The effects of buying a new car: an extension of the IDP Knowledge Based System

Pieter Van Hertum
Joost Vennekens
Bart Bogaerts
Jo Devriendt
Marc Denecker
Report CW 640, June 2013

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

The effects of buying a new car: an extension of the IDP Knowledge Based System

Pieter Van Hertum
Joost Vennekens
Bart Bogaerts
Jo Devriendt
Marc Denecker
Report CW 640, June 2013

Department of Computer Science, KU Leuven

Abstract

A long term goal in knowledge representation is the development of a knowledge based system (KBS). A KBS stores knowledge in a completely declarative way and is equipped with various domain independent inference methods to perform different reasoning tasks for a broad field of applications. In this paper we took a state-of-the-art KBS, IDP, with corresponding language FO(), and tested its applicability on a prototypical example from the business rule domain: a car rental system. We investigated whether we can indeed represent all required knowledge in FO(). The results are mixed: on the one hand, the expressible knowledge can be used to a greater extent (for example for profit maximisation), but on the other hand, some knowledge was not expressible in the current version of our system. One obstacle we encountered is that FO() does not have a good formalism to model the knowledge about situations where a new object is created. To remedy this, we propose an extension of the concept of inductive definitions which allows us to model this knowledge in a declarative way. As a result, this extension can be used for the car rental problem domain, as well as for many other applications and inference tasks.

Keywords : IDP, Knowledge Based System, Business Rules.

The effects of buying a new car: an extension of the IDP Knowledge Based System

Pieter Van Hertum and Joost Vennekens and Bart Bogaerts
Jo Devriendt and Marc Denecker
Department of Computer Science, KU Leuven

1 Introduction: A broadly applicable KBS-system

Computer science is – to a large extent – the study of information: how it can be stored, manipulated, transferred and, ultimately, used to solve the users' problems. However, the role that information plays in typical day-to-day software applications is currently still not very well understood. Such applications typically incorporate domain knowledge that a programmer has encoded in, e.g., a Java program, interact with information that is stored in a database, and obtain information from an end user. A detailed analysis of what these different "informations" are and how they interact will contribute to a greater understanding of the way in which such applications work, which may in turn eventually help to develop applications faster and with fewer bugs. Declarative programming is obviously an area that has great promise towards such an understanding, since it investigates precisely the use of declarative specifications – i.e., of information – to solve problems.

However, traditional Logic Programming systems have, from this perspective, one serious drawback, namely, that they always try to solve one particular task (query answering) by means of one particular class of algorithms (SLD-resolution and its kin). This is, on the one hand, convenient because it allows us to supplement the declarative semantics of a logic program with an operational semantics. On the other hand, though, it also means that they will never be able to help us understand how, during the life-time of an application, the same information may be used in different ways to compute different things and to perform different tasks. For this reason, our research group is currently attempting to build a Knowledge Based System (KBS), in which knowledge — in the particular case of our system, expressed in a combination of first-order logic and logic programs — is stored in a purely declarative way, such that it may be used by different inference algorithms to perform different tasks (Denecker and Vennekens 2008). The hope is that such a KBS will eventually allow us to build better models of real-world software systems.

In (Vlaeminck et al. 2009), this idea was first put to the test in the context of configuration software, where the goal is to help a user interactively construct a configuration that conforms to a specification. In this paper, we examine a more ambitious application. To be more concrete, we consider an existing description of the desired behaviour of a car rental system (Group 2000). This problem is a well-known benchmark application for business rule systems and includes various sorts of tasks and problems that, from a logic perspective, require a very rich logic and different forms of inference. We investigate whether we can indeed represent this knowledge in the language of our KBS, and, moreover, do this in such a way that all the different functionality that such a system needs to exhibit can indeed be characterised as different logical inference tasks applied to this same knowledge base. The results are mixed.

On the positive side, it turns out that our current KBS is indeed able to allocate cars to customers, based on a declarative specification. Moreover, it can do this in a way that is superior to how current state-of-the-art business rule systems perform this task (Halle 2001). However, on the negative side, it turns out that our system has difficulty in performing the comparatively mundane tasks of simply registering a new car into the system. This is due to the fact that the current system is based on inference tasks that construct models of a logical theory given some fixed set of objects that is to serve as the domain of these models. An alternative would be to rely instead on methods that allow for arbitrary domains, but that is also not what is needed here. What we need instead is a principled way of taking, in certain circumstances, an existing domain (e.g., the current set of cars) and then extending it with new elements (the newly bought car that is to be registered).

In the second part of this paper, we will show how we can extend the language of our KBS to allow such specifications. We discuss both the semantics of this new language construct and its implementation.

2 The IDP system for FO(·)

The term FO(·) stands for the class of conservative extensions of classical first-order logic (FO). The IDP system discussed below supports an FO(·) language. This section provides an overview of the core of the language followed by a description of the functionalities of the IDP system.

