Validating Database Constraints and Updates Using Automated Reasoning Techniques*

Remco Feenstra and Roel Wieringa
Department of Mathematics and Computer Science
Vrije Universiteit
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
email: rbfeens@cs.vu.nl, roelw@cs.vu.nl

November 23, 1995

Abstract

In this paper, we propose a new approach to the validation of formal specifications of integrity constraints. The validation problem of formal specifications consists of assuring whether the formal specification corresponds with what the domain specialist intends. This is distinct from the verification problem, which is the problem whether an implementation (which is a formal object) corresponds with a specification (which is also a formal object). We consider formal specifications of object-oriented database systems that are subject to static and dynamic integrity constraints. To validate that such a specification expresses what we intend, we propose a system that can answer *reachability queries*, in which it is asked whether the system can evolve from one state into another without violating the integrity constraints. If the query is answered positively, the system should exhibit an example path between the two states; if the answer is negative, the system should explain why this is so. We discuss the use of planning and theorem-proving techniques to answer such queries, illustrating their application to reachability queries relevant for database system validation.

Keywords: Database dynamics and behaviour, constraints and updates, validation of conceptual models.

1 Introduction

In any system development process, three tasks are performed: determining the required function of the system, modeling the external behavior of the system that would realize this function, and implementing this behavior. (In an ideal development process, these tasks are performed in the order just mentioned, but in practice, there will be interleaving and iteration between these tasks.) We call a model of external system behavior a **conceptual model** (CM) of the system. In many cases, a CM is required to be *formal*. On the other hand, the statement of the required function of the system is usually *informal*. This means that the transition from requirements to a conceptual model involves a transition from informal to formal specifications. In this paper, we are concerned with checking whether the formal CM matches the informal understanding of the system, also known as **validation** of the CM. Once we have obtained a validated CM, we (ideally) only have to transform the formal specification of the CM into an implementation (which is a low-level formal specification). **Verification**, which consists of checking whether a specification

*This research is partially supported by the Netherlands Organization for Scientific Research (NWO), SION project 612–317–408 and the Esprit Basic Research Action IS-CORE (working group 6071).

developed in a later phase corresponds to a specification developed in an earlier phase, can then be carried out by formal reasoning (at least in principle).

To benefit most from validation, it should be carried out as early as possible during development. But in this case, the modeled system does not yet exist in reality: it "exists" only informally in the minds of domain specialists. This situation precludes direct empirical validation of the CM. Instead, an analyst needs to adopt an indirect approach by soliciting comments from domain specialists as to whether the formal CM specification conforms to their ideas about the system. When trying to do so, the analyst encounters the problem that domain specialists with little background in formal specification often don't understand the formal CM specification. When the formal meaning of the specification differs from the analyst's idea of what was specified, for example due to mistakes, the analyst may have problems too in dealing with the formal meaning of the specification.

Studying concrete examples of the specified behavior can help in understanding the specification. An implementation of the system can provide these examples. To reduce the costs of creating such a system, we are often satisfied with a **prototype**: a low-cost, partial implementation that only is concerned with the aspects of the behavior of the system needed for answering a specific validation question. If we specify the CM in an executable specification language, a prototype is easily obtained. Independently from the way we produce the prototype, we can enhance the understanding reached by means of the prototype, by embellishing the runs of the prototype with **explanations** of the runs in terms of the CM. We call such explained runs of the system an **animation** of the system. The explanation of a system run may take the form of a nice graphical presentation of the run. For example, object behavior may be represented by finite state automatons, and object state may be represented in tabular or other form. This way, an prototype-based animator resembles a graphical debugger for formal specifications, that repeatedly shows the effects of a transaction selected by the analyst.

In this paper, we propose an approach to animation that can supplement and enhance the power of graphical animations. We argue that a system for creating animations capable of *reasoning about reachability properties* between system states would reduce two common problems of the prototype-based approach to animation mentioned above, viz.

(1) *(Relevance of information)* A snapshot of a system state of a system of any complexity, in whatever way it is presented, shows many irrelevant details, that distract the viewer from the properties he should watch carefully. The analyst can reduce the number of irrelevant details by instructing the animation system what to show and what not, but this requires so much effort that it is not practical.

(2) *(Reasoning about scenarios)* Simple animations of prototypes only consist of repeatedly executing transactions of the system on demand, and showing their effects on the system state. The analyst must still provide suitable starting states and select transactions demonstrating interesting behavior. Again, this requires much effort. In addition, it may cause the analyst to ignore some examples of behavior that can occur according to the specification, but not according to the analyst's intuition of the system.

In our approach, instead of basing an animation on showing the effects of transactions selected by the analyst on some example state, the analyst formulates properties of interesting behavior in terms of queries about the reachability of system states from other states under constraints. Software capable of automatically solving these queries will then generate examples of states and transactions that demonstrate behavior exhibiting these properties. This reduces problem (1) by enabling the animation system to derive relevance judgements about the information to be presented in the presentation, because it knows the intended context via the query. Problem (2) is reduced because more of the reasoning involved in finding examples of interesting behavior is automated. Reasoning about reachability properties is also important in analyzing safety properties of the system.

