Simulation and Analysis of Business Processes Using GOLOG

Dimitris Plexousakis
Department of Computer Science
University of Toronto
Toronto, Ontario,
M5S 1A4, Canada
E-mail: dp@ai.toronto.edu

Abstract

This paper describes a novel approach to simulating and analyzing business processes using GOLOG, a high-level logic programming language suitable for defining complex behaviors and capable of simulating action execution. The language is based on an extended version of the situation calculus and incorporates a formal theory of action. Business processes can be viewed as actions (physical or perceptual) that affect the state of affairs or an agent's knowledge of this state. Using GOLOG, business processes can be specified, synthesized and tested for feasibility and consistency. The theoretical framework behind GOLOG includes a solution to the frame problem for perceptual and complex actions, as well as, a formal method for process analysis. The latter uses a solution to the ramification problem for proving the satisfaction or violation of constraints. In case this is not possible, the method proposes strengthenings to the processes' pre- and post-conditions, so that any implementation that meets the process specification, provably guarantees that constraints will not be violated. In this manner, business process reengineering can be assisted by a formal analysis and simulation tool for testing the consistency of the process model.

1 Introduction

The role of Artificial Intelligence (AI) in Business Process Reengineering (BPR) is two-fold: AI can provide both the enabling technology for automatically reengineering processes and tools to support process redesign by the user [Ham94]. The majority of attempts to put AI to work in BPR have insisted on the first of these roles. It remains a challenging issue to develop tools for evaluating and for assisting the production of designs. There's a growing requirement to represent processes through which work is achieved. Process representation thus becomes a vital issue in redesigning work and allocating responsibilities. Many languages aiming to assist in the requirements engineering activity of information systems development have been proposed (see for instance RML [Gre84], Albert [DDBDP94], KAOS [vLDM94] and Telos [MBJK90]); few of them though provide a solid theoretical basis for reasoning about processes. Despite the importance of the representation issue, we, in this paper, will focus on the problems of simulating and analyzing business processes and propose the use of a novel logic programming language as a tool that will support the analysis phase in process reengineering.

Unlike other areas of process modeling, e.g., software process modeling [HL88], business process modeling attempts to capture phenomena enacted by humans rather than machines. This creates additional requirements, specifically for modeling actions that only involve acquiring or communicating knowledge or information. Reasoning about properties of business processes, such as, e.g., invariance or consistency with respect to a set of constraints, deadlock or communication bottleneck detection [Ell80], requires that actions of the above sort be formalized in addition to actions whose effect it is to change the state of affairs in the domain modeled. GOLOG [LLL+94] provides a theoretical platform for reasoning about both physical and perceptual actions, i.e., actions whose effect it is to change an agent's knowledge of the state of affairs.

GOLOG was initially conceived as a language for high-level robot programming. Hence, in itself, GOLOG lacks the conceptual richness of languages such as Telos [MBJK90] or KAOS [vLDM94] that are intended to be used in requirements modeling and conceptual modeling in general. However, GOLOG can be coupled with a requirements modeling language and function within the simulation and analysis component of a business process management system (BPMS) [Kar94]. GOLOG can also support the process enactment phase, where activity plans can be scheduled. Then the viability of the synthesized plans can be tested.

For the purposes of this paper we will adopt a functional/behavioral representation of processes [CKO92]. We will focus on the sequencing of process elements or their iterations, as well as necessary and sufficient conditions for process execution. These will be exploited for proving properties of processes within a formal framework. The majority of process modeling languages lack mathematical formality or employ a very low level thereof. The higher the degree of formal precision a language possesses, the easier the enactment of processes on a machine is. Business processes involve a relatively high degree of non-determinism, hence a formal language aiming to model business processes should incorporate machinery that goes beyond traditional programming paradigms.

A process modeling language should permit the evaluation of the adequacy of a proposed process. The model should be analyzable for properties such as syntactic correctness and consistency with respect to constraints. In this paper, we argue that a formal method for analyzing the consistency of process specifications with respect to constraints can be devised by using a solution to the well-known in AI *frame* [McC69] and *ramification* [Fin88] problems. Given a set of action specifications, the problem of succinctly stating that "nothing else changes" except the aspects of the state explicitly specified, has been called the frame problem. The *ramification* problem amounts to devising a way to avoid having to specify indirect effects of actions explicitly. Several attempts to solve these problems have appeared in the AI planning literature of recent years. [LR92] and [Pin94] present a solution to the frame and ramification problems with an application to database updates. An extension of this solution has been used for proving transaction safety in temporal deductive databases in [PM95]. Perceptual actions are dealt with in [LLL+94]. [BMR93] depicts how the frame problem becomes particularly acute in object-oriented specifications where transactions are inherited and specialized from superclasses to subclasses. In the area of information systems development and in process modeling in particular, the frame and ramification problems have traditionally been either ignored or bypassed by means of explicit assumptions (see, e.g., [DDBDP94], [HL88]). In [BMR93], a systematic solution to the frame and ramification problems for determinate transaction specifications is presented. We will employ this solution for strengthening process specifications - at design time - to guarantee the maintenance of constraints by any implementation that meets the specifications. Moreover, as will be shown in the sequel, a solution to the ramification problem can be used for proving whether constraints are preserved or violated by action execution.

We now introduce an example that is used throughout the paper for demonstrating the representation and simulation of actions and the process analysis method. We limit ourselves to presenting action specifications in an abstract language. We assume that actions are specified in terms of precondition/postcondition pairs, where both of these conditions are specified in a variant of first-order predicate calculus. The idea for the example below is borrowed from [DvLF93] although the specifications presented here are different. In this section we only give an initial specification of representative actions and objects. Primed predicates or function terms will denote the respective (truth) values in the state resulting from the action execution. The example will be refined as needed in the sections to follow.

Example 1.1 Consider a library management domain that includes as entities libraries, books and borrowers. Primitive actions include checking-out books, issuing reminders to borrowers, charging fines for late returns and so forth. A goal to be achieved in the library management system will be to satisfy book requests. Figure 1 depicts the specifications of the actions *Check-Out* and *Issue-Reminder*. We assume that an object of type *Library* has attributes *available* and *checkedOut* that denote the sets of books that are available and have been checked out respectively, and the set *requested* denoting the books for which there currently exists a loan request. The following predicates occur in action specifications: *borrows(br,bk,dt)*, meaning that borrower *br* borrows book *bk* on date *dt*, *requests(br,bk,dt)*, meaning that prospective borrower *br* requests book *bk* on date *dt*, *returns(br,bk,dt)*, meaning that borrower *br* returns book *bk* on date *dt*, and *reminderIssued(br,bk,dt)*, meaning that a reminder has been issued to borrower *br* on date *dt* concerning the loan of book *bk*. For simplicity, we assume a very coarse time line where the basic unit is that of a day. The global function $today()$ returns the current date.