We assume familiarity with basic concepts of classical logic and of logic programming. A vocabulary Σ is a set of function and predicate symbols. FO terms and FO formulas over Σ are defined using the well-known inductive rules for $\neg, \wedge, \vee, \forall, \exists, \Rightarrow, \Leftrightarrow$. An FO sentence is an FO formula without free occurrences of variables. The formal model semantics of FO is based on the notion of Σ-structures I which consist of a domain $dom(I)$, and an assignment of appropriate values σ^I in this domain to each symbol σ of Σ. The foundational definition of the truth/satisfaction $I \models \varphi$ of an FO sentence φ in a structure is well-known. From it, one derives the concepts of a model of a sentence, satisfiable sentences, valid sentences and entailment $\models$ between sentences or theories.

FO is extended here in various ways. First, we use a typed version of FO. Thus, vocabularies include types, and function, predicate and variable symbols are typed. Well-formed terms, formulas and sentences have the property that the type of each symbol occurrence matches the type of its argument position. Second, aggregate expressions can be included in FO. In the examples in this paper, we use the cardinality aggregate # in terms of the form $\#\{\bar{x} : \psi\}$. Third, we extend FO with a definition construct. A theory may contain multiple (formal) definitions. A (formal) definition Δ is a set of *definitional rules* of the form:

$$\forall \bar{x}(A \leftarrow \varphi)$$

where A is an atom and φ an FO formula. We call a predicate in A a *defined predicate* of Δ; any other function or predicate symbol in Δ is called a parameter symbol of Δ. The formal semantics of definitions is an extension of well-founded semantics (Van Gelder et al. 1991) to arbitrary FO-structures (not just Herbrand structures) and the extended syntax and was defined in (Denecker 2000) for the first time. Informally, a structure I satisfies Δ if the interpretation of a defined predicate P in the well-founded model of Δ, constructed relative to the restriction of I to the parameters of Δ, is exactly the relation P^I. The definition construct is modelled syntactically and semantically after the informal linguistic concept of definition as found in mathematical and scientific texts. For further explanation we refer to (Denecker and Ternovska 2008).

FO(·) is a "true" declarative logic in the sense that, contrary to procedural languages and most declarative programming paradigms, it has no operational semantics and has no unique associated form of inference. Hence, a specification of domain knowledge may be reused for solving multiple tasks and problems (Denecker and Vennekens 2008). This leads to the idea of a knowledge based system: a system that manages a logic theory and provides several forms of inference on them. The IDP system is a knowledge based system that is being developed to

support an extension of the above core FO($\cdot$) language with aggregates, interpreted arithmetic symbols, defined and partial functions, etc. The system manages a set of logic objects in the form of vocabularies, formulas, theories and structures and implements several forms of inference on them (Denecker and Vennekens 2008):

- Model checking and querying: Input: FO($\cdot$) sentence or theory T, structure I; output "true" iff $I \models T$. More generally, query answering: given a structure I and a set comprehension $\{\bar{x} : \psi\}$; output $\{\bar{x} : \psi\}^I$.
- Model expansion (MX): Input: FO($\cdot$) theory T, a finite partial structure I; output: models M of T that expand I. Notice that although FO($\cdot$) is an open domain logic, a model expansion task fixes the domain through input I.
- Herbrand model generation: Input: FO($\cdot$) theory T; output: Herbrand models M of T. Both MX and Herbrand model generation can be combined with optimisation of objective functions over structures.
- Propagation inference: Input: a theory T and partial structure I; output: a refined partial structure I' that approximates all models of T that refine I.
- Deduction: Input : an FO($\cdot$) theory T, FO sentence ψ; output: "true" iff $T \models \psi$. The method is sound but incomplete, and at present works only for a fragment of FO($\cdot$). It is implemented using the theorem prover Spass ().
- Model revision: Input: FO($\cdot$) theory T; a model M of T, a formula φ; output: models M' of $T \cup \{\varphi\}$ that are minimally different from M according to some objective function.
- Definition model expansion: Input: a definition Δ, a structure I interpreting all parameter symbols of Δ; Output: the unique structure I' of the defined symbols such that $I \cup I' \models \Delta$.

The IDP system comes with a programming environment in the scripting language LUA (De Pooter et al. 2011). IDP-lua programs may call various functionalities including the above forms of inference on the logic objects. Many other functionalities are available as well, e.g., to create or modify vocabularies, structures, theories; to parse or simplify formulas, put them in negation normal from, transform them into BDD's, etc.

One goal of this paper is to illustrate the reuse of FO($\cdot$) specification for solving multiple tasks and problems using various forms of inference. An earlier experiment in this is described in (Vlaeminck et al. 2009) where an interactive configuration tool for study programs was implemented in IDP that used five different forms of inference on the same theory to provide various functionalities of the software.

In the sequel, with slight abuse of terminology, we use the term FO($\cdot$) to refer to the language supported by the IDP system.