In general, solving reachability queries is undecidable. Fortunately, in the context of validation of database systems, we can make some reasonable assumptions such that techniques from planning

and model generation become applicable. We expect that an animation system, based on these techniques, will enable an animation system to solve simple cases of reachability queries that the analyst can use the practice of validation.

In section 2, we present a running example in situation calculus, the specification language chosen in this paper. In addition, a number of interesting reachability queries are presented and motivated. Reachability questions have been studied thoroughly in Petri net theory [13], but they are less well-studied in a logic-based context. Nevertheless, there are other sources for inspiration, such as plan generation techniques from AI and theorem-proving techniques for first-order logic. Section 3 discusses the ideas that we have borrowed from these fields in our approach to solving reachability queries. In addition, section 3 gives an outline for a procedure for solving reachability queries. The research reported on in this paper is still in the starting stage, and in section 4 we discuss the feasibility of our approach for the improvement of animation techniques, and list some topics for current and further research.

2 Reachability queries

Figures 1 and 2 show part of a specification of a simple database system about a library, that will be used as running example. The specification language used in this paper is sorted first-order predicate logic, in particular the situation calculus [11]. This is not essential to our approach, but it simplifies the presentation of the reachability queries. In practice, the specification will be written in a language like LCM [4], which is a syntactically sugared version of Dynamic Object Logic (DOL)[18]. In the next couple of paragraphs, we informally explain the declarative semantics of the specification. A more formal presentation of the semantics of LCM is given elsewhere [17].

A specification consists of a collection of class specifications, each of which contains minimally the declaration of a number of **object identifier sorts**. The elements of an identifier sort are closed terms, that are used as globally unique proper names for the objects about which we want to reason. For example, `MEMBER` is the set of infinitely many identifiers of member objects. The elements of `MEMBER` can informally be thought of as internal, system generated object identifiers (oids). In any particular state of the world, a number of members exist. This is represented by the predicate `Exists`, that is true of exactly those oids that identify existing objects. The set of `MEMBER` instances thus includes all actually existing and all *potential* member oids. Given a system state s_0, the formula

$$\exists \mathtt{m} : \mathtt{MEMBER}\ \mathtt{Exists}(\mathtt{m}, s_0)$$

selects, out of all possible member oids, a member that exists in state s_0.

There is in addition a predicate `Used`, that is true of those oids that have ever been used to identify an existing object. The `Used` predicate is used to prevent reuse of oids, which in turn is needed to guarantee that oids are globally unique proper names.

In addition to the identifier sorts, there are two built in sorts `STATE` and `TRANSACTION`. The specification declares two kinds of predicates, **state-dependent** and **state-independent** ones. The valuation of state-independent predicates depends on the state and may be changed by actions. Because we need to represent them in a finite way easily, we require that all state-dependent predicates have a finite extension, including the special state-dependent predicates `Used` and `Exists`.

The valuation of state-independent predicates, like the equality predicate, is independent of the current state. Their extension may be infinite, as long as we can derive their properties by computation. Functions whose result sort is `TRANSACTION` are called **transactions**. Example transactions are `borrow` and `return`. There is a built-in state-dependent predicate `Possible(TRANSACTION, STATE)`, that says when an action can be performed in a state. This predicate can be used to specify transaction preconditions. Finally, there is a function

$$\mathtt{do} : \mathtt{TRANSACTION} \times \mathtt{STATE} \rightarrow \mathtt{STATE}$$

that describes the effect of executing a transaction in a state.

Object Identifier Sorts:
MEMBER, PASS, COPY
Other Sorts:
STRING and the special built-in sorts STATE and TRANSACTION
State-dependent Predicates:
Special built-in predicates Possible(TRANSACTION, STATE)
Exists(MEMBER, STATE) and Used(MEMBER, STATE), similarly for PASS and COPY
Borrowing(MEMBER, COPY, STATE), Reservation(MEMBER, COPY, STATE)
MemberPass(MEMBER, PASS, STATE), MemberFine(MEMBER, STATE)
State-independent Predicates:
Equality predicate for the ADTs.
Transactions:
new_member(MEMBER, PASS), borrow(MEMBER, COPY), return(MEMBER, COPY)
overdue(MEMBER, COPY), pay(MEMBER), reserve(MEMBER, COPY)
get_reserved_copy(MEMBER, COPY)
Static constraints:
At each moment, each member has exactly one pass
$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \texttt{Exists}(m, s) \rightarrow \exists p : \texttt{PASS}\ \texttt{Exists}(p, s) \wedge \texttt{MemberPass}(m, p, s)$
$\forall s : \texttt{STATE}\ \forall p : \texttt{PASS}\ \forall m1, m2 : \texttt{MEMBER}$
$\texttt{Exists}(p, s) \wedge \texttt{MemberPass}(m1, p, s) \wedge \texttt{MemberPass}(m2, p, s) \rightarrow m1 = m2$
A copy can be borrowed by at most one member at a time.
$\forall s : \texttt{STATE}\ \forall c : \texttt{COPY}\ \forall m1, m2 : \texttt{MEMBER}\ \texttt{Borrowing}(m1, c, s) \wedge \texttt{Borrowing}(m2, c, s) \rightarrow m1 = m2$
A copy can be reserved by at most one member at a time.
$\forall s : \texttt{STATE}\ \forall c : \texttt{COPY}\ \forall m1, m2 : \texttt{MEMBER}\ \texttt{Reservation}(m1, c, s) \wedge \texttt{Reservation}(m2, c, s) \rightarrow m1 = m2$
The Used predicate captures at least all currently existing instances (similar axioms for PASS and COPY)
$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \texttt{Exists}(m, s) \rightarrow \texttt{Used}(m, s)$
Each state dependent predicate involving identifier sorts, can only hold if Exists holds for all these identifiers (similar axioms hold for Borrowing, Reservation and MemberFine)
$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \forall p : \texttt{PASS}$
$\texttt{MemberPass}(m, p, s) \rightarrow \texttt{Exists}(m, s) \wedge \texttt{Exists}(p, s)$

Figure 1: A specification of a simple library database system in situation calculus (first part). A specification of all identifier sorts and datatypes is omitted.

