Making Inconsistency Respectable: Part 2 - Meta-level handling of inconsistency

Dov Gabbay and Anthony Hunter

Department of Computing, Imperial College

180 Queen's Gate, London SW7 2BZ, UK

Abstract

Inconsistency in a database, when viewed purely logically, seem undesirable. Indeed the traditional approach to dealing with inconsistency in data is to employ means to restore consistency immediately. However, it is important to study the larger environment containing such databases, and the circumstances surrounding the inconsistency. We argue that within the larger environment, an inconsistency can be desirable, and useful, if we know appropriate actions to handle it. In some cases we may wish to remove the inconsistency, and in other cases we may wish to keep it. Moreover, we claim that inconsistencies only become meaningful when considered in the context of the larger environment, and in particular, of how they arise and are handled. In this paper we present a meta-level system that uses actions for handling inconsistent databases.

1 Introduction

In Part 1 (Gabbay and Hunter 1991), we presented the view that inconsistencies are not necessarily "bad", and that they can even be a useful as long as we can handle them appropriately. The traditional view of inconsistency is that it is local and relative to a database. However an inconsistency may have different meaning relative to the larger environment in which the database is used and with which it interacts.

We argue that dealing with inconsistencies is not necessarily done by restoring consistency but by supplying rules telling one how to act when the inconsistency arises. To illustrate our approach consider an airline booking system. It is normal practice for an airline to sell more tickets for a flight than there are seats on the flight. Even though this situation is inconsistent with the safety regulations, the airline will maintain the inconsistency until shortly before departure. The airline supports the inconsistency because it expects sufficient passengers not to show up at the airport, and therefore for the inconsistency to resolve itself. Furthermore, by maintaining the 'overbooking inconsistency' the airline can make more money. This is therefore an example of there being a cost-benefit in maintaining an inconsistency, and indeed of the inconsistency being desirable for the airline.

However, supporting this kind of inconsistency sometimes leads to difficult situations when, prior to departure, more people have checked in than expected. In this eventuality the airline staff are required to take some kind of action such as upgrading tourist class passengers to business class, or offering free tickets to passengers who are prepared to catch a later flight. In extreme situations they may even provide an extra aircraft. But despite the expense of some of these actions, they are rarely invoked, and therefore the 'overbooking inconsistency' is cost-effective overall.Furthermore, we are so used to the wider context of the airline inconsistency that some people might not even recognize it as an inconsistency.

Inconsistency handling in the booking system is an example of a general phenomenon found in database applications. Viewing the environment containing such databases, and the circumstances surrounding each inconsistency, indicates that we need to consider inconsistencies in terms of how and why they arise, and the actions that are performed on them. Indeed describing an inconsistency via the wider context of the database allows us to move away from the negative view of an inconsistency within a database. There are many other cases of database systems that can be described in a similar way to the booking system. For example, in a government tax database, inconsistencies in a taxpayers records are used to invoke inquiries into that taxpayer. Indeed from the perspective of the tax inspector, this is another application where inconsistencies are useful and desirable.

The problem we attempt to address here is the formalization of cases such as the booking system. Below, we consider some general requirements for the underlying languages. We provide a definition of a system for inconsistency handling, called the DA system, and we consider some of its positive and negative features.

2 Outline of a system for handling inconsistent data

To formalize inconsistency handling, we need to consider the object-language, and the action language. For example, for the booking system, we need an object-language for the declarative information about passengers, and flights, and we need an action language for representing the inconsistency handling undertaken. In effect, the action language connects the object-level inconsistencies to the larger environment in which the database operates.

For our action language, we need to be able to talk about the object-level inconsistency, and to be able to act on the inconsistency, either by invoking internal actions or external actions. Some of the key requirements can be summarized as follows: (1) Meta-language representation of object-level data and databases; (2) Facility to axiomatize object-level consequence relations; (3) State-based meta-languages for reasoning about the states of the data and databases; and (4) Separation of the object-level and meta-level semantics so that object level inconsistency does not force the meta-level to be inconsistent.