The precondition of the action *Check-Out* states that the action is possible only if the book is available on that particular day and the prosepctive borrower has issued a request for it. The action postcondition expresses the immediate effects of the action on the predicates and functions occurring in the domain specification. Specifically, in the stater resulting after the action's execution, the book is neither available nor requested, but belongs to the list of checked-out books. The cardinalities of the respective sets are updated and the predicate borrows becomes true of the borrower *br*, the book *bk* and the date *dt*. The second action, *Issue-Reminder*, is possible only if a book has been borrowed for more than two weeks and no reminder has been issued in the meantime. Its postcondition makes the predicate *reminderIssued* true of the action's parameters. Given such specifications, we would like to be able to infer whether action execution, either in isolation or as part of processes involving other actions, may violate constraints expressing library policies. □

The rest of this paper is organized as follows. Section 2 gives a short introduction to GOLOG and the situation calculus. The solution to the frame and ramification problems that is employed by the process analysis methods we propose is also sketched. Section 3 deals with the representation of business processes in GOLOG and the formulation of system goals by means of procedures in the language. Section 4 presents our process analysis method which relies on the solution to the frame and ramification problems. Finally, section 5 concludes the paper with an outlook for further research.

Action *Check-Out (br, bk, dt)*

Precondition: $bk \in available[dt] \cap requested[dt]$

Postcondition:

$\neg(bk \in available[dt]' \cup requested[dt]') \ \wedge \ (bk \in checkedOut[dt]') \ \wedge$
$borrows'(br, bk, dt) \ \wedge \ |available[dt]'| = |available[dt]| - 1 \ \wedge$
$|requested[dt]'| = |requested[dt]| - 1 \ \wedge \ |checkedOut[dt]'| = |checkedOut[dt]| + 1$

end

Action *Issue-Reminder (br,bk,dt)*

Precondition:

$\exists d \ (borrows(br, bk, d) \ \wedge \ (today() - d > 2w) \ \wedge$
$\neg\exists d'((d < d' < today()) \ \wedge \ reminderIssued(br, bk, d')))$

Postcondition: $reminderIssued'(br, bk, dt)$

end

Figure 1: Primitive action specifications in the library management system

2 Situation Calculus and GOLOG

GOLOG [LLL+94] is a novel programming language that is suitable for defining and executing complex actions. The language was initially conceived for high-level robot control and is the result of extending the *situation calculus* with perceptual and complex actions, including non-determinate ones. The development of GOLOG is part of an ongoing project in cognitive robotics that aims at the integration of reasoning, perception and action within a uniform framework that possesses a strong theoretical basis. The language has a situation calculus semantics and an interpreter that executes actions in a real or simulated environment.

Before presenting the complex action expressions that make up GOLOG, a few words should be said about the situation calculus. The situation calculus is a first-order language for representing dynamically evolving domains. Changes are brought to being in states of the world, *situations*, as results of actions performed by an agent. A situation calculus structure thus contains a set $\mathcal{A}$ of actions and a set S of situations. For an action $\alpha \in \mathcal{A}$ and a situation $s \in S$, the term $do(\alpha, s)$ denotes the situation that results from the execution of action α in situation s. Relations whose truth values may differ from one situation to another are called *fluents*. They are denoted by predicate symbols having a situation term as their last argument. Similarly, the term *functional fluent* is used to denote functions whose denotation varies from one situation to another.

All actions in the situation calculus are assumed to be primitive and determinate. GOLOG permits the definition of complex actions through sequencing, iteration, non-deterministic choice of actions and non-deterministic choice of action parameters. Complex actions, or simply actions henceforth, are defined as follows: Simple actions are actions. If α_1, α_2 are actions, then $[\alpha_1; \alpha_2]$ is the action that consists of α_1 followed by α_2, $[\alpha_1|\alpha_2]$ is the action consisting of non-deterministically choosing between α_1 and α_2, $\Pi_x(\alpha_1)$ denotes the non-deterministic choice of parameter x for α_1. Other actions include, for a situation calculus formula ϕ, tests $(?\phi)$, conditionals $(if\ \phi\ then\ \alpha_1\ else\ \alpha_2)$ and iteration $(while\ \phi\ do\ \alpha_1)$. Complex actions are treated as macros for situation calculus expressions. A predicate $Do(\alpha, s, s')$, where α is an action and $s, s' \in S$, is taken to mean that the agent's executing action α in situation s, leads to (a not necessarily unique) situation s'.

GOLOG possesses the property that complex actions defined therein decompose into primitive actions. Moreover, the language interpreter is in essence a theorem prover that performs arbitrary first-order reasoning. This is required for executing actions that include tests, conditionals or iterations.

In GOLOG, one can express goals to be achieved by schematic plans. The details about how actions are to be performed are inferred by the theorem prover during the course of evaluating action conditions. In using GOLOG, the user provides a specification of primitive actions in terms of their preconditions and their effects on the world modeled. As will be shown in the sequel, such a specification along with a completeness assumption, suffice for performing process analysis for determining consistency of specifications with respect to constraints and for strengthening of specifications so that the process execution will not violate any of the constraints. Procedures comprising complex actions can then be written and executed by the system interpreter.

Because of the presence of arbitrary situation calculus formulae as parts of tests or while-loops in procedures, GOLOG maintains the world model by keeping track of the effects of actions and by modifying the model to reflect the knowledge acquired through perceptual actions. To reason about actions and their effects, a form of *regression* [Rei91] can be used to reduce conditions on arbitrary states to ones involving only the initial state. Alternatively, the *progression* of the knowledge base describing the world model is proposed in [LR92] as a less costly - in certain cases - method of bringing the model up to date.

2.1 The Frame and Ramification Problems

Before we proceed into the simulation and analysis of business processes with GOLOG, we briefly present the core of the theoretical framework behind GOLOG. This consists of a solution to the frame and ramification problems for complex and perceptual actions. We only sketch the solution and apply it to our working example. More details can be found in [Rei91], [LR92], [Pin94] and [PM95].

A *ramification* of a goal is a condition that is inevitably true if the goal is true [Fin88]. This definition is amenable to different interpretations in different world models. If the world model in question is expressed as a first-order theory, then the concept of ramifications can be captured by first-order entailment. It is easy to verify the following properties of ramifications: (1) If a ramification of a goal is known to be unsatisfiable, then the goal is unsatisfiable, (2) Goal ramifications can reduce the search space for goal satisfaction, and (3) Transformations may be applicable to goal ramifications but not to the goals themselves. Hence, if there exists a way to systematically generate ramifications from a set of goals, then the derived ramifications can be used for making the task of meeting the goals simpler.