Dynamic constraints:

Necessary precondition for success axioms for the transactions:

* transaction `borrow`:

- the member and the copy must exist (similarly for the other transactions except `new_member`).

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \forall c : \texttt{COPY}\ \texttt{Possible}(\texttt{borrow}(m, c), s) \rightarrow \texttt{Exists}(m, s) \wedge \texttt{Exists}(c, s)$

- the member has no outstanding fines

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \forall c : \texttt{COPY}\ \texttt{Possible}(\texttt{borrow}(m, c), s) \rightarrow \neg\texttt{MemberFine}(m, s)$

- the copy is available

$\forall s : \texttt{STATE}\ \forall m, n : \texttt{MEMBER}\ \forall c : \texttt{COPY}\ \texttt{Possible}(\texttt{borrow}(m, c), s) \rightarrow \neg\texttt{Borrowing}(n, c, s)$

- the copy is not reserved

$\forall s : \texttt{STATE}\ \forall m, n : \texttt{MEMBER}\ \forall c : \texttt{COPY}\ \texttt{Possible}(\texttt{borrow}(m, c), s) \rightarrow \neg\texttt{Reservation}(n, c, s)$

* transaction `return`:

- There must be an outstanding borrowing of the copy by the member returning it.

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \forall c : \texttt{COPY}\ \texttt{Possible}(\texttt{return}(m, c), s) \rightarrow \texttt{Borrowing}(m, c, s)$

* transaction `new_member`:

- the member and his/her pass must be fresh

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \forall p : \texttt{PASS}\ \texttt{Possible}(\texttt{new\_member}(m, p), s) \rightarrow \neg\texttt{Used}(m, s) \wedge \neg\texttt{Used}(p, s)$

* transaction `pay`:

- There must exist an outstanding fine for the member

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \texttt{Possible}(\texttt{pay}(m), s) \rightarrow \texttt{MemberFine}(m, s)$

* transaction `reserve`:

- The copy must not be already reserved.

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \forall c : \texttt{COPY}\ \texttt{Possible}(\texttt{reserve}(m, c), s) \rightarrow \forall n : \texttt{MEMBER}\ \neg\texttt{Reservation}(n, c, s)$

* transaction `get_reserved_copy`:

- the copy must be reserved by the member

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ c : \texttt{COPY}\ \texttt{Possible}(\texttt{get\_reserved\_copy}(m, c), s) \rightarrow \texttt{Reservation}(m, c, s)$

Transaction effect axioms:

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \forall c : \texttt{COPY}$
$\texttt{Possible}(\texttt{borrow}(m, c), s) \rightarrow \texttt{Borrowing}(m, c, \texttt{do}(\texttt{borrow}(m, c), s))$

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \forall c : \texttt{COPY}$
$\texttt{Possible}(\texttt{return}(m, c), s) \rightarrow \neg\texttt{Borrowing}(m, c, \texttt{do}(\texttt{return}(m, c), s))$

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \forall p : \texttt{PASS}$
$\texttt{Possible}(\texttt{new\_member}(m, p), s) \rightarrow \texttt{Exists}(m, s) \wedge \texttt{Exists}(p, \texttt{do}(\texttt{new\_member}(m, p), s))$
$\wedge\texttt{Used}(m, \texttt{do}(\texttt{new\_member}(m, p), s)) \wedge \texttt{Used}(p, \texttt{do}(\texttt{new\_member}(m, p), s))$
$\wedge\texttt{MemberPass}(m, p, \texttt{do}(\texttt{new\_member}(m, p), s))$

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \forall c : \texttt{COPY}$
$\texttt{Possible}(\texttt{reserve}(m, c), s)) \rightarrow \texttt{Reservation}(m, c, \texttt{do}(\texttt{reserve}(m, c), s))$

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \forall c : \texttt{COPY}$
$\texttt{Possible}(\texttt{get\_reserved\_copy}(m, c), s) \rightarrow \neg\texttt{Reservation}(m, c, \texttt{do}(\texttt{get\_reserved\_copy}(m, c), s))$
$\wedge\ \texttt{Borrowing}(m, c, \texttt{do}(\texttt{get\_reserved\_copy}(m, c), s))$

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}$
$\texttt{Possible}(\texttt{pay}(m), s) \rightarrow \neg\texttt{MemberFine}(m, \texttt{do}(\texttt{pay}(m), s))$

$\forall s : \texttt{STATE}\ \forall m : \texttt{MEMBER}\ \forall c : \texttt{COPY}$
$\texttt{Possible}(\texttt{overdue}(m, c), s) \rightarrow \texttt{MemberFine}(m, \texttt{do}(\texttt{overdue}(m, c), s))$

Figure 2: A specification of a simple library database system in situation calculus (second part), showing its transactions.