One possibility for our state-based meta-language would be some form of linear-time temporal logic, where we would allow quantification over formulae, and so specifiy how an object-level database should evolve over time. If a temporal logic specification is consistent, it is satisfiable by a class of models. So for example, for the temporal logic specification $\alpha \rightarrow \beta$, the class of models that can satisfy the specification can be defined by stating that for each $t \in \mathbb{N}$, in each model M on the linear structure $(\mathbb{N}, \geq)$, if $M \models \alpha$ at t, then $M \models \beta$ at t. In this way, we can view the meta-level handling of inconsistent object-level databases in terms of satisfying temporal logic specifications.

For our object-level we want to be able to reason with the same data using different consequence relations. For example, it may be desirable to switch from classical logic to paraconsistent, or relevance, or even non-monotonic logics. We also want to label object- level formulae to support handling. Such labelling may facilitate truth maintenance, or conflict resolution. To address these requirements of the object-level, we use the framework of the Labelled Deductive System (Gabbay 1991, 1993). In the Labelled Deductive System (LDS) languages are based on using labelled formulae, and defining proof rules in terms of both the formulae and the labels. In this way logical reasoning is naturally extended. A wide variety of non-standard logics have been considered in terms of LDS, including linear, resource, modal,

and paraconsistent logics. The label can also be used to define new kinds of logic that are appropriate for certain applications. For example the family of prioritized logics are defined in terms of preferences expressed over labels, and can be used for non-monotonic reasoning (Hunter 1992). Another example, is the family of restricted access logics for inconsistent information, where data is labelled, and for certain combinations of data, access to proof rules is restricted, thereby avoiding trivialization from inconsistent data (Gabbay and Hunter 1993).

3 Syntax for the DA system

Below we provide a definition of a meta-level system, called the DA system, (the Data and Action system). We assume DA can be used as a meta-language for a variety of object-level languages including labelled languages. Object-level formulae are used as terms in the meta-level. This allows the meta-level to "talk" about object-level data. In this way we are connecting the object-level data to the meta-language via the naming of formulae as terms. The meta-level also has other terms that allow it to talk about things other than the object-level data.

The system is based on first-order linear temporal logic, with since and until. The rules of formation for the meta-level terms (ie the terms for DA) are defined as follows. Note that the language DA is not typed, and therefore the language makes no distinction between terms that are object-level formulae, and the other terms:

If X is an object-level variable then X is a meta-level variable

If X is a meta-level variable then X is a meta-level term

If s is an object-level logical, predicate or function symbol
then s is a meta-level function symbol

If f is a meta-level function, and $t_1, \ldots, t_n$ are meta-level terms
then $f(t_1, \ldots, t_n)$ is a meta-level term

The rules of formation for the meta-level formulae (ie the formulae of the DA system) are defined as usual from the sets of meta-level terms, meta-level predicate symbols, logical symbols and the temporal operators { LAST, NEXT, FUTURE, PAST, ALWAYS, . . }. For example, let Holds be a meta-level predicate symbol, let p, q be object-level predicate symbols, and let X, Y be variables, then the following is a formula,

$\forall$X,Y ((Holds(p(X)) $\wedge$ Holds(q(Y)) $\rightarrow$ Holds(p(X) $\wedge$ q(Y)))

The proof theory is just the usual proof theory for first order linear temporal logic, with since and until, and includes first order classical proof theory. We do not present the axiomatization here, but but below we do provide a definition for the semantics. We need not commit ourselves further on the significance of time in this meta-language. However, we can use it for representing the real-time evolution of the database, or for the stepwise progress of actions on the database.

For some object-languages, the meta-language can be used to present an axiomatization of the consequence relations for the object-level. For example, we can consider an information system as a pair (Δ, Γ) where Γ is a an object-level database and Δ is a meta-level database that contains rules for acting on inconsistency in Γ. In this way, Holds(α) is a consequence of

Δ if and only if α is a consequence of Γ. Though in general, it is not always possible to provide a decidable axiomatization for the meta-level relation Holds, since some of the object-level languages we may be interested in are not decidable.

