UDC: 007.5
Original scientific paper

KNOWLEDGE BASE AMALGAMATION USING THE HIGHER-ORDER LOGIC-BASED LANGUAGE HiLog

Alen Lovrenčić

University of Zagreb
Faculty of Organization and Informatics, Varaždin, Croatia
e-mail: alovrenc@foi.hr

The increasing number of data sources that are used for decision support in business meant it was necessary to integrate all sources and their unique representation to a user. The sources we may want to integrate can be internal (local databases, etc.) or external (Internet, etc.). Each one of these sources may have a different query language, with a different syntax, different semantics and querying power. Because of this different kinds of problems may arise. The most common problems are inconsistency, partiality of data, and conflicts between sources. We need a language powerful enough to solve those problems. In this paper I will show the way to solve these problems using a logic-based language HiLog as a mediator language. Different strategies to solve these problems will be presented, as well as HiLog solutions for those strategies.

Keywords: integration, knowledge bases, logic programming.

1. INTRODUCTION

1.1. History of database theory

For more than 25 years databases have been extensively studied. Databases are large structures of data usually stored in the secondary memory, and well-organized and easy to query and easy to use.

In the early 90s the database field changed dramatically. New database models have been introduced, for example the object-oriented database model, the deductive database model etc. Logic-oriented languages are being used more and more, especially in theoretical research, as database query languages. There are many special deduction strategies that have been conformed for database querying. Global and local networking has been introduced into database theory, and that has goals that may be reached in distributed database research. Uncertainty has been introduced into database theory, so the theory has become more realistic.

One of the main areas of database research is the integrating of different databases, be they local or accessible through WAN, in a unique database model. The first model for database integration that is well known today and that has already been implemented is called a data warehouse. This data warehouse is a way of integrating more databases that can be accessed, and produce synthetic data form the analytical data those databases contain. To represent synthetic data, a data cube model is invented, and for searching unknown relationships between data, data mining was introduced. But, the data warehouse model had some drawbacks. First of all when building a data warehouse it is necessary that all of the databases that have to be integrated are relational, that the data models for them are completely known, and that the integrator has unrestricted access to all data in the underlying databases. Second, an even greater disadvantage of data warehouses is that the warehouse database is filled periodically. Therefore, data in the warehouse is historical, not on-line. Because of this, and because of the long time period needed for filling operations, the data warehouse concept is useless for application domains with a large amount of data that can change quickly.

Because of the disadvantages with the data warehouse concept, a new concept has been invented. This concept is with an on-line interpretation of the underlying databases, with a language translation between these underlying database query languages and the query language of the whole system. The new concept is called a *knowledge base amalgamation.*

1.2. Integrated knowledge-base system

The concept of knowledge bases is also a new concept in database theory and it is connected to the usage of logic-based query languages. As we well know, databases are used for storing great amounts of data. In the knowledge base, knowledge is stored in the same way as data. The term knowledge as it is used in artificial intelligence, is different from the philosophical meaning of the term. In the sense of knowledge bases, knowledge is complex data. Knowledge represents implicit data, that is, knowledge is not data that is presented to the user, but it is part of algorithms, formulas, etc. that are used to compose new data that is not explicitly stored in the knowledge base. Knowledge bases are connected to logic-based query languages because logic programs are natural representations of knowledge bases, where the facts represent data, and clauses represent knowledge. So, with this reasoning it can be said that the terms *knowledge base* and *deductive database* are synonyms.

In Figure 1 we can see the graphical representation of an integrated database system with 3 underlying sources. Notice that we do not use the term database for any underlying sources. It is because those sources do not have to be databases in the classical sense. It is very often the case that underlying sources are legacy sources – data files, or Internet legacy sources – HTML files, or a data structure, such as a Prolog program etc.