The axioms of the specification are classified as **static constraints** and **dynamic constraints**. A static constraint is applicable to each individual state (i.e. it can be evaluated in a single state), whereas a dynamic constraint refers to the relationship between two states. Examples of dynamic constraints are the effect axioms and precondition axioms shown in figure 2.

In addition to the explicit axioms in the specification, there are implicit *frame axioms*, that say, for each action, what does not change during the action. We assume that the explicit axioms give a complete specification of what does change during an action. Informally, in the intended model, a state s_2 can be reached from another state s_1 via a transaction t, iff all preconditions for t are satisfied in s_1, both states satisfy the static integrity constraints, and the transition corresponds to a minimal change respecting the transaction effect axioms and frame axioms. Note that the static integrity constraints only restrict the applicability of transactions; they do not cause derived updates.

For each specification, we can view its intended model as a particular labeled transition system (LTS) $\mathcal{T} = \langle \mathcal{S}, \{\xrightarrow{t} | \ t \in \mathcal{L}\}\rangle$, where $\mathcal{S}$ is the set of possible valuations for equivalence classes of ground terms of type `STATE` that satisfy the integrity constraints, and the label set $\mathcal{L}$ is the set of equivalence classes of ground transaction terms, and for each $t \in \mathcal{L}$, $\xrightarrow{t} \subseteq \mathcal{S} \times \mathcal{S}$. Two terms are considered equivalent if they are equal according to the underlying ADT specification. (Equality of data values is assumed to be defined an ADT specification, not treated here.) The transitions $\xrightarrow{t}$ reflect the possibility of transactions in the states, and their effects. A **scenario** in a LTS $\mathcal{T}$ is a particular finite "run" in $\mathcal{T}$. More formally, a scenario is a finite sequence $\mathbf{s} = s_0, s_1, \ldots, s_n$ of states $s_i \in \mathcal{S}$ and a finite sequence $\mathbf{p} = t_0, t_1, \ldots, t_{n-1}$ of transitions $t_i \in \mathcal{L}$ for some integer $n \geq 0$, such that in $\mathcal{T}$, $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n$.

A **reachability query** asks for a scenario that is an example of behaviour of the specified system that satisfies additional constraints. In the rest of this paper, we will assume that $\mathcal{T}$ consists of deterministic actions only, i.e. from one state, two executions of the same action will always lead to the same result state.

A general reachability query asks for an example scenario that demonstrates the truth of an existentially qualified formula, as below, in the intended model of the specification:

$$\exists n \geq 0 \exists \mathbf{s} = \langle s_0, s_1, \ldots, s_n \rangle \exists \mathbf{p} = t_0; t_1; \cdots t_{n-1} \ s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n \wedge \Phi(\mathbf{s}, \mathbf{p})$$

In this formula, $\Phi(\mathbf{s}, \mathbf{p})$ is a situation calculus formula that expresses constraints on the states and transactions occuring in the scenario.

Below, we will show five examples of typical questions about the specified system that arise during validation of the example library specification, and discuss how they can be formulated as reachability queries.

(1) Is there a possible system state? If there is not, we have modeled a system without states, which is unlikely to be the intended system. This situation might easily occur when we specified contradictory global integrity constraints. Checking whether there is a (consistent) system state is also known as testing the satisfiability of the static constraints. To demonstrate that there is at least one possible state, we can ask for one by means of a (degenerate) reachability query:

$$\exists n \geq 0 \exists \mathbf{s} \exists \mathbf{p} \mathit{true}$$

In our case, the system might generate the scenario for $n = 0$, consisting of a single state s_0, in which the extension of all state-dependent predicates is empty. As this includes the `Exists` predicates, no objects exists in this state. Thus we can be sure that our system has at least this state. Thus, we have shown the formal property of consistency of the specification, but the informal property as whether this "empty" state is also considered an example of a library system by the domain specialists is a different matter. This can be tested by presenting this example to them and asking whether they agree that this is what they intended.

(2) Is there a system state in which a member exists? This is an example of a question that can be used to test whether the specification allows states that can occur according to the domain specialists. Usually, these states described incompletely. We can write our example question as a reachability query in the following way:

$$\exists n \geq 0 \exists \mathbf{s} = \langle s_0, s_1, \ldots s_n \rangle \exists \mathbf{p} \exists \mathtt{m} : \mathtt{MEMBER}\ \mathtt{Exists}(\mathtt{m}, s_0)$$

In this case, the system must construct a complete example state s_0, where both `Exists(m)` is true for some `m`, and all static integrity constraints are satisfied. For example, the cardinality constraint requiring exactly one `PASS` instance for each `MEMBER` instance makes inclusion of a `PASS` instance in the state necessary. A question like (2) can reasonably be asked for each class in the specification. If the system replies there is no state with an instance of the class, we have discovered a problem with our specification, since we have specified classes that never can have any instances. If this is intended, we wonder why we specified the classes in the first place. Note that a facility to give example answer to questions like (2) gives us a convenient way to create example states, without having to spell them out in full detail.