3 The Car Rental application

The goal of our KBS is to implement complex behaviours in practical applications by means of declarative specifications. The current industry standard for this appears to be the use of *business rule systems*. In such a business rule systems, applications are encoded in the form of sets of procedural "IF-THEN"-rules. The rules derive their (operational) meaning from the way they are interpreted in the match-resolve-act cycle, in which the THEN part of rules may be derived or executed given the satisfaction of the IF-part at some stage in the process. By contrast, an FO($\cdot$) theory (which might contain several types of rules) has only a declarative meaning, cannot be executed, but can be reasoned upon by various inference mechanisms. Our KBS does not encode a specific problem, only a specification of the problem domain. An advantage of this approach is that we can re-use the knowledge for implementing several functionalities.

To compare the current state of our KBS with that of business rule systems, we will focus on a standard benchmark for business rule systems: the EU-Rent application.

3.1 Case study: the EU-Rent application

EU-Rent is a fictitious car rental company spread over different countries. It has a collection of cars that are located in specific branches. They are allowed to be moved between the branches and a customer can rent a car in one branch and return it to another branch. The application has to process all reservations and make a planning so that cars are at the right branch on the right time. Moreover, it should fulfill as many reservations as possible, while minimising the expenses. Next to this obvious task, it should be able to process smaller changes to the database, like adding or deleting a reservation, buying or selling a car, ...

The EU-Rent application is a well-known benchmark application used in literature to evaluate business rule systems. This application provides a simple but varied class of use cases to evaluate the expressivity and functionality of such systems.

Here we examine two specific use cases that we will try to specify in FO(·), after which the IDP system can run the appropriate inferences. The first use case is the optimal planning and allocation of cars to a given collection of reservations. Because our system is well-suited to solving search problems of this kind, it will turn out that we can perform this task quite easily. The other use case is the purchase of a new car (or the decision of the system to purchase a new car when certain conditions are satisfied). This second use case may seem like a simple operation, but it will actually turn out to be more difficult to model in the declarative language FO(·). This is because most of the inference tasks that are supported by our current KBS are concerned with generating or expanding models of a theory, given a fixed set of objects that is to serve as the domain of these models. However, in this use case, the set of cars can be modified, which will require us to extend the language.

The full EU-Rent application is quite big. For the purpose of this paper we work out a simplified version. Figure 1 represents the domain model of this simplified version, which consists of four types of objects: drivers, cars, reservations and branches. Each car has a purchase date and a mileage count. For each driver we know whether he or she has a driver's license. Reservations are requests to hire a car for a certain period, denoted by a start and end date. Reservation can be accepted or rejected, which is represented by a boolean attribute.

The objects themselves are related to each other, represented by the arrows in Figure 1. Every reservation has one or multiple drivers associated with it, can have a car allocated to it, and is requested to be picked up at a certain branch of the company. Every car is stationed at a certain branch.

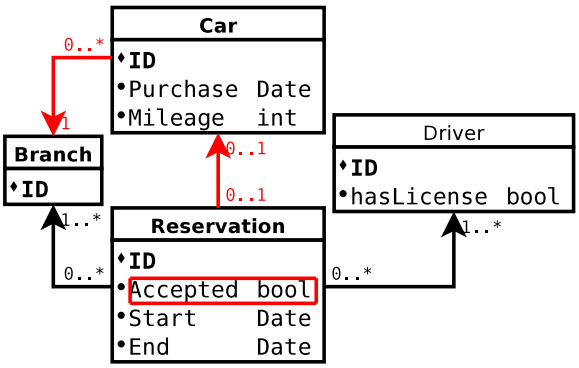

Figure 1. Domain model of the modelled EU-Rent application

Our fragment consists of 4 types of objects: drivers, cars, reservations and branches. A car has a purchase date and a mileage. About a driver we know if he has a license or not. Reservations are requests to reserve a car, during a certain period, given by its start and end date. Reservations also have a boolean property that represents whether the system has accepted the request. These

objects are related to each other, represented by the arrows in Figure 1. Every reservation has one or more drivers associated to it, may have a car, and is made in a branch (where the car will be collected). Every car is currently allocated to a branch, where it is stationed.

4 Towards a declarative implementation of the EU-Rent application with IDP

4.1 General Description: Valid States

A car rental system maintains a state recording available cars, reservations, planning of car transports, information about clients, etc. In this section, we describe the states and their invariants.

In order to express all invariants a state should satisfy, we introduce a vocabulary based on the domain specification of Figure 1. This vocabulary will be used to represent a single state of the application. A structure of this vocabulary records the actual state of the world (which cars are where, who has requested which reservations, etc.), as well as the current plans of the system (which reservations will be fulfilled by which cars, when should cars be moved between branches, etc.). The vocabulary can be found in Figure 2. It is represented as the union of Σ_{types} of type symbols and Σ_{fp} of function and predicate symbols. Note that predicate symbols are expressed as boolean functions.