The main parts of this system are the mediator and the wrappers. In this system we have one mediator and three wrappers. Wrappers are modules that are, basically, translators of the underlying source query languages for the mediator query language and vice versa. But, they are more than that. The query languages of the underlying sources can be different in their syntax, and in their querying power, too. For example, the legacy sources have less querying capabilities than in relational databases, and classical relational databases have less querying capabilities than deductive databases. Secondly, an even more important goal of these wrappers is to equalize querying capabilities of the underlying sources and the mediator. There are different techniques for that, and they are explained in [3].

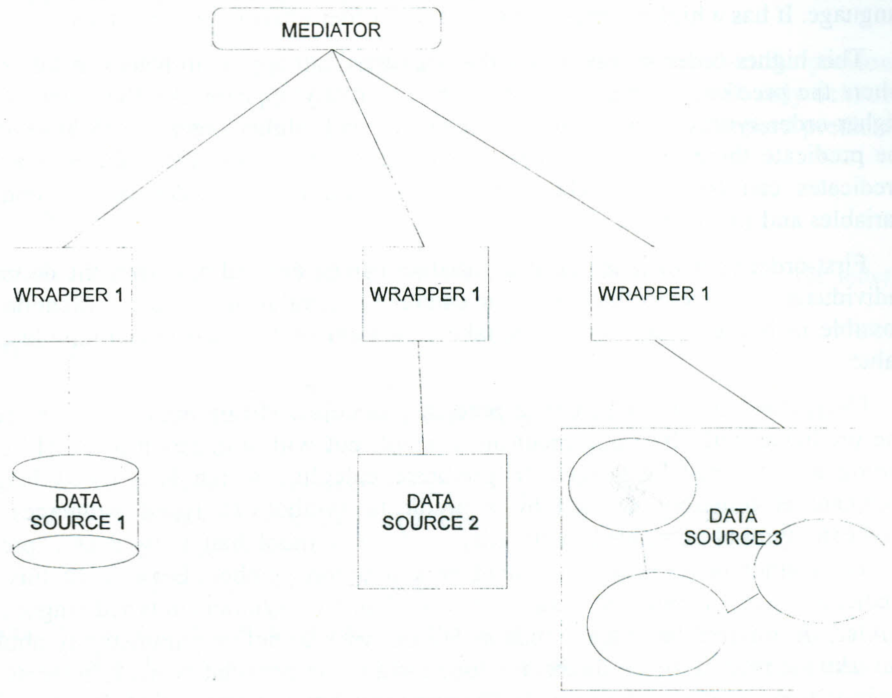

Figure 1. A simple integrated system

The mediator is the module that integrates information given by the sources, and is translated by the wrappers, into one coherent knowledge base. The mediator is the module that resolves inconsistencies and conflicts that are produced by integrating data from different, independent sources.

In this paper we shall refer to a source and its wrapper as a single unit. So, all sources in this paper take queries and give answers in the same language, i.e. the language of the mediator. We shall not study the wrapper construction, or techniques to improve the query capabilities of sources. The part of the integrated system that we shall concentrate on is the mediator, and the strategies to resolve inconsistencies and conflicts between sources. We shall introduce a new query language for the mediator that is based on a syntactically high-order logic language HiLog. HiLog is a Prolog-type language with higher-order syntax and first-order semantics. It is much more clean syntactically than Prolog, because many non-logical elements in Prolog have been introduced that are in conflict with the first-order semantics of Prolog. The deductive procedure involved in HiLog is an improved resolution procedure, the so called SLG-resolution. The SLG-resolution is a tabled variant of the SLDNF-resolution that is implemented in Prolog. The SLG-resolution is also known as an OLDT-resolution. It is interesting that the SLG-resolution is even more appropriate for database querying than bottom-up deduction that is specially invented for logical programming in databases. So, the bottom-up procedure is safe for so-called *stratified programs*, and the SLG-resolution is safe for larger class of programs, the class of *partially stratified programs*. Lastly, let us say that HiLog is, like Prolog, an untyped language. It has a higher order syntax and first-order intensional semantics.