(3) Can each transaction of the specification be executed in some state? This property, also called *operation applicability* [7], can be investigated by constructing a separate reachability query for each (parametrized) transaction. For the `borrow` transaction, it might look like:

$$\exists n \geq 0 \exists \mathbf{s} = \langle s_0, s_1, \ldots s_n \rangle \exists \mathbf{p} = t_0; t_1; \cdots; t_{n-1}$$
$$\exists \mathtt{m} : \mathtt{MEMBER}\ \exists \mathtt{c} : \mathtt{COPY}\ t_0 = \mathtt{borrow}(\mathtt{m}, \mathtt{c}) \wedge n = 1 \wedge s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n$$

The system might answer that the formula is true, and demonstrate this by showing two states connected by an instance of `borrow(m,c)`. This involves finding a state satisfying all preconditions for success of the transaction and the static integrity constraints, and computing its effects according to the effect axioms. Thus, it constructs an example of an occurrence of a `borrow` transaction. Comparing this to the case of prototype-style animation, we find that the analyst no longer has to find a suitable starting state and `borrow` parameters. Besides, he no longer has to communicate these to the animator explicitly. Of course, we can also do ordinary prototype-style animation simulating the effects of a transaction step by completely specifying the transactions constituting $\mathbf{p}$ and adding constraints on $\mathbf{s}$ to restrict s_0 to one particular state.

(4) Can a certain state be reached from another? An example of this form of question occurs in the case where we agree that the "empty" state found as an answer to question (1) is indeed what we intended. Starting from this state, can we reach, by means of one or more transactions, a state in which a member has borrowed a book? This can be formulated as

$$\exists n \geq 0 \exists \mathbf{s} = \langle s_0, s_1, \ldots, s_n \rangle \exists \mathbf{p} = t_0; t_1; \cdots t_{n-1}$$
$$s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n \wedge n \geq 1 \wedge \Phi_{\text{empty}}(s_0) \wedge \exists \mathtt{m} : \mathtt{MEMBER}\ \exists \mathtt{c} : \mathtt{COPY}\ \mathtt{Borrowing}(\mathtt{m}, \mathtt{c}, s_n)$$

In this case, the system should answer that no such state can be reached. It might explain this unexpected result by pointing out that such a state can only be reached by a some `borrow(m,c)` transaction, but this action is only applicable from states where a copy exists, due to the precondition for success of `borrow`. As this is not the case in the empty state, and there are no transactions specified that can create an instance of `COPY`, we cannot find a scenario satisfying the query.

(5) According to domain specialists, a member who has an outstanding fine should not be able to obtain more copies until he has paid the fine. Does this property hold in the specified system? We will try to refute this property by asking for a scenario in which a member with an outstanding fine is able to arrive in a state where he has borrowed another copy, without paying the fine first:

$$\exists n \geq 0 \exists \mathbf{s} = \langle s_0, s_1, \ldots, s_n \rangle \exists \mathbf{p} = t_0; t_1; \cdots t_{n-1}$$
$$\exists \mathtt{m} : \mathtt{MEMBER}\ \exists \mathtt{c} : \mathtt{COPY}\ \mathtt{Exists}(\mathtt{m}, s_0) \wedge \mathtt{MemberFine}(\mathtt{m}, s_0) \wedge \neg \mathtt{Borrowing}(\mathtt{m}, \mathtt{c}, s_0)$$
$$\wedge \mathtt{Borrowing}(\mathtt{m}, \mathtt{c}, s_n) \wedge \neg \mathtt{Occurs}(\mathtt{pay}(\mathtt{m}), \mathbf{p})$$

As it turns out, there *is* a scenario that satisfies these constraints. Because the preconditions of `reserve` and `get_reserved_copy` transactions omit the check on outstanding fines, a member can still obtain more copies while having an outstanding fine, by a backdoor approach: first reserve them and then get the reserved copies. Whether this was intended by the domain specialists, and how to modify it if not, can be discussed with the domain specialists showing them this example scenario and asking for their comments.

3 Solving reachability queries

In their general form, answering reachability queries automatically is out of question because this problem is undecidable. We can, however, try to solve interesting subcases by automated reasoning techniques. We will first discuss the relation of the problem of solving reachability question with the problem of plan generation, which has received much attention in the AI literature. Then we describe planning techniques and how to adapt them for solving reachability queries.

3.1 Plan generation versus solving reachability queries

The problem of generation plans to achieve goals bears resemblance to the problem of generating scenarios as solutions to reachability queries. When we look in more detail, however, we can also see some important differences.