$$
\begin{aligned}
&\Sigma_{types} = \{Car, Driver, Res, Branch, Date\}\\
&\Sigma_{fp} = \{\\
&\quad SerialNr : Car \to Int,\ PurchaseDate : Car \to Date\\
&\quad Mileage : Car \to Int\\
&\quad BirthDate : Driver \to Date,\ HasLicense : Driver \to Bool\\
&\quad Accepted : Res \to Bool\\
&\quad StartDate : Res \to Date,\ EndDate : Res \to Date\\
&\quad InitialLocation : Car \times Branch \to Bool,\ Location : Car \times Date \times Branch \to Bool\\
&\quad Location : Res \to Branch\\
&\quad Allocated : Res \times Car \to Bool,\ Driver : Res \to Driver\\
&\quad Overlapping : Res \times Res \to Bool\\
&\quad Move : Car \times Branch \times Branch \times Date \to Bool\\
&\}
\end{aligned}
$$

Figure 2. Vocabulary SingleState

Our implementation of EU-rent will maintain a structure of vocabulary SingleState and will modify it using suitable transactions. Even though we are interested in modelling the behaviour of a dynamic system, we first consider a static aspect of the system. At every time point, assignments of reservation should satisfy certain invariants. In typical Business Rule solutions, these invariants are left implicit and the programmer should make sure to specify the dynamic behaviour of the system in such a way that the invariants will always hold during the execution of this specification. However, since we use logic as a modelling language, in our approach it is a natural choice to explicate those invariants and, possibly, use techniques like theorem proving to prove that our dynamic system actually respects those invariants.

The laws of the car scheduling problem that we extracted from the specification can be found in the theory ValidState in Figure 3. They form a set of invariants of the system. An interesting feature of this theory is that although it specifies conditions that should be satisfied in each individual state of the system, it is still a temporal theory. Indeed, the laws of reservations are

temporal in nature, e.g., to state that the same car should not be assigned to two reservations that overlap in time.

Note that the combination of a structure of vocabulary SingleState and the definitions of predicates *Location* and *Overlapping* can be viewed as kind of deductive database in which all of the FO axioms in theory ValidState can be evaluated. This leads us to a first use of theory $ValidState$. Assume that an employee manually updates the current state of the system. Such things frequently happen. In that case, model checking inference can be applied on ValidState to check if the invariants are intact.

//Relation Constraints
$\forall r : \#\{c : Allocated(r, c)\} \leq 1.$
$\forall c : \#\{r : Allocated(r, c)\} \leq 1.$
$\forall c\ d : \#\{b1\ b2 : Move(c, b1, b2, d)\} \leq 1.$
$\forall c\ b\ d : \neg Move(c, b, b, d).$

//Preconditions for accepting a reservation:
$\forall r : Accepted(r) \Rightarrow \exists c[Car] : Allocated(r, c).$
$\forall r\ c : Allocated(r, c) \Rightarrow Location(c, StartDate(r)) = Location(r).$
$\forall r : Accepted(r) \Rightarrow HasLicense(ResDriver(r)).$

//Maximum one reservation at a time per Driver
$\forall r1\ r2 : Overlapping(r1, r2) \wedge r1 \neq r2 \Rightarrow \neg Accepted(r1) \vee \neg Accepted(r2)$
$\quad \vee Driver(r1) \neq Driver(r2).$

//Location is defined in terms of moves and initial location

$$\left\{ \begin{array}{l} \forall c\ b1 : Location(c, Today, b1) \leftarrow InitialLocation(c, b1). \\ \forall c\ d\ b1 : Location(c, d+1, b1) \leftarrow Location(c, d) = b1 \wedge \neg(\exists b2 : Move(c, b1, b2, d)) \\ \qquad\qquad \wedge\neg\,(\exists r : Location(r, b) \wedge Accepted(r) \\ \qquad\qquad \wedge Allocated(c) = r \wedge StartDate(r) = d)\,. \\ \forall c\ d\ b2 : Location(c, d+1, b2) \leftarrow \exists b1 : Move(c, b1, b2, d). \\ \forall c\ d\ b2 : Location(c, d+1, b2) \leftarrow \exists r : Accepted(r) \wedge Allocated(c) = r \wedge EndDate(r) = d+1). \end{array} \right\}$$

//Precondition for the move action
$\forall c\ b1\ b2\ d : Move(c, b1, b2, d) \Rightarrow Location(c, d) = b1.$

//Definition of Overlapping

$$\left\{ \begin{array}{l} \forall r1\ r2 : Overlapping(r1, r2) \leftarrow StartDate(r2) \leq EndDate(r1) \wedge StartDate(r1) \leq EndDate(r2). \end{array} \right\}$$

Figure 3. The scheduling theory ValidState

4.2 Building a dynamic system

The EU-rent system that we implement is a dynamic system in which new reservation requests are made, existing are cancelled, new cars are bought, or cars are sold or wrecked in accidents, or stolen, reservation dates are changed, etc.. These represent a large number of use cases two of which will be worked out.

The backbone of the solution is a dynamic linear time theory representing how transactions of the above kind modify the state of the system. To express this theory, we will need a new vocabulary SequenceOfStates, such that each structure for this vocabulary now corresponds to an entire sequence of states. This can be derived from our vocabulary SingleState in a principled way, by introducing an additional type State and then adding a state argument to all predicates and functions that may change due to transactions. In the case of our example, these are the

functions and relations that are drawn in red in Figure 1. This new vocabulary allows us to model our system over a series of states.