This higher-order syntax means that variables can appear in places in the formula where the predicate or the function symbol normally appears. Another way to define higher-order syntax is to say that in a language with higher-order syntax arguments of the predicate there can be other predicates. In a first-order syntax the arguments of predicates can be terms, which are, in the case of first-order syntax, constants, variables and functions.

First-order semantics means that variables can be defined only over the domains of individuals. This means that a variable can take as a value any object, but cannot, as is possible in higher-order semantics, take a relation or function over individuals as a value.

Untypeness means that, unlike predicate calculus, HiLog approves of more than one predicate with the same predicate symbol, but with a different arity. HiLog and Prolog are untyped languages. In predicate calculus, which is a typed language, predicate is uniquely defined by a predicate symbol. In typed languages every predicate symbol is related to its arity, and the symbol that is used as a predicate symbol cannot be used as a constant or a function symbol. Because of this every predicate symbol is mapped to some relation. Function symbols in typed languages are similar. In untyped languages, such as HiLog, we can define parameter symbols that can take the role of the predicate, the function and the constant symbol, in terms of the position they appear in the formula.The same symbol can also define one relation and one function of any possible arity. This affects the way the semantic structure is defined. It has to be defined as an infinite tuple of functions and relations for every parameter symbol.

The intensionality of semantics is another important property that separates HiLog from predicate calculus. When you are in the world of first-order, intensionality or extensionality doesn't mean much. An extensional language, like predicate calculus, assigns proper relations to every predicate symbol, and a proper function to every functional symbol. Intensional languages, like HiLog, assign intension or meaning to every parameter symbol. Then, a semantic structure is defined as the relation between intension and proper extension, i.e. relation or function. To assure extensibility of a language it is necessary for intensions and extensions to be undistinguished. The way to do that is to introduce extensional axioms to logic. This is also related to equality theory embodied in logic. Extensional equality says that two predicate symbols are equal if they both represent the same relation. Intensional equality, on the other hand, says that two predicate symbols are not equal, regardless of the relations they represent, until they are explicitly equated. That means that in the intensional equality theory two predicates p and q can represent the same relation, and atom $p=q$ can still be false. It is well known that strong, extensional equality can make a language undecidable. Strong extensional equality embodied in predicate calculus is the reason why the predicate calculus is not decidable. On the other hand, HiLog, with its higher order syntax, but when it only has trivial intensional equality embodied, is decidable.

2. THE LOGIC-BASED LANGUAGE HILOG

The main characteristics of the language of HiLog are mentioned in the previous section. In this section those properties are going to be formalized. It will only be those definitions of the language given, that are different in HiLog and in standard predicate calculus.

2.1. The syntax of HiLog

First we have to define the alphabet, i. e. the set of symbols from which the letters for creating the words of the language are taken.

Definition 1: (The alphabet)
The *alphabet of HiLog* contains:
1. An infinite set of parameter symbols $\mathcal{S}$,
2. An infinite set of variables $\mathcal{V}$,
3. A set of logical connectives $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$,
4. A set of quantifiers $\{\forall, \exists\}$,
5. Parentheses **)**, **(**.

Sets $\mathcal{S}$ and $\mathcal{V}$ have to be disjoint.

There is one significant difference in defining the alphabet here from that of predicate calculus. When the alphabet of predicate calculus is defined it is necessary to separate the constant, function and predicate symbols. That is because predicate calculus is a typed language. On the other hand, as we said in the previous section, HiLog is a non-typed language. That means that some symbol regarding semantic context may represent a constant, a function of any arity, or a relation of any arity. The meaning of a symbol is defined by context in which it appears in the formula. Because of this, in the definition of the HiLog alphabet there is no request that appears in the definition of the alphabet of predicate calculus, that sets of constants, function symbols and predicate symbols have to be disjoint. We only have demand for the disjointness of a set of variables and a set of parameter symbols.