We have not discussed here issues of differentiating different sorts of terms. For example, if p(X) is an object-level formula, then it is likely that for purposes of inconsistency handling, we would wish to prohibit p(p(X)) as a term - since allowing such instatiations can lead to problems such as the liar paradox. Restrictions on such kinds of terms can be captured by adding appropriate axioms to the meta-level database. For more detailed discussion of issues pertaining to logic meta-languages, the reader is referred to Bowen (1982), Hill (1988), and Barringer (1991).

4 Semantics for the DA system

For this meta-language, we separate the proof theory and semantics for the object- language from the proof theory and semantics for the meta-level. We base the interpretation of DA on the natural numbers $(\mathbb{N}, \geq)$ as the flow of time. An interpretation for the DA meta-language is a tuple $(D, \mathbb{N}, \geq, h)$ where D is a non-empty domain, and h is a truth-assignment function. For this definition, the following conditions hold: (1) The set D is the Herbrand Universe generated from the terms of the meta-langauge; (2) If f is an n-place meta-level function then f is assigned to the mapping from $D^n \longrightarrow D$ as defined by $(\sigma_1, \ldots, \sigma_n) \longrightarrow f(\sigma_1, \ldots, \sigma_n)$; (3) If t is a ground meta-level term then t is assigned an object in D. The truth assignment function h is defined as follows:

For each m-place meta-level predicate symbol P and each $n \in \mathbb{N}$,
then $h(n, P): \mathbb{N} \times D^m \longrightarrow \{0,1\}$

If $P(t_1, \ldots, t_n)$ is a ground meta-level atom and $n \in \mathbb{N}$
then $h(n, P(t_1, \ldots, t_n)) = h(n, P)(t_1, \ldots, t_n)$

An interpretation as defined above is a model of α iff for all $n \in \mathbb{N}$, $h(n, \alpha) = 1$. The truth assignment function h can be extended to any meta-level formulae α and β as follows:

$h(n, \alpha \wedge \beta) = 1$ iff $h(n, \alpha) = 1$ and $h(n, \beta) = 1$
$h(n, \alpha \rightarrow \beta) = 1$ iff $h(n, \alpha) = 0$ or $h(n, \beta) = 1$
$h(n, \neg\alpha) = 1$ iff $h(n, \alpha) = 0$
$h(n, U(\alpha, \beta)) = 1$ iff $\exists m(m > n$ and $h(m, \alpha) = 1)$ and $\forall k(n < k < m$ and $h(k, \beta) = 1)$
$h(n, S(\alpha, \beta)) = 1$ iff $\exists m(m < n$ and $h(m, \alpha) = 1)$ and $\forall k(m < k < n$ and $h(k, \beta) = 1)$
$h(n, \forall X.\alpha) = 1$ iff $\forall \beta \in D$, $h(n, \alpha[\beta/X]) = 1$

Note that the above definitions imply a rigid interpretation of variables. In other words, the binding of a variable is fixed over time. Below we define the semantics for some extra temporal operators that are definable using the US operators.

$h(n, \text{NEXT } \alpha) = 1$ iff $h(n + 1, \alpha) = 1$
$h(n, \text{ALWAYS } \alpha) = 1$ iff for all $i \in \mathbb{N}$, $h(i, \alpha) = 1$

For these semantics we have just considered the language for the meta-level. The truth, or falsity, of a formula at the object-level does not necessarily affect the truth, or falsity, of formula at the meta-level. For example, an object-level database with the object-level formula House(red) false does not necessarily force the meta-level formula Holds(House(red)) also to

be false. (Though appropriate axioms could be added to the meta-level database to form a direct connection between the object-level and the meta-level.)