Generating ramifications may require an arbitrary amount of inferencing. From the semi-decidability of first-order entailment, it follows that the problem of finding ramifications is, in its generality, intractable. Tractability can be achieved by restricting the class of goals for which ramifications are sought. For instance, the task is tractable for the case of ordered conjunctive goals [Fin88]. Furthermore, not all derivable ramifications may be useful for simplifying the task of proving a goal. For that, the generator may be guided to derive only "useful" ramifications by providing appropriate input clauses. In fact, the solution to the ramification problem that we sketch here generates exactly those ramifications that are needed for proving the specifications won't violate any constraints. The systematic solution presented here was initially proposed in [LR92] and [Pin94], and was extended in [PM95].

The method generates *successor-state axioms* from a given set of *effect axioms* in the presence of a limited class of state constraints. Effect axioms specify the direct effects of actions on predicates. Successor state axioms characterize all conditions under which fluents may change value as a result of action execution. The intent of the generation method is to "compile" the constraints into the successor-state axioms. The generation process is described in the sequel.

We use a many-sorted first-order language in which we distinguish one sort for situations; the rest of the sorts are object sorts. Constraints are of the form:

$$\forall x_1/S_1, \ldots, x_k/S_k, \forall t_1, t_2/S\ \phi(x_1, \ldots, x_k, t_1, t_2) \vee (\neg)p_1(x_1, \ldots, x_k, t_1) \vee (\neg)p_2(x_1, \ldots, x_k, t_2)$$

where, p_1, p_2 are $(k+1)$-ary predicates, $S_1, \ldots, S_k$ are object sorts and $\phi(x_1, \ldots, x_k, t_1, t_2)$ is a formula in which variables $x_1, \ldots, x_k, t_1, t_2$ occur free, if at all, and does not mention any predicate other than evaluable predicates. This class includes static constraints and transition constraints, but not general dynamic constraints. For example, the transition constraint specifying the property that an employee's salary can never decrease, can be specified by the formula

$$\forall e/Employee\ \forall s_1, s_2/Salary\ \forall t_1, t_2/S\ (s_1 < s_2) \vee \neg(t_1 < t_2) \vee \neg salary(e, s_1, t_1) \vee \neg salary(e, s_2, t_2)$$

An action α with parameters $\overline{x}$ is specified by a pair $(pre_\alpha(\overline{x}), post_\alpha(\overline{x}))$, where $pre_\alpha(\overline{x})$ and $post_\alpha(\overline{x})$) denote the action pre- and post-condition respectively.

The generator assumes the existence of causal rules describing the direct effects of transactions. These rules, called the direct effect axioms, have the form $\forall \overline{x}(Occur(\alpha(\overline{x}), s) \Rightarrow pre_\alpha(\overline{x}, s) \wedge post_\alpha(\overline{x}, do(\alpha, s)))$. Given any action specification, the effect axioms are derived independently of any other action specification. Hence, we can avoid having to specify the axioms in a reified logic that interprets the predicate *Occur* outside a standard first-order interpretation. The above axiom can now be written as: $\forall \overline{x}(\alpha(\overline{x}, s) \Rightarrow pre_\alpha(\overline{x}, s) \wedge post_\alpha(\overline{x}, do(\alpha, s)))$. For instance, from the specification of the action *Check-out* in figure 1, we can derive the following direct effect axiom[1]:

$$\begin{aligned}
&\forall br/Borrower\ \forall bk/Book\ \forall dt/Date\ (Check-Out(br, bk, dt) \\
&\Rightarrow\ (bk \in available[dt] \cap requested[dt])\ \wedge \\
&\quad (\neg(bk \in available[dt]' \cup requested[dt]')\ \wedge \\
&\quad (bk \in checkedOut[dt]')\ \wedge borrows'(br, bk, dt)\ \wedge \\
&\quad |available[dt]'| = |available[dt]| - 1\ \wedge \\
&\quad |requested[dt]'| = |requested[dt]| - 1\ \wedge \\
&\quad |checkedOut[dt]'| = |checkedOut[dt]| + 1))
\end{aligned} \tag{1}$$

From the direct effect axioms we can systematically generate *positive* and *negative* effect axioms [BMR93] for every predicate P that occurs in $post_\alpha$. A positive (negative) effect axiom for a predicate P expresses necessary conditions for the change of the truth value of P from $True$ ($False$) to $False$ ($True$ respectively) during the transition from a situation s to the situation $do(\alpha, s)$ resulting from the execution of action α. The systematic generation of positive and negative effect axioms is given in appendix A. Here we only show the effect axioms derived for the predicate *borrows*, under the assumption that the only actions available are the actions *Check-Out* and *Issue-Reminder*:

$$\begin{aligned}
&\forall br/Borrower\ \forall bk/Book\ \forall dt/Date \\
&\quad \neg borrows(br, bk, dt)\ \wedge\ borrows'(br, bk, dt)\ \wedge \\
&\quad Check-Out(br, bk, dt)\ \Rightarrow\ (\neg(bk \in available[dt]' \cup \\
&\quad requested[dt]')\ \wedge (bk \in checkedOut[dt]')\ \wedge \\
&\quad |available[dt]'| = |available[dt]| - 1\ \wedge \\
&\quad |requested[dt]'| = |requested[dt]| - 1\ \wedge \\
&\quad |checkedOut[dt]'| = |checkedOut[dt]| + 1))
\end{aligned} \tag{2}$$

$$\begin{aligned}
&\forall br/Borrower\ \forall bk/Book\ \forall dt/Date \\
&\quad borrows(br, bk, dt)\ \wedge\ \neg borrows'(br, bk, dt)\ \wedge \\
&\quad Issue-Reminder(br, bk, dt)\ \Rightarrow\ False
\end{aligned} \tag{3}$$

[1] We use the primed/unprimed notation here to avoid the quantification over situations.

The consequent of the negative effect axiom is $False$ since there is no action specification with the effect of falsifying the fluent $borrows$.

In addition to the effect axioms the knowledge base is augmented with unique name axioms for actions, predicates and functions.

The next step is the generation of successor-state axioms (see appendix A for a formal presentation). As mentioned earlier, successor-state axioms characterize all conditions under which fluents may change truth value. Thus, the generation of a set of successor-state axioms for the fluents of a particular domain constitutes a solution to the frame problem. To account for constraint ramifications, the fluents occurring in the expression of constraints are considered as well.