What is to be defined are terms, i.e. the words of the language, the well-formed formulas, and the sentences of the language.

Definition 2: 1. Every parameter symbol and variable is a term
2. If $t, t_1,..., t_n$ are terms, then $t(t, t_1,..., t_n)$ is also a term
3. Terms are only those expressions that can be generated by using rules 1 and 2 of this definition a finite number of times.

Definition 3: The *atomic formula* or *atom* is every HiLog term.

Now we will show the standard definition for the formulas:

Definition 4: 1. Every atom is *well-formed formula (wff)*.
2. If **F** is wff then $(\neg\mathbf{F})$ is also wff.
3. If **F** and **G** are wffs and $\Theta\in\{\wedge, \vee, \rightarrow, \leftrightarrow\}$ then $\mathbf{F}\Theta\mathbf{G}$ is also wff.
4. If **F** is wff, $\mathbf{K}\in\{\forall, \exists\}$ and $\nu\in\mathcal{V}$ variable in **F**, then $(\mathbf{K}\nu\mathbf{F})$ is also wff.

5. Wffs are only those expressions that can be created by using rules 1-4 a finite number of times.

These definitions show another significant difference between HiLog and predicate calculus. While in predicate calculus we distinguish between terms and formulas, in HiLog every term is also a formula. That means that there are expressions like p, $p(p,p)$, $p(g(a),p(a,b))$ and so on which are wffs of HiLog but not wffs in predicate calculus. The first expression is not a predicate calculus formula but it is a predicate calculus term. For same reason the first expression is a Prolog term, but it is not a Prolog formula. The second expression is a HiLog and a Prolog formula, but it is not a predicate calculus formula, because the parameter symbol p appears in the expression and has two different functions – the predicate symbol of arity 2 and the constant symbol. The third expression is the same, where p appears as a predicate symbol of arity 2 and a function symbol of arity 2.This definition is the one that makes the syntax of HiLog to be of a higher-order.

All other syntax definitions – the definition of a closed formula, the conjunctive and disjunctive normal form formulas etc. – are the same formally as they are in predicate calculus. But they create different classes of objects because they are built on a different set of formulas.

2.2. The semantics of HiLog

There are many more differences between HiLog and predicate calculus in semantics than there are when you look at their syntax.

In predicate calculus, a semantic structure is defined as a pair $<U,I>$ where U is the domain, and I is an *interpretation*, a function that is defined through two mappings:

1. $\mathcal{F}$ – a mapping that for every n-ary, $n>0$ function symbol defines the function $\mathbf{U^n \rightarrow U}$,
2. $\mathcal{P}$–a mapping that for every n-ary, $n\geq 0$ predicate symbol defines the relation on $\mathbf{U^n}$

In this case constants can be treated as 0-ary function symbols, so $\mathcal{F}$ can be used to interpret constants too. Now, for a predicate symbol p $I(\mathrm{p})=\mathcal{P}(p)$, and for a function symbol or a constant symbol f $I(f)=\mathcal{F}(f)$. In the case of predicate calculus the alphabet is defined in such a way that sets of predicate symbols and function symbols are disjoint.

As we said before, a semantic structure in HiLog is defined in different ways, in order to make the language intensional.

Definition 5: The *semantic structure* of HiLog is a quadruple $\mathbf{M}=<U, U_{true}, I, \mathcal{F}>$ where:

1. U is a nonempty set of intensions for the domain of $\mathbf{M}$,
2. $U_{true} \subseteq U$ set specifies which of the elements from U are intensions of the true propositions,
3. $I:\mathcal{S} \rightarrow U$ is a function that associates an intension with every logical symbol.

4. $\mathcal{F}: U \to \prod_{k=1}^{\infty} [U^k \to U]$ is a function, such that for every $u \in U$

   and $k > 0$ kth projection of $\mathcal{F}(u)$, which is also denoted as $u_{\mathcal{F}}^{(k)}$. and is a function $[U^k \to U]$.