In what follows, we will present various logical theories over (extensions of) vocabulary SequenceOfStates. Such a logical theory will represent all valid executions of the intended dynamic system. E.g., the following pair of definitional rules expresses that accepted reservations are maintained unless cancelled and that new reservation requests (represented by the partial function $NewRes(t)$) are accepted iff a car was allocated to them.

$$\left\{ \begin{array}{ll} \forall r\ t : Accepted(r, t+1) & \leftarrow Accepted(r, t) \wedge \neg Cancelled(r, t) \\ \forall t : Accepted(NewRes(t), t) & \leftarrow \exists c : Allocated(NewRes(t), c, t). \end{array} \right\}$$

Figure 4. Definition of Accepted

Such a logical theory will represent all valid executions of the intended dynamic system and can be used to simulate, verify, execute, ... Let us focus on execution.

As mentioned above, the EU-rent application is a dynamic system that maintains a state. This state is represented by a structure I of vocabulary SingleState. At any stage, users may apply a transaction on this state. E.g., a reservation request may be issued. What form of inference is required to implement such transactions? Essentially it is a form of *progression inference* (Lin and Reiter 1997): computing a possible next state from a previous state given a temporal theory.

At present, IDP does not support progression inference but it can be emulated through the following steps. With the current state I and the requested transaction, a partial structure I' over SequenceOfStates can be built with two states $0, 1$ in which the interpretation of *Accepted* and *Allocated* at state 0 is given by I, and is undefined at state 1. Additionally the given transaction is inserted in the interpretation (e.g, as an interpretation of $Cancelled$ or of $NewRes$). The values of *Accepted* and *Allocated*, and of *Available* and *Candidate* at state 1 are then computed by definition model expansion on input I' using the theory. By projecting away the state 1 argument from these relations, the new structure resulting from the transaction is obtained.

4.3 A rule-driven approach for scheduling reservations

In our first use case, the goal is to handle new reservations for renting a car. Depending on the available cars, such a request can be denied or accepted, in which case the planning must be modified. We solve this use case in two ways. A first approach is by mimicking a business rule solution to this use case. In a second approach in the next section, we propose an alternative solution that more directly uses the theory ValidState.

In a business rule solution, If-Then rules are used to implement an algorithm that will decide whether cars are available, and if so, will update the current reservation planning. In practice, the business rule engine is run on a rule base and a working memory that includes the current state and the new request. In our car rental example, the addition of a fact representing a reservation request might trigger rules in such a way that the resulting information allows or prevents the reservation's acceptance.

Business rules often implement a sort of logic rules of the kind found in definitions and their operational semantics corresponds to a sort of bottom up evaluation of definitions similar to the computation of intentional predicates in deductive databases. This is of course not always true but it was the case in this use case. Not surprisingly then, we could easily turn these rules into definitional rules of FO$(\cdot)$. For example, a typical business rule could be that a reservation can be accepted if a car is available for the required period, and a car is available if it is not allocated to overlapping reservations. If cars are available, another rule would assign an available one for the reservation (the smallest in some order). The definition in Figure 5 augmented with the rules for *Accepted* in the previous section, captures this logic but in a declarative way through

definitions. Much more elaborate reservation strategies can be implemented in this style but here we just show the principle.

$$\left\{\begin{array}{ll}
\forall c\ r\ t : Available(c,r,t) & \leftarrow \forall r2 : r \neq r2 \wedge Allocated(r2,c,t) \\
 & \quad \Rightarrow \neg Overlapping(r,r2). \\
\forall c\ t : Candidate(t+1) = c & \leftarrow Available(c, ReqRes(t), t+1) \wedge \\
 & \quad (\forall c2 : c2 < c \Rightarrow \neg Available(c2, ReqRes(t), t+1)). \\
\forall c\ t : Allocated(ReqRes(t), c, t+1) & \leftarrow c = Candidate(t+1). \\
\forall r\ c\ t : Allocated(r,c,t+1) & \leftarrow Allocated(r,c,t) \wedge r \neq ReqRes(t).
\end{array}\right\}$$

Figure 5. Definition RentalRequest

The Theory RentalRequest can now be used to execute a reservation request transaction as described in the previous section, using progression inference.

Other inferential uses of the above theory are for *verification* and *simulation*. Essentially, RentalRequest implements an algorithm that should transform a state satisfying theory ValidState into another state satisfying this condition. By applying model expansion inference on RentalRequest together with instances of axioms of ValidState for state 0 and the negation of instances of axioms for state 1, we search for errors in the system in a way similar as in Alloy (Jackson 2002). More ambitiously, if from a theory including RentalRequest and the instance of ValidState for state 0, it is possible to prove the instance of ValidState at state 1 (using deductive inference), we have established that the axioms of ValidState are indeed invariants of the system. Such applications are only possible with declarative theories.

The work in this paper is new and lack of time has prevented us to build a fully automated system performing the above forms of inference. What is missing in the current implementation are the automatic theory translations of adding and dropping arguments to parts of the theories. This is not hard to implement using IDP's interface for manipulating theories and formulas.