For example, assume that the following constraint is defined:

$$\forall br/Borrower\ \forall bk/Book\ \forall dt/Date$$
$$\neg borrows(br, bk, dt) \vee \neg returns(br, bk, dt)$$

It expresses the property that, on a certain date, a person may be only borrowing or returning a book. For brevity in presentation, let us rewrite the positive effect axiom (2) for *borrows* as

$$\forall br/Borrower\ \forall bk/Book\ \forall dt/Date$$
$$\neg borrows(br, bk, dt)\ \wedge\ borrows'(br, bk, dt)\ \Rightarrow\ \phi \qquad (2')$$

where ϕ is the formula

$$Check - Out(br, bk, dt)\ \Rightarrow\ (\neg(bk \in available[dt]' \cup requested[dt]')\ \wedge (bk \in checkedOut[dt]')\ \wedge |available[dt]'| = |available[dt]| - 1\ \wedge |requested[dt]'| = |requested[dt]| - 1\ \wedge |checkedOut[dt]'| = |checkedOut[dt]| + 1))$$

The syntactic generator will generate the following successor-state axiom for the predicate *borrows*:

$$\forall br/Borrower\ \forall bk/Book\ \forall dt/Date$$
$$\neg borrows(br, bk, dt)\ \wedge\ \neg\phi\ \wedge\ returns(br, bk, dt)\ \Rightarrow \neg borrows'(br, bk, dt)$$

or, equivalently,

$$\forall br/Borrower\ \forall bk/Book\ \forall dt/Date$$
$$\neg borrows(br, bk, dt)\ \wedge\ borrows'(br, bk, dt)\ \Rightarrow \phi\ \vee\ \neg returns(br, bk, dt) \qquad (4)$$

Axiom (4) captures the ramification $\neg\phi \wedge returns(br, bk, dt)$ of (a form of) the constraint and the postcondition of the action *Check-Out* as a necessary condition for the change in the truth value of the predicate *borrows*.

Finally, in appendix A we show how knowledge producing actions are taken into account [LLL+94]. The succesor state axioms in this case only state that a particular predicate or function term becomes known as a result of a knowledge producing action taking place.

The correctness of the syntactic generation has been proven in [LR92],[BMR93], [Pin94] and [PM95].

3 Process Simulation

In this section we present the simulation of complex business processes in GOLOG. The language permits the representation of goals to be achieved as programs consisting of actions that achieve subgoals of the original goal. The use of GOLOG language constructs is illustrated through process specification in the library management example.

First, a specification of the application domain is needed. This consists of the definition of primitive actions and the derivation of the action precondition and effect axioms. Both types of axioms can be derived from action specifications of the type used in the examples of section 1.

We consider as primitive the actions *Check-Out* and *Issue-Reminder* of example 1.1, as well as the action *Request* which is specified in figure 2. We assume that the action *Request* is always possible. Moreover, we define an action *SenseRequests* which receives all book requests issued by prospective borrowers. Unlike the rest of the actions, *SenseRequests* is a knowledge-producing action and is considered to be possible at all times. Its effect is to update the set of books that are currently on request with those that someone has requested and that have not been check-out as of yet. Figure 2 depicts the specification of the new actions. From this specification of actions, the effect axioms can be easily derived as described in section 2.1. For example, assuming that the fluent *requests* only occurs positively in the postcondition of action *Request*, the (positive) effect axiom for *Request* will have the form:
$\forall\alpha, br, bk, dt\quad \neg requests(br, bk, dt, s)\ \wedge\ (\alpha = Request(br, bk, dt, s)) \Rightarrow requests(br, bk, dt, do(\alpha, s))$.

Now we are in a position to write procedures in GOLOG that implement goals in the library management system. For instance, the goal *SatisfyRequests* expresses the requirement that the system should serve all requests for book loans while such requests exist. The GOLOG procedure *SatisfyRequests* first "senses" all requests (if such exist) for book loans on a particular day. While there exists a request to be served, the system non-deterministically chooses some request and checks if it can be satisfied. The procedure is shown in figure 3.

$Check - Status$ is the name of another GOLOG procedure that is called by $SatisfyRequests$. The specification of the procedure is shown in figure 4. The procedure checks if an unavailable book is overdue. If this is the case, then it issues a reminder to its borrower.

In a similar fashion, we can write procedures in GOLOG to achieve all goals that are set forth in the system's requirements specification. As shown in procedure $SatisfyRequests$, non-deterministic and knowledge producing actions can be part of procedure specifications. These

Action *Request (br,bk,dt)*
  Precondition: $True$
  Postcondition: $requests(br, bk, dt)$
end

Action *SenseRequests(dt)*
  Precondition: $True$
  Postcondition: $requested[dt]' := requested[dt] \cup$
    $\{bk | \exists bor \; requests'(bor, bk, dt) \wedge \neg borrows'(bor, bk, dt)\}$
end

Figure 2: Specification of the library management system (cont'd)

```
procedure SatisfyRequests(dt)
        SenseRequests(dt);
        while (requested[dt] ≠ ∅) do
                    Π br, bk [if (requests(br, bk, dt) ∧ (bk ∈ available[dt]))
                                then Check − Out(br, bk, dt)
                                else Check − Status(bk, dt);]
        endwhile
end
```

Figure 3: Process specification in GOLOG

specifications can be also executed by a GOLOG interpreter. What is needed apart from the domain axiomatization and the procedure specification, is the specification of the initial state. The specification of the initial state need not be complete. This gives the ability to model incompletely known domain as well. The domain modeled is maintained by the interpreter by regression so that testing of conditions on arbitrary domain state, amount to evaluating conditions on the initial state only.

Albeit short and simple, the above example aims at depicting the advantages of the specification of complex business processes in a high-level programming language. Firstly, perceptual actions, that are highly likely to be part of a business process, can be included in the procedures. In addition, a formal account of such actions exists. Secondly, non-deterministic actions can be specified. Thirdly, processes can be written in a modular manner and be executed via an interpreter. These procedures specify how goals are to be achieved without necessarily specifying all the details about how the goal is to be achieved. It's the responsibility of the theorem-prover to fill all the details about action execution.

The interpreter of GOLOG accepts definitions of primitive and complex actions in Prolog. The specifier must also supply the precondition and effect for primitive events. The successor-state axioms can then be compiled automatically for the effect axioms. Figure 5 contains part of an initial specification of the library management example in GOLOG. The simulation is rather simplistic at this stage, as compared to other simulation tools (e.g., [DDBD94]), which do not perform theorem proving or plan generation. The interpreter returns a complete trace of the execution of processes, showing the actions that took place since the initial situation. As a side-effect of the evaluation of conditions in tests or while-loops, a complete plan that achieves the goals that the procedures implement is returned. The declarative approach to process specification permits the simulation of processes even when only partial specifications are given.