The plan generation problem from AI can be formulated as finding a value for a plan variable $\mathbf{p}$, consisting of a finite sequence of actions, demonstrating the truth of an existentially quantified formula of the form

$$\exists n \geq 0 \exists \mathbf{p} = t_0; t_1; \ldots; t_{n-1} \forall \mathbf{s} = \langle s_0, s_1, \ldots, s_n \rangle$$
$$s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n \wedge \Phi_{\text{initial}}(s_0) \rightarrow \Phi_{\text{final}}(s_n)$$

Compare this with finding scenarios as solutions to reachability queries, expressed as the problem of finding a value for a plan variable $\mathbf{p}$ and a trace variable $\mathbf{s}$:

$$\exists n \geq 0 \exists \mathbf{s} = \langle s_0, s_1, \ldots, s_n \rangle \exists \mathbf{p} = t_0; t_1; \cdots; t_{n-1}$$
$$s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n \wedge \Phi_{\text{initial}}(s_0) \wedge \Phi_{\text{final}}(s_n)$$

The difference can be appreciated by careful comparison of the order of quantification in these examples. It is caused by a difference in their intended application. In plan generation and the similar problem of program synthesis, our primary interest is in obtaining a plan or program that will always lead to a state satisfying Φ_{final}, when started in some state satisfying Φ_{initial}, whereas in scenario generation we are interested in a particular execution of a plan or program. Nevertheless, some planning techniques can be used to solve reachability queries, as will be argued next.

3.2 Planning techniques

Solving reachability queries involves reasoning about actions, their applicability and their effects. These themes where studied in the AI literature about problem solving, especially in the context of generating plans to achieve goals. Green[6] showed that many of these problem solving tasks can be tackled by formulating them in first-order logic using answer terms, and then applying a (resolution-style) general theorem proving program. As a side-effect of the search for a proof of the existence of a goal state, an answer is constructed in these answer terms, that is communicated to the user if the attempt succeeds.

The direct applicability of this technique is seriously hampered in practical situations by the frame problem[11]. At least two aspects of this problem are of our concern here:

- *(Specification)* To specify the result of actions explicitly, we need to write down not only the aspects of the state that are influenced by the action, but also state somehow that

everything else does *not* change due to the action. The straightforward approach to this problem consists of adding a frame axiom for each combination of a property and action that doesn't change the property, stating that after executing of the action, the property is the same as before it was executed. This requires much effort, and reduces the readability of the specification.

- *(Reasoning)* How to reason efficiently about action effects in the presence of frame assumptions? The straightforward approach mentioned above makes search by a theorem prover much more difficult, because it must consider many frame axioms that most often don't contribute to finding a proof.

When we are able to reformulate planning problems in the language of the famous STRIPS planning system[5] we deal with both problems, although in a less general framework than full situation calculus. The first problem is circumvented by specifying actions more operationally in terms of a precondition and addition and deletion operators on a *representation* of a system state, together with the assumptions that (1) these are the only effects of the action, and that (2) the state can only change by applying specified actions. The second problem is circumvented by making the frame assumptions implicit in the planning process. Given a specification in STRIPS format, we are able to reason about action preconditions and their effects by **regression** of a formula through an action. This amounts to computing symbolically the precondition that would cause the formula to hold after execution of the action. By repeated application of regression operators, a formula describing a condition on the goal state is reduced to a condition on the initial state. Then, a standard first-order theorem prover is used to test whether the plans it has constructed are executable in the given starting state, by verifying that this condition is implied by the given description of the initial state. The plan is generated from the final state towards the initial state, because the branching factor of the search tree is usually smaller in this direction than when working from the initial state towards the goal.

Problems with STRIPS-style planning are unclarities in its semantics[9] and its restricted applicability due to its required representation of actions and states. For example, in the STRIPS language it is difficult to specify an action that has different effects under different circumstances.

Pednault's ADL language[12] improves on these points by providing an exact semantics for ADL based on a subset of situation calculus, and assuming that the circumstances that cause a property to hold are completely specified. Compared to the STRIPS language, ADL allows us to express more complicated actions, while still being able to apply regression to reason about action effects.

Schubert[15] proposes an alternative to the straightforward axiomatization of the frame assumption by using **explanation closure axioms**. These give for every possible change of property an exhaustive list of actions and preconditions that can cause this change. Since most actions typically leave most properties intact, an exhaustive specification of the frame assumption in this form is more concise and manageable than the straightforward approach of supplying frame axioms.

More recently, Reiter[14] has shown how to combine Pednault's and Schubert's proposals into a convenient specification language of action effects in a subset of situation calculus. It is based on supplying a **successor state axiom** for each state-dependent predicate P and the **specification completeness assumption** that these successor state axioms capture all conditions leading to change of the updatable predicates. Assuming that we can express a sufficient precondition for success of the transaction t in a state s as a formula (abbreviated here as $SPFS(t,s)$), the successor state axiom for the state-dependent predicate P has the form

$$\forall t : \texttt{TRANSACTION}\ \forall s : \texttt{STATE}\ SPFS(t,s) \rightarrow \forall \mathbf{x}\ P(\mathbf{x}, do(t,s)) \equiv \Psi_P(\mathbf{x},t,s)$$

where $\Psi_P(\mathbf{x},t,s)$ is a formula containing no applications of `do`. For instance, it might have the form

$$\Psi_P(\mathbf{x},t,s) \equiv \gamma_P^+(\mathbf{x},t,s) \vee P(\mathbf{x},s) \wedge \neg\gamma_P^-(\mathbf{x},t,s)$$

where $\gamma_P^+(\mathbf{x}, t, s)$ exhaustively expresses all conditions that can lead P to become true after execution of transaction t, and $\gamma_P^-(\mathbf{x}, t, s)$ all conditions that cause P to become false after execution of t. Both are expressed in terms of conditions on state s.