4.4 A search-based approach for scheduling reservations

The business rule-like definition of the previous section can be refined to implement improved reservation algorithms that, e.g., would also re-schedule assignments to previous car reservations to be able to satisfy more requests or to be more profitable for the company by reducing expenses or increasing income. However, such a methodology soon becomes very tedious and error-prone. Luckily, the power of logic can be used to simplify this task. The theory ValidState of invariants in Figure 3 can be used more directly to automatically generate new states that are correct and/or optimise some objective function.

The idea is simply to add the invariants of theory ValidState at each state. This is achieved by taking theory ValidState, rephrasing it in the vocabulary SequenceOfStates with a state argument s and quantifying s universally. In the resulting theory, the scheduling predicate *Allocate* is now undefined (has no definitional rules) and its value is merely constrained by the theory. This theory describes our whole system over a long period of time. We can now use this theory to execute a transaction request, simulate runs of our system. Given a run of our system, we can use this theory to see if it was a valid run. We can also use it to simulate a run, if we have a structure representing the current state and we can add some decisions in the future, we can check if there exists a valid run of the system, that starts in the current structure and executes the given decisions.

Advantages of this approach over the business rule one are simplicity and guaranteed correctness. Indeed, provided that the theory ValidState is correct, it automatically gives us a correct solution. An operational difference with the BR approach is that the result is no longer deterministic: at each state, the system can freely choose a new schedule. This may pose practical

problem for the company in the sense that planning of manpower and resources becomes very difficult (e.g. will it be necessary to move a car from one branch to another?). A solution for this problem would be to apply model revision or to compute solutions minimising an objective function that penalises modifications to the previous schedule.

4.5 Maximising Profit

In Section 4.4, we specified a dynamic system that is able to generate any valid next state. This contrasts sharply with the rule driven approach, where a deterministic description of a next state is specified. In practice, the main and only concern of a business manager is to make profit. Therefore, such a manager will not care which next state is selected by a system as long as the decisions made by the system are profitable. This is in fact a search problem, where we want to maximise a certain profit function, namely, the income we obtain from renting out cars minus the sum of all expenses.

In fact the ultimate goal is, given a current state of the system and some new reservations, to calculate a new optimal state of the system: a new optimal plan incorporating new reservations, accepting as many as possible, with making as few changes as possible. A disadvantage of the rule based approach is that the used optimality criterion is left implicit. This new state should definitely be a valid state, hence it hast to satisfy the theory in Figure 3. The rule based approach is a first attempt to maximise the profit: in that approach, manual heuristics, such as "preferably assign a free car to a new reservation, only if that is impossible, try shifting the schedule". In this section, we will describe a more fundamental approach, using a weighted model revision inference. The input for the model revision inference are a theory which the new model has to satisfy, a structure that is to be revised, and a mapping from predicates and functions in the vocabulary to tuples of integers: the cost to make this predicate/function more true and the cost to it make more false. In this specific situation we give these arguments to our model revision:

$$CurrentState,\ ValidState,$$
$$\{Accepted \mapsto (100, -300), Location \mapsto (-50, -50), Allocated \mapsto (-20, -20)\}$$

This input expresses for example that only Accepted, Location and Allocated can be changed. We also see that it is very good (profit 100) to accept more reservations, while it is costly (profit -300) to cancel a previously confirmed reservation. Of course, these parameters can be fine-tuned depending on real life information. The weighted model revision then calculates a new model: a revision of the old model that maximises the profit (sum over all changed atoms of the weighted assigned to that particular change). Even though optimisation is not a part of progression inference described in 4.2, these optimisation steps still fit in that paradigm. In fact, in order to express a step-by-step optimisation in a general dynamic system, we would need second-order quantifiers. Progression inference on such a theory would then be the same as executing a minimisation step at every time point. Unfortunately, the IDP system does not support these expressive constraints yet. Other systems, such as the Pro-B system (Leuschel and Butler 2008) do support higher-order constructs and could in theory be used to execute other inferences with such higher-order theories.

4.6 Adding New Objects.

Our second use case is an extension of the first one: we want to extend the theory in Figure 3 with the possibility of adding new objects. In many software systems, adding a new object is a trivial task: new classes or new entries in a database are constantly being created. The car rental problem domain also allows for cars to be purchased, and hence it must be possible to add a new car to the set of cars. In logic speak: it must be possible to add a new object to the Car domain.

The obvious way to do this would simply be to extend the structure (or database) with an extra car. However, this means that the knowledge about the introduction of a new car is not

present in the logical theory representing the problem domain. This also means that information about this transaction is not reusable for other forms of reasoning, e.g. making simulations of the system, verification, etc. To remedy this, we propose a new logical operator representing the knowledge that a new domain element is created.