4 Process Analysis

In this section we present a formal technique intended to assist in the analysis phase of business process reengineering. The method is used to determine whether consistency with respect to defined constraints is preserved or violated as a result of process execution. In cases where such proof or disproof is not possible at process specification time, strengthenings to specifications of actions that are relevant to the constraints are proposed so that, any implementation meeting the strengthened specifications provably guarantees that constraints will not be violated in the state resulting from action execution.

We argue that providing such a functionality to the process analysis component of a BPMS, is essential for the process reengineering effort. The process specifier realizes the implications of actions and the implementor is saved the burden of finding ways to meet the postcondition and maintain the invariant. Consistency is guaranteed by the correctness of an analysis tool such as the one presented here. Moreover, optimized forms of conditions to be verified can be incorporated into process specifications.

Constraints serve the role of invariants of actions. No action may violate the relevant constraints in order to be accepted. The problem of proving that action execution does not violate its invariants is formalized in the following definition.

```
procedure Check − Status(bk, dt)
        if (bk ∈ checkedOut[dt])
        then [(∃br, d (borrows(br, bk, d) ∧ (dt − d > 2w))?;
                Issue − Reminder(br, bk, dt)]
end
```

Figure 4: Process specification in GOLOG (cont'd)

```
:- op(950,xfy, [:]).  /* Event sequence.  */
:- op(950, xfy, [#]).  /* Nondeterministic event choice.  */

/* Primitive events */
primitive_event(checkOut(Br,Bk)).
primitive_event(issueReminder(Br,Bk)).
primitive_event(request(Br,Bk)).
primitive_event(senseRequests).

/* Process SatisfyRequests */
proc(satisfyRequests, senseRequests :
      while(nonempty(requests),
            pi([Br,Bk], if (and(requests(Br,Bk),available(Bk)),
                              checkOut(Br,Bk),checkStatus(Bk))))).

/* Preconditions for Primitive Events */
poss(checkOut(Br,Bk),S):- available(Bk,S), requested(Bk,S).
poss(issueReminder(Br,Bk),S):- borrows(Br,Bk,S1), today-S1>2,
                        not(reminderIssued(Br,Bk,S2), S1<S2, S2<today).
poss(request(Br,Bk),S).
poss(senseRequests,S).

/* Successor state axioms for primitive events.  */
requests(Br,Bk,do(A,S)):- A=request(Br,Bk) ; not A=request(Br,Bk),
                        requests(Br,Bk,S).

reminderIssued(Br,Bk,do(A,S)):- A=issueReminder(Br,Bk);
                        not A=issueReminder(Br,Bk), reminderIssued(Br,Bk,S).
......
```

Figure 5: Event and axiom specifications for the library management example

Definition 4.1 (*Invariant Maintenance*) Let A be an action specification with precondition P, postcondition Q and invariant I. A is said to maintain invariant I, if $I \wedge P \Rightarrow (Q \Rightarrow I')$.

Proving that actions maintain invariants is a difficult task since it requires theorem proving. A way to avoid checking whether actions maintain their invariants is to augment their specifications in a way such that the invariant is maintained as a result of meeting the new specification. We consider the single-action case first. Assume an action specification A includes a pair (P, Q) of a precondition and a postcondition expressed in first-order predicate calculus. Let I be an integrity constraint relevant to the action[2]. We need to find a formula N such that $(Q \wedge N \Rightarrow I')$, or equivalently $(Q \wedge \neg I' \Rightarrow \neg N)$ is logically implied by the knowledge base and the axiomatization of the domain. If $\neg N$ is a ramification of $Q \wedge \neg I'$, then the desired entailment relationship holds. This leads to the following theorem whose proof is an easy consequence of the definition of ramifications and invariant maintenance[3]:

Theorem 4.1 Let A be an action specification including a precondition/postcondition pair (P, Q) and let I be an invariant of A. If R is a ramification of $Q \wedge \neg I'$, then the invariant is maintained in the state resulting after the action's execution if the postcondition $Q \wedge \neg R$ is met.

Although the transformation of postconditions is unidirectional, it is sufficient for proving constraint maintenance. It is assumed that actions are always executed in states that respect the constraints. Satisfaction of a constraint's ramification does not, in general, imply the constraint's satisfaction; if a ramification of a constraint is violated, the constraint itself is violated. Moreover, the ramifications of constraints can be simpler formulae than the constraints themselves.

Certain ramifications can suggest that invariants do not have to be checked and postconditions need not be modified in order to guarantee the invariants. The case in which the propositional constant $False$ is derived as a ramification, is of particular interest since, as the following corollary specifies, no change in the postcondition is needed in order to meet the invariant.

Corollary 4.1 If $False$ is a ramification of $Q \wedge \neg I'$, then I' is maintained by any action execution meeting Q.

Hence, the process of generating ramifications can also discover inconsistent specifications of actions that may have escaped the specifier's attention. The following example will help illustrate the application of the ramification method.

Example 4.1 Assume that the following constraint has been defined in the library management system for the purpose of maximizing the number of book requests the library can serve:

$$I \equiv (\forall br \ (|\{bk | \exists dt \ borrows(br, bk, dt)\}| \leq Max(br))$$

where Max is a function on the class of borrowers. The constraint expresses the property that there exists an upper limit on the number of books a borrower can borrow from the library. Clearly, the constraint can be violated as a resulting of executing the action *Check-Out*. Violation of the constraint can be prevented if the action postcondition is strengthened by conjoining it with a ramification of the constraint. Let $\#borrowed(br)$ denote the cardinality of the set $\{bk | \exists dt \ borrows(br, bk, dt)\}$ for every borrower br. Then $I' \equiv \#borrowed'(br) \leq Max(br)$ and thus, $\neg I' \equiv \#borrowed'(br) > Max(br)$. Then we form the conjunction of the negated invariant and the action postcondition:

$$\begin{aligned}
\neg I' \wedge Post \equiv & \#borrowed'(br) > Max(br) \ \wedge \\
& \neg(bk \in available[dt]' \cup requested[dt]') \ \wedge \\
& (bk \in checkedOut[dt]') \ \wedge \ borrows'(br, bk, dt) \ \wedge \\
& |available[dt]'| = |available[dt]| - 1 \ \wedge \\
& |requested[dt]'| = |requested[dt]| - 1 \ \wedge \\
& |checkedOut[dt]'| = |checkedOut[dt]| + 1.
\end{aligned}$$

Exploiting the fact that after an action's execution it's direct effects, as these are specified by the postcondition, become true, we can derive $\#borrowed'(br) > Max(br)$ as a consequence of the above conjunction. Given that $\#borrowed'(br) = \#borrowed(br) + 1$, we derive $\#borrowed(br) \leq Max(br) - 1$ as the condition that suffices to be added to the postcondition of action *Check-Out* in order not to violate the constraint. □

The above example intends to convey the idea behind using the ramification method for strengthening action specifications. The process of deriving ramifications can be automated as described in section 2.1. The ramifications required to augment the postconditions are part of the successor-state axioms derived. Appendix B shows the application of the method to another example.