Separating the semantics for the meta-level from the semantics of the object-level gives us increased flexibility in handling uncertain and inconsistent object-level data. For example, if we view our database from the meta-level, then as we can update or ammend our object-level database, and if that object-level database becomes inconsisitent it does not necessarily cause the meta-level database to become inconsistent. We discuss this further below. However, it is staightforward to write specifications in the meta-language that can act on any inconsistencies in the object-level. Furthermore, these specifications can be written so that if they are not met, then the meta-level also becomes inconsistent.

The definition of the Holds predicate does mean that all the object-level data and consequences are reflected upwards. In other words, via the Holds predicate, all the object-level data and consequences are represented at the meta-level, and so changing the object-level database will cause a change in the meta-level database. In contrast, the reflection downwards depends on the DA specification in the meta-level database. The actions specified by the DA axioms could directly affect the object-level - for example by truth maintenance - and hence constitute reflection downwards, or they may influence the outside world - and so not directly affect the object-level database.

5 Executing DA Specifications

In this section we illustrate how we can execute DA specifications, and in the next section we return to our case study.

The traditional view on temporal logics is of declarative statements about the world, or about possible worlds over time. These relate the truth of propositions in the past, in the present and in the future. An alternative view is to consider the logics in terms of a declarative past, and an imperative future, based on the intuition that a statement about the future can be imperative, initiating steps of action to ensure it becoming true.

More specifically, if we write DA specifications in the following form, where the antecedent refers to the past and present, and the consequent refers to the future, then we can execute such specifications so as to construct a model of the specification,

$$\text{ALWAYS}(\ \wedge_i \alpha_i \rightarrow \vee_j \beta_j)$$

Suppose then that we have a specification Δ in the form of a finite conjunction of such clauses such that each α_i and β_j is either a positive or negative literal. The executing agent tries to execute Δ in such a way as to build a model of Δ. It must make Δ true dynamically at each point in time. So at any time point, it will consider each such clause. If $\wedge_i \alpha_i$ is true then it must make the disjunction $\vee_j \beta_j$ true. The choosing of which β_j to make true (remember that β_j is a future formula) is a subtle (and not necessarily decidable) problem, and the agent will take into account several factors, such as commitment to make other clauses true, possible future deadlocks, and the environment at the time. Executable temporal logics that have been developed include USF (Gabbay 1989), MetateM (Barringer 1989), and MML (Barringer 1991).

6 Case Study: Airline booking system

In the introduction, we described an airline booking system that uses certain forms of inconsistency advantageously. Below we axiomatize interesting aspects of this system in the meta-language. For the object-level we assume the paraconsistent logic C_ω (da Costa 1974), though we will use an unlabelled version to ease exposition.

For the formalization of this problem we require the following object-level predicates, where P is the set of passengers, F is the flight, D is the date, and C is the set of 'checked- in' passsengers: Passengers(P, F, D); Overbooked(F, D); Checked-in(C, F, D), Commercial(F); and Legal(F). We also require the following object-level functions: size(P) which returns the number of elements in P; and capacity(F) returns the maximum number of passengers for the flight F. The following two axioms capture conditions under which Overbooked, or its negation, hold. The first holds when the number of passengers exceeds the capacity of the flight. The second axiom captures the specification that if the flight is a commercial flight, and that the flight is legal, then the relation Overbooked(F, D) is false.

$$\text{Passengers}(P, F, D) \wedge \text{Checked-in}(C, F, D) \wedge \text{size}(C) = X \wedge \text{capacity}(F) = Y \wedge (X > Y) \rightarrow \text{Overbooked}(F, D)$$

$$\text{Commercial}(F) \wedge \text{Legal}(F) \rightarrow \neg\text{Overbooked}(F, D)$$

