere (UK) 1990*, pages 415–438. North-Holland, 1991.

[Wir90] M. Wirsing. Algebraic Specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 677–788. North-Holland, Amsterdam, 1990.