Before we explain the exact syntax and semantics of this operator in Section 4.7, we take a look at how the addition of a novel logic operator could solve the creation of a car in our car rental problem domain:

$$\left\{ \begin{array}{l} \forall s\ sn:\ \textbf{new}\ c:\ SerialNr(c) = sn \\ \qquad \wedge PurchaseDate(c) = s \wedge Mileage(c) = 0 \leftarrow BuyCar(sn, s). \end{array} \right\}$$

This definition states that if a car with serial number sn is bought in a certain state s, a new car has to be added to the system with that specific serial number, purchased at that point in time, with zero mileage, and some other initial attributes. Note that we added a predicate $BuyCar(Int, State)$ to the vocabulary SequenceOfStates, representing the serial numbers of any cars bought. In order to support this kind of expressions, we introduce a new logic, FO(ID^+), which basically extends FO($\cdot$) by allowing more complex formulas in the head.

4.6.1 FO(ID^+)*: Syntax and Intuitions*

Definition 4.1. A **head formula** is defined inductively as follows:

- any atom is a head formula
- if φ_1 and φ_2 are head formulas, then $\varphi_1 \wedge \varphi_2$ is a head formula
- if φ is a head formula, then **new** $x : \varphi$ is also a head formula.

An extended inductive definition is a set of rules of the form

$$\forall \bar{x} : \varphi_h \leftarrow \varphi_b$$

where

- $\bar{x}$ is a tuple of variables;
- φ_h is a head formula and φ_b is a first-order formula;
- all free variables of φ_h and φ_b are among the variables in $\bar{x}$.

Like in informal definitions, any rule produces a set of atoms described by the head formula. Even though conjunctive head formulas resemble simple FO formulas a lot, we emphasise that the intended meaning of an extended rule is **not** "if a body is true, the corresponding head formula is true". No, instead the intuition should be: if the body of a certain rule is true, then the head formula will determine a set of atoms to be true (and nothing more). The reason for this is that inductive definitions describe a process that starts from an empty interpretation for open symbols and keeps executing rules (or deriving that certain heads can never be derived anymore) until fixpoint. In the extended formalism, one rule can make several atoms true at once (and create new elements). Conjunctive head-rules represent multiple effects of the same action: multiple things becoming true together. Hence, the set of atoms made true is simply the union of the atoms made true by subformulas. The **new** operator causes the creation of a new domain element different from all other elements of this same type. Of course, in logic no new elements get created. These are just the intuitions how such rules should be read. We now describe a formal semantics for this logic that preserves the intuitions described in this section.

4.7 FO(ID^+)*: Semantics*

We define a transformational semantics for FO(ID^+): a transformation from FO(ID^+) to FO(ID) with an explanation of how this transformation preserves the intuitions from the previous section.

The transformation we describe is a recursive transformation that translates a rule in one or more rules with strictly smaller head formula. We recursively apply this transformation until all

rules have simple heads (atoms). In order to do this, for every defined sort S, we introduce a new predicate InSort_S representing all "created" elements. This predicate will be used to formulate the Domain Closure Axioms (DCA).

The simplest rules to transform are conjunctive rules.Since they represent multiple effects of the same action, they can simply be translated to multiple rules.

The harder type of rules to translate are the new-rules since they simultaneously define a new sort and properties for the newly created elements of that sort. In order to translate a rule of the form

$$\forall \bar{x}[\bar{t}] : \textbf{new } y[s] : \varphi_h(\bar{x}, y) \leftarrow \varphi_b$$

we introduce a new partial function symbol $F : \bar{t} \to s$ (for every "new-rule", this is a new function) that maps every tuple $\bar{x}$ for which φ_b is true to the associated domain element and a new predicate $\text{Dom}_F(\bar{t})$, representing the domain of F. I.e. all tuples for which a new element is created. Then, we transform the "new-rule" into these 3 rules

$$\begin{aligned} \forall \bar{x} : \text{Dom}_F(\bar{x}) &\leftarrow \varphi_b \\ \forall \bar{x}, y : \text{InSort}_s(y) &\leftarrow \text{Dom}_F(\bar{x}) \wedge F(\bar{x}) = y \\ \forall \bar{x}, y : \varphi_h(\bar{x}, y) &\leftarrow \text{Dom}_F(\bar{x}) \wedge F(\bar{x}) = y \end{aligned}$$

The intuitions behind these rules are the following: by doing this for every rule containing creator symbols, we get that InSort_s is defined as the union of all elements that occur as image of some creation function. I.e. $InSort$ will be true exactly for all those elements such that a new-rule gets triggered. Furthermore, for those elements, φ_h should hold. In order to make this work completely, we should still add unique name axioms (UNA) for the created elements:

- Every creatorfunction is injective: $\forall \bar{x}, \bar{y} : F(\bar{x}) = F(\bar{y}) \Rightarrow \bar{x} = \bar{y}$.
- Elements created by different creatorfunctions are unique: $\forall \bar{x} : F_1(\bar{x}) \neq F_2(\bar{x})$

Besides UNA, a few more constraints are needed to be added to the theory. First of all, domain closure axioms: the defined types consist exactly of those elements that are "created" by a certain constructing rule. Secondly, we want to establish the link between the domain predicate and the corresponding partial function: creatorfunctions are defined on (and only on) their domain. Hence we add the following constraints to our theory:

- $\forall x[S] : \text{InSort}_S(x)$,
- $\forall \bar{x} : \text{Dom}_F(\bar{x}) \Leftrightarrow \exists y[s] : F(\bar{x}) = y$.