If the 'overbooked inconsistency' occurs prior to departure time, no action is specified. However, if it occurs at departure time, then below are specified three courses of action. The first is to upgrade the unplaced passengers. If that fails, then the second option is to offer bonus tickets and a later flight. Finally, if both the previous options fail, then arrange alternative travel. For this we require the following object-level relation, where P is the set of passengers, F is the flight, D is the date, and X is the set of passengers without seats: Unplaced-passengers(X, F, D), and the following meta-level relations, Offer-upgrade(X, F, D), Offer-bonus-tickets(X, F, D), and Arrange-alternative-travel(X, F, D).

$$\text{ALWAYS}(\forall X, F, D\ (\text{Holds}(\text{Overbooked}(F, D)) \wedge \text{Holds}(\neg\text{Overbooked}(F, D)) \wedge \text{Departure-time} \wedge \text{Holds}(\text{Unplaced-passengers}(X, F, D)) \rightarrow \text{NEXT}(\text{Offer-upgrade}(X, F, D))))$$

$$\text{ALWAYS}(\forall X, F, D\ \text{Holds}(\text{Overbooked}(F, D)) \wedge \text{Holds}(\neg\text{Overbooked}(F, D)) \wedge \text{LAST Departure-time} \wedge \text{Holds}(\text{Unplaced-passengers}(X, F, D)) \rightarrow \text{NEXT}(\text{Offer-bonus-tickets}(X, F, D))))$$

$$\text{ALWAYS}(\forall X, F, D\ \text{Holds}(\text{Overbooked}(F, D)) \wedge \text{Holds}(\neg\text{Overbooked}(F, D)) \wedge \text{LAST}(\text{LAST}(\text{Departure-time})) \wedge \text{Holds}(\text{Unplaced-passengers}(X, F, D)) \rightarrow \text{NEXT}(\text{Arrange-alternative-travel}(X, F, D))))$$

In this example, we can see how the object-level inconsistency is reflected upwards, via the Holds predicate, and how the meta-level actions are suggestions for the user to solve the inconsistency. The way this specification would be executed is that at departure time, if there is the inconsistency, then the antecedent of the first of these rules would hold, and the system would be forced to satisfy the specification by making the Offer-upgrade hold in the meta-level database. This would cause the Offer-upgrade suggestion to be made available to the user. If at the first time-point after this suggestion had been made, the inconsistency still held, then the antecedent of the second rule in the specification would hold, and this second rule would be executed. Finally, if at the second time-point after the departure time,

the inconsistency still held, then the third rule would be executed. Finally, if at the third time-point the inconsistency still held, then the meta-language would have no further actions to handle the inconsistency. Note, in none of these rules is there a reflection downwards.

Obviously, we require a series of further axioms to fully describe the booking system. However, we have illustrated how the inconsistent object-level does not cause the meta- level to be inconsistent, and how appropriate external actions can be formalized.

7 Discussion

In this paper, we have illustrated how we can handle inconsistent databases in a formal way. We use a meta-language to specify how we act on an inconsistency, and this leads to a new perspective on inconsistency handling. We now need to use the formal language to further characterize the nature of inconsistency handling. In particular, we wish to identify axiom schemas that capture common features of inconsistency handling.

Since the DA system uses temporal logic, it is based on a well-developed theoretical basis. It is straightforward to show that the meta-level of the DA system inherits desirable properties of first-order US temporal logic such as a complete and sound proof theory, and of semi-decidability. Furthermore for some useful subsets of US temporal logic there are viable model building algorithms, such that if the meta-level specification is consistent then the algorithm is guaranteed to find a model of the specification (Barringer 1989).

Using this approach to handling uncertain and inconsistent data constitutes a fundamental move away from traditional views of database management. From this perspective of the DA language, we don't worry, per se, about the object-level databases. All we worry about is satisfying the specification for the meta-level language. In this way, we give up a requirement to make the object-level database consistent, and rather accept such situations as inevitable. We abstract away from the object-level, and shift the requirement of consistency to the level of the meta-level being consistent.