For example, we can reformulate the ways the state-dependent predicate `Borrowing` gets its value based on the effect, precondition and (implicit) frame axioms and the assumption of completeness of specification as:

$$\forall t : \texttt{TRANSACTION}\ \forall s : \texttt{STATE}$$
$$SPFS(t, s) \rightarrow \forall m\text{: }\texttt{MEMBER}\ \forall c\text{: }\texttt{COPY}\ \texttt{Borrowing}(m, c, do(t, s)) \equiv \Psi_{\texttt{Borrowing}}(m, c, t, s)$$

with

$$\Psi_{\texttt{Borrowing}}(m, c, t, s) \equiv \gamma^+_{\texttt{Borrowing}}(m, c, t, s) \vee \texttt{Borrowing}(m, c, s) \wedge \neg\gamma^-_{\texttt{Borrowing}}(m, c, t, s)$$

$$\gamma^+_{\texttt{Borrowing}}(m, c, t, s) \equiv t = \texttt{borrow}(m, c) \wedge SPFS(\texttt{borrow}(m, c), s)$$
$$\vee t = \texttt{get\_reserved\_copy}(m, c) \wedge SPFS(\texttt{get\_reserved\_copy}(m, c), s)$$

$$\gamma^-_{\texttt{Borrowing}}(m, c, t, s) \equiv t = \texttt{return}(m, c) \wedge SPFS(\texttt{return}(m, c), s)$$

In our case, the form of these formulas is especially simple, because our specification does not contain a transaction with effects depending on other conditions. Now to find a formula describing the requirements on actions and states to arrive via one step in a state where $\phi = \texttt{Borrowing}(\texttt{MEMBER}_1, \texttt{COPY}_{10})$ holds, we can apply a regression operator $\mathcal{R}$, that essentially substitutes the Ψ_P's for terms relating to the state dependent predicate after a transaction. In our case, this results in the formula expressing that it is only possible to arrive via one transaction in a state where ϕ holds, if that transaction was either a $\texttt{borrow}(\texttt{MEMBER}_1, \texttt{COPY}_{10})$ or a $\texttt{get\_reserved\_copy}(\texttt{MEMBER}_1, \texttt{COPY}_{10})$, and it was executable, or any transaction different from a $\texttt{return}(\texttt{MEMBER}_1, \texttt{COPY}_{10})$ if ϕ did happen to hold already before the transaction was executed.

When we are able to convert our specifications to this format, we avoid the first aspect frame problem by not having to specify frame axioms explicitly (they are implicit in the specification completeness assumption about the action effect specification). Its second aspect is avoided since we are still able to reason by means of goal regression. Compared with the STRIPS language, we have gained expressivity and a clear semantics without falling back to inefficient reasoning in the general situation calculus.

The generation of a plan of actions leading from state s_1 to s_2 is now reduced to two subproblems:

- Search for sequences of actions that could lead from s_1 to s_2, generating a condition for applicability of this plan in terms of properties of s_1 by means of regression.
- Testing whether the description of s_1 implies this condition. This is a standard task for an automatic theorem prover.

Conversion of our specification to Reiter's specification format is reasonable in our database context. We need to make the following assumptions:

- To be able to formulate action effects as successor state axioms, we need to have complete knowledge of the action effects. Although this is a problem in general, in a database modeling context this assumption is satisfied. This is because we need to bound and simplify our domain anyway to obtain a model. Moreover, the action effect specification has to be in a format that allows us to derive the successor state axioms from it automatically.
- We have complete knowledge of action preconditions (the qualification problem) in order to build sufficient conditions for success. For the same reason as above, this assumption is

satisfied. We can easily specify them by writing a specification that mentions only the necessary preconditions for success, and making the assumption that the specification mentioned all of them. Then the sufficient precondition for success of an action is the conjunction of all these necessary preconditions for its success.

- We have no derived updates due to constraints. This is a restriction that may have to be lifted later.
- We have deterministic actions only, as is usual in database updates.
- We have no disjunctive information in states. Again, this is a restriction that may have to be lifted later.

3.3 Construction of example states

Reiter's representation and regression discussed above allow us to reduce problem of planning to theorem proving. The same approach could be applied for solving reachability queries too, if we modify it to reflect the way it differs from planning in the way quantification over states is done, as discussed in section 3.1. This modification consists of replacing the theorem prover, used to test whether the initial state formulas imply the regressed goal formula, by a component that tests the satisfiability of the conjunction of the initial state formulas and the regressed goal formula. In the case of satisfiability, it should also construct a model that demonstrates this property. How can such a component be built?

Methods for testing the *satisfiability* of a set of formulas are scarcely found in the automated reasoning literature. Instead, most approaches aim at showing the *validity* of a formula, often by means of showing the inconsistency of its negation. Sometimes, we can get a proof of satisfiability as spin-off of a failed attempt to prove inconsistency. This is for example the case in the method of analytic tableaux[16], when a path in the proof tree is encountered that cannot be extended any further by the tableau extension rules, but doesn't contain a contradiction. The formulas on such a path constitute a Hintikka set, corresponding to a model for the formula. Caferra[3] tried to integrate the systematic search for models into the tableau method by means of techniques for solving equational problems. Similar proposals exist for resolution-based methods.

In their original form, these methods are not directly applicable to our problem, because of our additional (non-first-order) requirement on our intended model that all extensions of state dependent predicates in them must be finite. This is related to the problem of finding models with a finite number of elements of first-order formulas.