In this definition U has a similar function as it has in the semantic structure of predicate calculus. The difference is that in predicate calculus U was the domain of interpretation and contains extensions for constant symbols. In HiLog U is a set of intensions, for the set $\mathcal{S}$ of parameter symbols. Next, since each parameter symbol can be constant, we need to know which of them are true in the interpretation **M**. A subset of U, U_{true} is introduced because of this and it contains true propositions. In predicate calculus we didn't have to introduce this set. In predicate calculus all the terms do not have a truth-value.

We have I. This I is not the same I introduced in predicate calculus semantic structure. In the HiLog semantic structure I is an intensional function that associates an element from the domain U to each parameter symbol. In predicate calculus I was an interpretation, that had a function similar to set U_{true} in the HiLog semantic structure.

The next element of the HiLog semantic structure is $\mathcal{F}$. Remember that in predicate calculus $\mathcal{F}$ was a mapping from a set of function symbols to the set of functions such that for each n-ary function symbol $f \in \mathcal{L}$ $\mathcal{F}(f)$ is n-ary function from U to U. So, in predicate calculus it is $\mathcal{F}: \mathcal{L} \to [U^k \to U]$. Because HiLog is an untyped language, the arity of the symbol is not defined in the alphabet. Each symbol can have every possible arity. Since each arity symbol represents a different function. So, it is not enough to associate a single function to each symbol. We have to associate with a symbol one function for each arity. That means that we have to associate an infinite tuple of functions to each parameter symbol from $\mathcal{S}$. So we define $\mathcal{F}$ as it is defined in definition 5. We didn't use the set $\mathcal{S}$ here because it is better to use U for defining functions and I for mapping from $\mathcal{S}$ to U.

In order to finish a description of the semantic structure in HiLog, we have to see how set $\mathcal{P}$, that assigns an infinite tuple of relations to every parameter symbol, disappears from the HiLog semantic structure. In HiLog, in the same way that we did for $\mathcal{F}$, we have to associate the parameter symbol with the infinite tuple of relations, for every parameter symbol can represent a predicate symbol of any possible arity.

It is necessary to define how we can interpret HiLog regular formulas like $X(a)$ where X is a variable. In predicate calculus it was a defined variable assignment $v: \mathcal{V} \to U$. However, $v(X)$ is an element of the semantic domain $\mathcal{S}$. So in order to interpret $X(a)$ it is necessary to, with respect to v, associate functions and relations to elements of U, not to elements of $\mathcal{L}$. As was said before, we can use function I to associate elements of U to parameter symbols. To interpret the formula mentioned above, it is necessary to extend the interpretation to variables. In this way we introduce variable assignment to the set $\mathcal{T}$ of all terms, as follows:

- $v(s) = I(s)$ for every parameter symbol from $\mathcal{S}$
- $v(t(t_1, \ldots, t_n)) = \mathcal{F}(v(t))(v(t_1), \ldots, v(t_n))$

There are two different ways to define the truth of a formula $t(t_1,\ldots,t_n)$ where t is treated as a predicate symbol. One way is to define truth with respect to a n-ary relation that is associated with $v(t)$ by $\mathcal{P}$. In this case we have:

$$v(t(t_1,\ldots,t_n))=\mathcal{P}(v(t))(v(t_1),\ldots,v(t_n)).$$

Another way to interpret the above formula is with respect to 0-ary relation which is associated with $v(t(t_1,\ldots,t_n))$ by $\mathcal{P}$. In this case we have

$$v(t(t_1,\ldots,t_n))=\mathcal{P}(v(t(t_1,\ldots,t_n))).$$

Both of those ways give the same semantics. For a more uniform treatment, we will use a second alternative.

It is obvious that if we use a second alternative, there is no need for $\mathcal{P}$ to be explicitly defined in the semantic structure. All we need is to highlight the true propositions from U, and store them in a new set U_{true}. Now, the structure defined in definition 5 is explained in detail.