There have been a number of other approaches to addressing issues of inconsistency in data. There are the paraconsistent logics (for example da Costa 1974, Anderson and Belnap 1975), but these only localize inconsistency - they don't offer strategies for acting on inconsistency. In contrast, many approaches force consistency on data without consideration of the environment. Truth maintenance systems (de Kleer 1978, Doyle 1979), and belief revision theory (Gardenfors 1988) ensure consistency by rejecting formulae upon finding inconsistency. Similarly, Fagin et al (1983) proposed amending the database when finding inconsistency during updating. Even more restrictive is the use of integrity constraints in databases - which prohibit inconsistent data even entering the database.

However, recently, attempts have been made to accommodate inconsistent data in a database by taking account of the environment. For example, Balzer (1991) suggests "guards" on inconsistent data to minimize the negative ramifications, and then to warn the user of the inconsistency, and in Naqui and Rossi (1990) inconsistent data is allowed to enter the database, but the time that the data is entered is recorded, and newer the data takes precedence over the older data when resolving inconsistencies. We see our approach as generalizing these approaches. Though, of course, many of the details have not yet been addressed in our approach.

8 Acknowledgements

This work is currently being funded by UK SERC grant GR/G 29861, and by the CEC ESPRIT DRUMS 2 project. The first author is a SERC senior research fellow.

9 References

Anderson A and Belnap N (1975) Entailment, Princeton University Press
Balzer R (1991) Tolerating inconsistnecy, in proceedings of the 13th International Conference on Software Engineering, IEEE Press
Barringer H, Fisher M, Gabbay D, Gough G and Owens R (1989) MetateM: A framework for programming in temporal logic, in REX Workshop on Stepwise Refinement of Distributed Systems, LNCS 430, Springer Verlag
Barringer H, Fisher M, Gabbay D, and Hunter A (1991) Meta-reasoning in executable temporal logic, in Principles of Knowledge and Reasoning: Proceedings of the Second International Conference (KR91), Morgan Kaufmann
Bowen K and Kowalski R (1982) Amalgamating language and meta-language, in Clark K and Tarnlund S, Logic Programming, Academic Press
da Costa N C (1974) On the theory of inconsistent formal systems, Notre Dame Journal of Formal Logic, 15, 497-510
Doyle J (1979) A truth maintenance system, Artificial Intelligence, 12, 231 - 297
Fagin R, Ullman J and Vardi M (1983) On the semantics of updates in databases, in Proceedings of the Second Annual Association of Computing Machinery Symposium on Principles of Database Systems
Gabbay D (1989) Declarative past and imperative future: Executable temporal logic for intereactive systems, in Banieqbal B, Barringer H and Pneuli A, Proceedings of Colloquium on Temporal Logic in Specification, Lecture Notes in Computer Science, 398, Springer
Gabbay D (1991) Labelled deductive systems, Technical report, Centrum fur Informations und Sprachverarbeitung, Universitat Munchen
Gabbay D (1993) Labelled deductive systems: A position paper, in Proceedings of Logic Colloquium '90, Lecture Notes in Logic 1, Springer Verlag
Gabbay D and Hunter A (1991) Making inconsistency respectable: Part I, in Proceedings of Fundamentals of Artificial Intelligence Research '91, LNCS 535, Springer Verlag
Gabbay D and Hunter A (1993) Restricted access logics for inconsistent information, in Proceedings ESQARU'93, LNCS, Springer
Gardenfors P (1988) Knowledge in Flux, MIT Press
Hill P and Lloyd J (1988) Analysis of meta-programs, in Proceedings of the Workshop on Meta-programming in Logic Programming, University of Bristol
Hunter A (1992) A conceptualization of preferences in non-monotonic proof theory, in Pearce D and Wagner G, Logics in AI, Lecture Notes in Artificial Intelligence 633, Springer
de Kleer J (1978) An assumption-based TMS, Artificial Intelligence, 28, 127 - 162
Naqvi S and Rossi F (1990) Reasoning in inconsistent databases, in Debray S and Hermenegildo M, Logic Programming: Proceedings of the North American Conference, MIT Press