We would like to be able to propose similar augmentations to postconditions when the invariant refers to any number of states, both before and after an action takes place. In other words, we need to extend the method to take into account dynamic constraints. The solution to the ramification problem, as formulated in [Pin94], does not deal with constraints more general than transitional ones. We have extended the syntactic generator of ramifications to a more general class of constraints in [PM95], where some initial results were given through examples. The extension of the method to general dynamic constraints is a topic of current research.

The above results concerned the single-action single-constraint case. In the remainder of this section, we show how the method applies to the case of multiple

[2] The notion of "relevance" is defined formally in the sequel.

[3] [PM95] contains similar results for the case of database transactions.

actions, multiple constraints, conjunction and inheritance of specifications.

4.1 Multiple Actions and Multiple Constraints

In this case, all action specifications have to be taken into account since a constraint may be relevant to, or affected by, more than one actions. The notion of relevance is formally defined here. For that we will assume that the specifications are given in first-order predicate calculus (ordinary or temporal). Moreover, we will assume that no interleaving of actions is allowed. An action is regarded as the only means of state change.

Definition 4.2 (*Relevance)* A constraint I is relevant to an action specification A that includes a precondition/postcondition pair $(pre, post)$, if $post$ contains a literal that occurs in I.

A constraint relevant to a set of action specifications A_i with respective pre/post-conditions $(pre_i, post_i)$ has to be considered for the modification of each $post_i$, so that the execution of any action provably maintains the constraint. Hence, it suffices to repeat the process presented before for every A_i. The process may be optimized by reusing the derivation of ramifications for actions whose postconditions involve common predicates.

The symmetric case, where an action specification is associated with more than one invariants, is dealt with by simply taking the conjunction of the invariants as the new invariant. Then the derived ramification depends on all invariants, provided that the union of the invariants is a satisfiable set.

4.2 Conjoining Action Specifications

The conjunction of specifications α_1, α_2- denoted by $\alpha_1 || \alpha_2$ - is formed by conjoining the respective pre/post-conditions. Conjunction of action specifications corresponds to the sequencing $\alpha_1; \alpha_2$ of α_1 and α_2. Then, as theorem 4.2 suggests, it suffices to conjoin the ramifications of the two invariant-postcondition pairs, to guarantee that the invariant will be maintained if the new postcondition is met. The theorem provides a sufficient condition for the maintenance of constraints by sequences of actions.

Theorem 4.2 Let A_1, A_2 be action specifications with pre/post-conditions $(pre_1, post_1)$ and $A_2 = (pre_2, post_2)$ respectively. Let I be a constraint relevant to both A_1 and A_2. If there exist ramifications N_1 and N_2 which, if conjoined with the postconditions $post_1$ and $post_2$ guarantee the maintenance of I in A_1 and A_2 respectively, then $N_1 \wedge N_2$ is a ramification which, if conjoined with $post_1 \wedge post_2$ guarantees the maintenance of I in $A = A_1 || A_2$.

An important consequence of theorem 4.2 is the ability to accommodate new invariants without having to redo the entire process. Specifically, if a new invariant is to be added and is relevant to an action specification whose postcondition has already been augmented by computed ramifications, it suffices to verify that the new invariant does not introduce any contradiction, and, if this is the case, to generate ramifications of the new invariant and the postcondition. The new ramification can be conjoined with the previously derived ones, so that the new postcondition guarantees the invariants.

4.3 Inheritance of Action Specifications

Refinement of business processes can be accomplished via specialization in object-oriented specification languages. The inheritance of action specifications implies that the pre/post-conditions of the specialized actions are conditions stronger than those of the more general action, and, hence, include the pre/post-conditions of the more general action as conjuncts in their expression.

Assume action specification A_2 with pre/post-conditions $(pre_2, post_2)$ is a specialization of action specification A_1 with pre/post-conditions $(pre_1, post_1)$ and that there exists a ramification N_1 with the desired property of maintaining the action's invariants I if used to augment the postcondition $post_1$. The specification A_1 is inherited by A_2. If neither of the conjunctions $(pre_1 \wedge pre_2)$ and $(post_1 \wedge post_2)$ is a contradiction, then, according to theorem 4.2, if a ramification N_2 can be found, then augmenting $post_2$ with $N_1 \wedge N_2$ suffices to guarantee that the invariant will be maintained if the strengthened postcondition is met.

5 Conclusions

We have presented an adaptation of ideas from AI for the development of tools intended to assist in the reengineering of business processes. Specifically, we elaborated on the use of a high-level logic programming language, GOLOG, for the specification and simulation of business processes. The language's features for specifying sequences or iterations of actions, non-deterministic and perceptual actions make it suitable for specifying business processes. Moreover, GOLOG can be used for the simulation of processes and for proving schematic plans of actions correct. Most importantly, the theoretical basis of GOLOG which includes a solution to the frame and ramification problems, permits us to develop formal techniques for process analysis. We proposed techniques for verifying the consistency of specifications with respect to the defined constraints and for strengthening action specifications so that constraints are provably guaranteed by any implementation that meets the new action specifications. We also demonstrated the ability to incrementally accommodate new invariants or action specifications and the conjunction and inheritance of process specifications. We argue that providing tools of this kind is essential for the development of business process management systems.

Many issues need to be looked further into. In particular, we intend to extend the current results for proving properties of processes synthesized by any of the constructs provided by GOLOG. A concurrent version of GOLOG is also under development. A solution to the frame and ramification problems for the case of concurrent actions will provide the basis for extending the results presented in this paper for the case of concurrent processes. The time-sensitive nature of requirements that arise in business processes require that optimizations of the sort proposed here, should be extended to deal with arbitrary dynamic and real-time constraints. Last, but not least, we are looking into interfacing GOLOG with a more abstract language for process definition so that the precondition and successor-state axioms can be derived automatically from the specification of processes.

6 Acknowledgements

I wish to thank Yves Lesperance for his valuable insight on GOLOG and for his comments on an earlier draft of this paper, as well as Eric Yu for providng pointers to related literature, and Thodoros Topaloglou for comments that helped improve the presentation.

References

[BMR93] A. Borgida, J. Mylopoulos, and R. Reiter. And nothing else changes: The Frame Problem in Procedure Specifications. In *Proceedings of the 15th Int. Conference on Software Engineering*, 1993.

[CKO92] B. Curtis, M. Kellner, and J. Over. Process Modelling. *Communications of the ACM*, 35(9):75–90, 1992.