When a semantic structure is defined, the next step is to define when the structure **M** is a model for the formula F. It is obvious what this means that a grounded formula is satisfied in some structure **M**: A grounded formula F is satisfied in the structure **M**, and is denoted by $\mathbf{M}\models F$, iff $F\in U_{true}$.

Satisfaction for a nongrounded formula is defined in the same way as for predicate calculus, using the definition of satisfaction for grounded formulas and the variable assignment for **M**.

2.3. HiLog as a programming language

To use logic as a programming language it is necessary to restrict the syntax and semantics of the language to *clauses*.

The definition of a *clause*, *Horn clause*, *definite clause*, *program*, *goal,* and *query* in HiLog is formally the same as it was in predicate calculus.

We need to define a Herbrand universe and the Herbrand base of HiLog in order to achieve the procedural semantics of HiLog and the semantics of amalgamated HiLog that are going to be developed next.

Definition 6: The *Herbrand universe* $\mathcal{HU}$ of language $\mathcal{L}$ of HiLog is a set of all grounded terms of language $\mathcal{L}$.

This is a well known definition used in every logic-based language. The universe of language is defined with respect to the definition of the terms in the language. The next definition is also the same for all formal languages – the definition of the Herbrand base.

Definition 7: The *Herbrand base* $\mathcal{HB}$ is the set of all grounded atoms of a language.

Because of definitions 2 and 3, for HiLog, definitions 6 and 7 imply the next corollary:

Corollary 1: $\mathcal{HU}=\mathcal{HB}$.

The proof of this corollary is obvious.

The corollary 1 is a property that is often appears in languages with a higher-order syntax. The next definition gives us the fundamental term of logic programming, the Herbrand interpretation. It is well known that some formula is satisfiable iff there is a Herbrand interpretation where the formula is true. So, to prove that the formula is satisfiable, we only have to find a Herbrand interpretation which is a model for the formula. On the other hand, if there is no Herbrand interpretation that is a model for the formula, it can be concluded that the formula is antitauthology.

Definition 8: The *Herbrand interpretation* $\mathcal{HJ}$ of HiLog is every subset of the Herbrand base.

It is one of the many ways you can define the Herbrand interpretation. This definition MEANS that $\mathcal{HJ}(\alpha)=1$ iff $\alpha\in\mathcal{HJ}$.

What is needed next is to define the strategy of deduction. There are two basic strategies for deducing facts. A *top-down* strategy called resolution, and a *bottom-up* evaluation. The second one was invented for deductive databases, in order to resolve the drawbacks of a resolution: the one-at-time strategy of giving answers and infinite loops that often appear. But, the bottom-up evaluation has a drawback of its own – it works properly only with stratified programs. The best we can do here is to implement a tabled resolution (SLG-resolution), that is safe with respect to infinite loops, and that works properly for a larger class of programs – the so called partially stratified programs. The SLG-resolution will not be explained here. Any readers who are interested in SLG-resolution can find detailed information in [6].

3. STRATEGIES FOR SOLVING INCONSISTENCY IN THE AMALGAMATION OF INDEPENDENT SOURCES

As we said in section 1, those sources that we want to integrate are independent. They can be a database on the local machine or on the LAN or WAN, knowledge bases, legacy sources, programs, web pages, sensors and so on. The underlying model of these sources may be unknown. We are only aware of the view that is presented by the source designer. It is also important to mention that there is no way of changing the design of either the source or the view. This means that we can only assume a read priority to sources. This concept enables us to define the integrating system which is independent of the definitions of sources. It is not necessary to change the syntax or the semantics of the sources to integrate them into an integrated system.

So all we can do is to receive data from a source as it is and interpret them with a wrapper in the way we need them and integrate them in the unique and consistent knowledge presented to the user.