4.7.1 Implementation: Modelexpansion for $\text{FO}(ID^+)$ *Theories*

In fact in order to reason with defined sorts, we should be able to reason with unknown, open, domains. Since the IDP system doesn't support open-domain reasoning yet, we implemented this as follows. Initially, we overapproximate all defined sorts, i.e. we replace every defined sort with a (possibly infinite) set of new domain elements. These are all elements that might possibly be created. Since we overapproximate sorts, we should be careful: all quantifications over defined sorts should become conditional, i.e. we replace any universal quantification $\forall x[S] : \varphi(x)$ by $\forall x[S] : \text{InSort}_S(x) \Rightarrow \varphi(x)$ and any existential quantification $\exists x[S] : \varphi(x)$ by $\exists x[S] : \text{InSort}_S(x) \wedge \varphi(x)$.

After solving, we redefine the created type by filtering out all the created domain elements for which $InSort_s$ is not true. If we now project the calculated structure on our original vocabulary, we get a structure which satisfies all constraints and where the new elements have been created.

4.7.2 symmetry breaking

It should be clear that all domain elements in created sorts are mutually interchangeable. Hence in order to avoid an explosions of the number of models of our theory, we should impose some restrictions. Let $\leq$ be any total ordering on our domain elements (including the infinitely many

domain elements in the overapproximations of defined sorts), extended to tuples using lexicographic ordering and let $\trianglelefteq$ be any total ordering on the creatorfunctions for a given sort. Then we add the following constraints to our theory:

- $\forall \bar{x}, \bar{y} : \mathrm{Dom}_F(\bar{x}) \wedge \mathrm{Dom}_F(\bar{y}) \wedge \bar{x} \leq \bar{y} \Rightarrow F(\bar{x}) \leq F(\bar{y})$
- $\forall \bar{x}, \bar{y} : \mathrm{Dom}_F(\bar{x}) \wedge \mathrm{Dom}_G(\bar{y}) \Rightarrow F(\bar{x}) \leq F(\bar{y})$ (if $F \trianglelefteq G$)
- $\forall x[s] : \mathrm{InSort}_s(x) \Rightarrow : (\forall y[s] : y < x \Rightarrow \mathrm{InSort}_s(y))$.

5 Conclusion

In this paper, we put our Knowledge Based System, IDP, to the test in a real life car rental application from the business rules domain. We argued that an implementation in a KBS is more useful than the traditional approach taken by business rule systems since modelling it in a KBS allows for reuse of knowledge and specifications in various inferences. With this in mind, we modelled the rule-based approach in our own system and uncovered yet another drawback of this approach. The determinism that is implicit in such a rule-based approach doesn't always allow the system to optimise profit; it encodes a heuristic to deterministically pick a quite good solution, and excludes other, possibly better solutions. For this reason we modelled a declarative description of the EU-Rent application, and explained how to solve different tasks in this domain using the various forms of inference a KBS offers. Doing this, we noticed that some small tasks did not fit well in this approach. These lacks were no fundamental faults of the KBS paradigm, but rather a lack of expressivity of the currently used version of FO(·). Therefore, we described a new logic, an extra extension of FO(·) which filled the gap that was left in our modelling of a car rental system.

References

De Pooter, S., Wittocx, J., and Denecker, M. 2011. A prototype of a knowledge-based programming environment. In *International Conference on Applications of Declarative Programming and Knowledge Management*.

Denecker, M. 2000. Extending classical logic with inductive definitions. In *CL*, J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, Eds. LNCS, vol. 1861. Springer, 703–717.

Denecker, M. and Ternovska, E. 2008. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic (TOCL) 9,* 2 (Apr.), 14:1–14:52.

Denecker, M. and Vennekens, J. 2008. Building a knowledge base system for an integration of logic programming and classical logic. In *ICLP*, M. García de la Banda and E. Pontelli, Eds. LNCS, vol. 5366. Springer, 71–76.

Group, B. R. 2000. Defining Business Rules ˜ What Are They Really? Tech. rep.

Halle, B. V. 2001. *Business Rules Applied: Building Better Systems Using the Business Rules Approach*. John Wiley & Sons, Inc., New York, NY, USA.

Jackson, D. 2002. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM'02) 11,* 2, 256–290.

Leuschel, M. and Butler, M. J. 2008. ProB: An automated analysis toolset for the B method. *STTT 10,* 2, 185–203.

Lin, F. and Reiter, R. 1997. How to progress a database. *Artif. Intell. 92,* 1-2, 131–167.

Van Gelder, A., Ross, K. A., and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM 38,* 3, 620–650.

Vlaeminck, H., Vennekens, J., and Denecker, M. 2009. A logical framework for configuration software. In *PPDP*, A. Porto and F. J. López-Fraguas, Eds. ACM, 141–148.