[DDBD94] E. Dubois, P. Du Bois, and F. Dubru. Animating Formal Requirements Specifications of Cooperative Information Systems. In *Proceedings of the 2nd International Conference on Cooperative Information Systems*, pages 101–112, 1994.

[DDBDP94] E. Dubois, P. Du Bois, F. Dubru, and M. Petit. Agent-Oriented Requirements Engineering - a Case Study Using the Albert Language. In *Proceedings of the 4th International Working Conference on Dynamic Modelling and Information Systems*, 1994.

[DvLF93] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-Directed Requirements Acquisition. *Science of Computer Programming*, 20(1):3–50, 1993.

[Ell80] C. Ellis. Office Streamlining. In N. Naffah, editor, *Integrated Office Systems*, pages 111–125. North Holland, 1980.

[Fin88] J. Finger. Exploiting Constraints in Design Synthesis. Technical Report STAN-CS-88-1204, Stanford University, 1988.

[Gre84] Sol Greenspan. *Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition.* PhD thesis, Department of Computer Science, University of Toronto, 1984.

[Ham94] Walter Hamscher. AI in Business Process Reengineering. *AI Magazine*, 15(4), 1994. Report on the AAAI Workshop.

[HL88] K. Huff and V. Lesser. A Plan-based Intelligent Assistant that Supports the Software Development Process. In *Proceedings of SIGSOFT-88*, pages 97–106, 1988.

[Kar94] Dimitris Karagiannis. Towards Business Process Management Systems, 1994. Tutorial at the 2nd Int. Conference on Cooperative Information Systems.

[LLL+94] Y. Lesperance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl. A Logical Approach to High-Level Robot Programming - A Progress Report. In *Proceedings of the AAAI Fall Symposium on Control of the Physical World by Intelligent Systems*, New Orleans, LA, 1994.

[LR92] F. Lin and R. Reiter. State Constraints Revisited – Extended Abstract. In *Proceedings of the 2nd Symposium on Logical Formalizations of Commonsense Reasoning*, pages 114–121, 1992.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: A Language for Representing Knowledge in Information Systems. *ACM Transactions On Information Systems*, 8(4):325–362, 1990.

[McC69] J. McCarthy. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 4*, pages 463–502. Edinburg University Press, 1969.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1991. Vol. 1: Specification.

[Pin94] J. Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, Department of Computer Science, University of Toronto, 1994. Also available as Tech. Report KRR-TR-94-1.

[PM95] Dimitris Plexousakis and John Mylopoulos. Accommodating Integrity Constrainst During Database Design, 1995. Submitted for Publication.

[Rei91] R. Reiter. The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Model for Goal Regression. In V. Lifschitz, editor, *Artificial Intelligence and the Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[vLDM94] A. van Lamsweerde, R. Darimont, and P. Massonet. Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt. Technical Report RR-94-10, Departement d'Informatique, Universite Catholique de Louvain, 1994.

A Generating Effect and Successor-State Axioms

For every predicate P that occurs in the postcondition $post_\alpha$ of action α the following process generates negative and positive effect axioms.[4] The formulae presented below are assumed to be universally quantified over actions.

1. Construct the following positive and negative axioms:

$$\forall \overline{x}/\overline{S}\ \forall s/S\ (\neg P(\overline{x}, s) \wedge P(\overline{x}, do(\alpha, s)) \wedge \alpha(\overline{x}, s) \Rightarrow False)$$
$$\forall \overline{x}/\overline{S}\ \forall s/S\ (P(\overline{x}, s) \wedge \neg P(\overline{x}, do(\alpha, s)) \wedge \alpha(\overline{x}, s) \Rightarrow False)$$

2. If $post_\alpha$ is $P(\overline{x}, do(\alpha, s))$ $(\neg P(\overline{x}, do(\alpha, s)))$, add $True$ as a disjunct to the positive (negative) effect axiom for P.
3. If $post_\alpha$ is of the form $\gamma(\overline{x}, s) \Rightarrow (\neg)P(\overline{x}, s)$, where γ does not contain terms referring to any situation except s, add a disjunct $\gamma(\overline{x}, s)$ to the positive (negative) effect axiom for P.
4. If $post_\alpha(\overline{x})$ is of the form $\exists \overline{z}\ (\gamma(\overline{x}, \overline{z}, s) \Rightarrow (\neg)P(\overline{w}, do(\alpha, s))$, where $\overline{w}$ consists of constants and variables from $\overline{x}, \overline{z}$, then augment the positive (negative) axiom for P with a disjunct $\exists \overline{z}\ (\gamma(\overline{x}, \overline{z}, s) \wedge (\overline{x} = \overline{w}))$.

This process results in a set of effect axioms of the form:

$$\forall \overline{x}/\overline{S}\ \forall s/S\ \neg P(\overline{x}, s) \wedge \neg\Phi_{1P}(\overline{x}, s) \Rightarrow \neg P(\overline{x}, do(\alpha, s)) \quad (1)$$
$$\forall \overline{x}/\overline{S}\ \forall s/S\ P(\overline{x}, s) \wedge \neg\Phi_{2P}(\overline{x}, s) \Rightarrow P(\overline{x}, do(\alpha, s)) \quad (2)$$

Let $\forall \overline{x}/\overline{S}, \forall s_1, s_2/S\ \phi(\overline{x}, s_1, s_2) \vee P(\overline{x}, s_1) \vee Q(\overline{x}, s_2)$ be a constraint that has to be satisfied at all times. Then, for each effect axiom of type (1) for P, the following axiom for Q is generated: $\forall \overline{x}/\overline{S}\ \forall s/S\ \neg P(\overline{x}, s) \wedge \neg\Phi_{1P}(\overline{x}, s) \wedge \neg\phi(\overline{x}, s, do(\alpha, s)) \Rightarrow Q(\overline{x}, do(\alpha, s))$ Symmetrically, for each effect axiom of type (1) for Q, generate the following axiom for P: $\forall \overline{x}/\overline{S}\ \forall s/S\ \neg Q(\overline{x}, s) \wedge \neg\Phi_{1Q}(\overline{x}, s) \wedge \neg\phi(\overline{x}, s, do(\alpha, s)) \Rightarrow P(\overline{x}, do(\alpha, s))$
The respective process takes place if the constraint contains negated predicates. Then we can generate *successor-state axioms* as follows: Let $\Psi_P(\overline{x}, s) = \neg\Phi_{1Q}(\overline{x}, s) \wedge \neg\phi(\overline{x}, s, do(\alpha, s))$ and $\Psi_{\neg P}(\overline{x}, s) = \neg\Phi_{2Q}(\overline{x}, s) \wedge \neg\phi(\overline{x}, s, do(\alpha, s))$. $\Psi_Q(\overline{x})$ and $\Psi_{\neg Q}(\overline{x})$ are defined analogous ly. Then, the successor-state axiom for P is:

$$\forall \overline{x}/\overline{S}\ \forall s/S\ \alpha(\overline{x}, s) \equiv [\Psi_P(\overline{x}, s) \vee (\neg\Psi_{\neg P}(\overline{x}, s) \wedge P(\overline{x}, s))]$$