The method of analytical tableaux, nor its extension by Caferra are complete for finding finite models. This is caused by the liberal introduction of constants of each sort by the tableau extension rules for quantifiers. For example, the tableau extension rule for a formula of the form $\exists x \phi$ introduces a fresh constant and requires $\phi[a/x]$ to hold. The problem is that it does so even if another already introduced constant would do as well.

Kung[8] noted this problem when applying the tableaux method for proving the consistency of database specification, and proposed a variant of the tableaux method that restricts the introduction of new constants. Although this idea was good, his method is still not complete for finite satisfiability.

Bry and Manthey[2] analyzed the problem of semideciding finite satisfiability and unsatisfiability of integrity constraints. They describe how case analysis of models with different function evaluations can be added to the tableaux method to solve this problem.

Of related interest is the SATCHMO theorem proving system[10] by the same authors, that tries to prove theorems by failure of a systematic attempt to create concrete models of them, exploiting range-restrictedness in clauses. In its original form, SATCHMO is not complete for finite satisfiability, but Bry, Decker and Manthey[1] apply a variant of it to test constraint satisfiability in a database context, where constraints are formulas with only restricted quantification. Constraints with restricted quantification arise quite naturally in our framework, by our separation of the set of possible object identifiers from the set of actually existing ones in a state, by means of the

`Exists` predicate. Combined with the order-sorted approach of specifying a separate sort for each class and the fact that integrity constraints usually refer only to the existing objects only, range restrictedness is a reasonable assumption in our context. Although concrete database states are constructed in their paper, they seem to serve only as a temporary result of an approach to give a yes/no answer to finite satisfiability questions. Of course, in our intended application, we are very much interested in these models themselves.

3.4 Integration of techniques for solving reachability queries

Given the techniques discussed above, we are now in a position to outline a method for solving reachability queries:

1. Transform the specification into Reiter's format with successor state axioms and sufficient preconditions for success.
2. Use search and regression of the goal formulas of the reachability query, obtaining as a subproblem a test for finite (un)satisfiability of a set of formulas relating to a single starting state.
3. Apply the adapted version of the tableau method for testing finite satisfiability of a set of range-restricted constraints as described by Bry, Decker and Manthey.
4. When a starting state has been found that satisfies the integrity constraints and the regressed goal constraints, apply progression (computation of the successor states) on the generated action sequence to obtain the concrete intermediate and final states of the scenario.

4 Discussion and future work

We believe that the approach outlined in this paper is feasible and would yield a useful addition to animation techniques. In the introduction, we mentioned two problems with traditional approaches to animation. The first problem is that there is a mass of irrelevant details to be suppressed during the animation. By answering reachability questions by means of theorem-proving and planning techniques, the system has, at least in principle, the information available that allows it to relate the (non)existence of a path from one state to another to the axioms in the specification. It can therefore (again at least in principle) focus on the presentation of the relevant aspects of the animation.

The second problem mentioned in the introduction is that in order to find interesting animations, the analyst has to reason about the specification himself. Obviously, this problem would be avoided by our approach to animation.

Needless to say, there is still a considerable amount of research to be done before an animation system such as proposed in this paper is implemented. In order to further work out the ideas presented in this paper, we will build a small prototype based on a model-generation theorem prover like SATCHMO and work through a number of small examples with it.

References

[1] François Bry, Hendrik Decker, and Rainer 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.

[2] François Bry and Rainer Manthey. Checking consistency of database constraints: a logical basis. In *Proc. 12th International Conference on Very Large Data Bases*, pages 13–20, Kyoto, August 1986.

[3] Ricardo Caferra. A tableaux method for systematic simultaneous search for refutations and models using equational problems. *J. Logic Comput.*, 3(1):3–25, 1993.

[4] 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.

[5] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3 and 4), 1971.

[6] C. Green. Application of theorem proving to problem solving. In *Proc. 1st IJCAI*, pages 219–239, 1969.

[7] V. Karakostas and P. Loucopoulos. Verification of conceptual schemata based on hybrid object-oriented and logic paradigm. *Information and Software Technology*, 30(10):587–594, December 1988.

[8] C. Kung. A tableaux approach for consistency checking. In A. Sernadas, J. Bubenko Jr., and A. Olivé, editors, *Information Systems: Theoretical and Formal Aspects*, pages 191–207. Elsevier Science Publishers (North-Holland), 1985.

[9] V. Lifschitz. On the semantics of STRIPS. In M.P. Georgeff and A.L. Lansky, editors, *Reasoning about actions and plans*, pages 1–9. 1987.

[10] Rainer Manthey and François Bry. SATCHMO: A theorem prover in PROLOG. In Jörg H. Siekman, editor, *Proceedings CADE-8*, 1986.

[11] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[12] E.P.D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In R.J. Brachman, H. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332. Morgan Kaufmann Publishers, 1989.

[13] W. Reisig. *Petri Nets.* Springer, 1985.

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

[15] Lenhart Schubert. Monotonic solution of the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. In Henry E. Kyburg, Ronald P. Loui, and Greg N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning.* Kluwer, 1990.

[16] R. Smullyan. *First-order Logic.* Springer-Verlag, 1968.

[17] 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.

[18] 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).*