Finally, knowledge producing actions can be taken into account as follows. An accessibility relation K between situations is introduced. For s, s' in S, $K(s', s)$ means that situation s' is accessible from s. Then, if actions $\alpha_1, \ldots, \alpha_n$ are all knowledge producing actions and each one is associated with a formula ϕ_i expressing the predicate or function term that becomes known, then the successor-state axiom for K is as follows:

$$\forall s, s''(K(s'', do(\alpha, s)) \equiv [\exists s'(K(s', s) \wedge (s'' = do(\alpha, s')) \wedge ((\alpha = \alpha_1) \rightarrow \phi_1) \wedge \ldots \wedge ((\alpha = \alpha_n) \rightarrow \phi_n))])$$

It has been shown in [LR92] that a set of successor-state axioms constitutes a solution to the frame and ramification problem if, for every predicate p, $\neg(\Psi_p \wedge \Psi_{\neg p})$ is entailed by the unique name axioms.

B Example: A Simple Elevator Controller

In this section we depict the application of specification strengthening to a simple example borrowed from [DvLF93]. Constraints here are expressed in a variant of first-order temporal logic[MP91]. The example aims at showing the applicability of the proposed analysis techniques to the case where specifications are given in a temporal logic. Hence, we will focus on postcondition strengthening rather than giving a fairly complete specification of the domain in question.

Let us assume that the constraint expressing the requirement that doors be closed while the elevator is moving ensures the abstract goal of safe transportation in the elevator system. The constraint is formally expressed as the following many-sorted temporal formula, where the operators $\bigcirc$ and $\odot$ mean "in then next state" and "in the previous state" respectively:

$$I \equiv \forall l/Lift\ d/Door\ f, f'/Floor\ \neg PartOf(d, l) \Rightarrow (LiftAt(l, f) \wedge \bigcirc LiftAt(l, f') \wedge (f \neq f') \Rightarrow ((d.state = \text{"}closed\text{"}) \wedge \bigcirc(d.state = \text{"}closed\text{"})))$$

For simplicity, let us assume that the only actions relevant to the constraints are the actions $GotoFloor$ and $OpenDoors$, whose specification is given in figure B. We need to add the axiom

$$\forall d/Door\ (d.state = \text{"}open\text{"}) \Rightarrow \neg(d.state = \text{"}closed\text{"})$$

in order to be able to reason about the state of elevator doors.

We first consider the action $GotoFloor$ for the derivation of ramifications. The first step is to instantiate the constraint I in the state after the action's execution. This amounts to parameterizing the universally quantified variables corresponding to the parameters of the action and to replacing every occurrence of $\bigcirc P$ by P and every occurrence of P with $\odot P$ for every predicate P. Negating

[4] We only show the derivation of effect axioms for predicates. Effect axioms for functions can be derived quite similarly.

Action *GotoFloor* (l, f, f')
  Precondition: $LiftAt(l, f)$
  Postcondition: $LiftAt(l, f') \ \wedge \ (f \neq f')$
end

Action *OpenDoors(l,d)*
  Precondition: $PartOf(d, l) \wedge (d.state = \text{"}closed\text{"})$
  Postcondition: $(d.state = \text{"}open\text{"})$
end

Figure 6: Specification of an Elevator Controller

the instantiated constraint and conjoining it with the action's postcondition yields

$$\neg I' \ \wedge \ Post \ \equiv \circledcirc PartOf(d, l) \ \wedge \ \circledcirc LiftAt(l, f) \ \wedge \ LiftAt(l, f') \ \wedge \ (f \neq f') \ \wedge \ (\neg \circledcirc (d.state = \text{"}closed\text{"}) \ \vee \ \neg(d.state = \text{"}closed\text{"}))$$

By exploiting the facts that the predicates occurring positively in the action's postcondition must evaluate to $True$ in the state after the action's execution and that the precondition must have been satisfied, $\neg I' \ \wedge \ Post$ implies the formula

$$\circledcirc PartOf(d, l) \ \wedge \ (\neg \circledcirc (d.state = \text{"}closed\text{"}) \ \vee \ \neg(d.state = \text{"}closed\text{"}))$$

We now exploit the assumption that the constraint was known to be satisfied in the state prior to the action's execution. This means that either the antecedent $\circledcirc PartOf(d, l)$ of I is $False$ or, both the antecedent of I and its consequent are $True$. In the former case we derive $False$ as a ramification and we can conclude that the constraint remain unaffected by the transaction. In the latter case, $\circledcirc PartOf(d, l)$ can be substituted by $True$ and the derived formula is $\neg R \ \equiv \ \neg \circledcirc (d.state = \text{"}closed\text{"}) \ \vee \ \neg(d.state = \text{"}closed\text{"}))$, which yield the ramification $R \equiv \ \circledcirc (d.state = \text{"}closed\text{"}) \ \wedge \ (d.state = \text{"}closed\text{"})$. Intuitively, this means that both in the state before and in the state after the action execution, the state of the elevator door must be "closed".

Similarly, we can derive ramifications with respect to the action $OpenDoors$. In this case, the ramification derived is expressed by the formula

$$R \ \equiv \ \exists f, f'/Floor \ (\circledcirc LiftAt(l, f) \ \wedge \ LiftAt(l, f')) \ \Rightarrow \ \neg(f \neq f')$$

which means that both before and after the action execution the elevator must be at the same floor. It is easy to verify that if we strengthen the action specifications by conjoining their postconditions with the derived ramifications, then the constraint is maintained by any implementation of the actions that meet the new specifications. Figure B shows the strengthened specification of the actions $GotoFloor$ and $OpenDoors$.

Action *GotoFloor* (l, f, f')
  Precondition: $LiftAt(l, f) \ \wedge \ (d.state = \text{"}closed\text{"})$
  Postcondition:
    $LiftAt(l, f') \ \wedge \ (f \neq f') \ \wedge (d.state = \text{"}closed\text{"})$
end

Action *OpenDoors(l,d)*
  Precondition: $PartOf(d, l) \wedge (d.state = \text{"}closed\text{"})$
  Postcondition:
    $PartOf(d, l) \ \wedge \ (d.state = \text{"}open\text{"}) \ \wedge$
    $\exists f, f'/Floor \ (\circledcirc LiftAt(l, f) \wedge LiftAt(l, f')) \ \Rightarrow$
    $\neg(f \neq f')$
end

Figure 7: Strengthened Specification of an Elevator Controller