But the independence of the data sources can cause some problems. The first problem is the incompleteness of the data. The data source can contain only a part of the attributes we need for some object we are dealing with. Some other sources can contain other parts of information for the object. This problem is partially solved by a wrapper, and partially by a mediator. The wrapper has to interpret this partial information, and compose that information in a record that is used for that object by the mediator. This means that the wrapper adds nulls to the unknown data from the sources. The mediator uses information derived by the wrapper, and decides how to treat it and how to integrate it with information about the object taken from other sources. Information received from different sources can be inconsistent. This means that different sources may retrieve contrary data for each instance of an object. This inconsistency has to be resolved before information can be presented to the user. There are two basic strategies for resolving inconsistency:

- *Naïve strategy* – Present all the received data to the user, whether they are inconsistent or not.
- *Pessimistic strategy* – Remove all the contradictory data from the answer, and present to the user only those answers about which a consensus of all the data sources was made.

It is obvious that both of these strategies have advantages and drawbacks. The naïve strategy will give more answers to the user and the user can then make more competent decisions with respect to those answers. But, that strategy is not good if the system has to make a decision on its own. It is not good if the system has to be reliable, too. In this case we have to use a pessimistic strategy. The pessimistic strategy will present to the user only those answers that are 100% true according to the underlying sources. On the other hand, by using a pessimistic strategy some interesting answers can be "cut off".

Sometimes it is hard to decide which strategy to use. Usually it is not convenient to use the same strategy for all objects. Even more, sometimes it is a fact that some sources are more reliable for some types of information than for others. But, by using this pessimistic strategy, an answer will be "cut off" even if the most unreliable source disagree with it. Therefore, there is one, more complex strategy we can introduce – the *strategy of source hierarchy*. This strategy introduces the reliability of sources with regard to some information. Reliability can be expressed in the terms of the probability that a source will have reliable information about the object. Using this strategy it is possible to present the probability for some answer to the user, or to present only those answers that have a high probability. The problem with this strategy is that a great amount of programming is needed to define all probabilities. If there is n sources and m relevant objects in a system, to define reliability for each source for each object it is necessary to define $m \cdot n$ probabilities. But it is possible that, for example, source s_1 has a reliability of 0.7, source s_2 reliability 0.3 and s_3 0.2, but if sources s_2 and s_3 agree about this information, then it will have a reliability of 0.8. But if the system has this nonlinear property, then to define all probabilities, it is necessary to define $2^n \cdot m$ probabilities, and that could be a problem for a larger number of data sources.

Therefore, we shall introduce a flexible strategy that will be used in amalgamated systems. Firstly, we will define a default method to deal with inconsistency. For the default strategy we will use a pessimistic strategy. This strategy is implemented in the amalgamation management system to deal with inconsistency when it is not otherwise defined. But, for the implementation of some other strategy a logic-based language is used in the mediator. The advantages of this strategy are that there is no need to define every possible combination of source reliability, and that it is more flexible then any other strategy that has been described before, because this strategy allows one to implement any other strategy that has been explained.

4. FORMAL THEORY OF AMALGAMATION USING HILOG

First we have to define an extension of the standard HiLog to make it more appropriate for programming an amalgamated system. We will define a logic-based language called *amalgamated* HiLog, which is, as is shown in [2], only a syntactic extension, that can be implemented in pure HiLog. Let there be n local data sources. Every data source i can be formally represented by the local database, that is represented by the local HiLog program BP_i. So, our goal is to integrate local programs $BP_1, \dots, BP_n$ into one consistent program. To deal with these different programs it is necessary to define a *mediator database* or *mediator program* that contains the knowledge necessary for dealing with heterogeneous sources. If we label the local sources with numbers $1, \dots, n$ and the mediator system with s then we will have our next definition:

Definition 9: The *alphabet* of *BP*-language $\mathcal{L}_{BP}$ is created from:
- $\{1, 2, \dots, n, s\}$
- an infinite set of *BP*-variables

Definition 10: A *BP-term* is a nonempty finite subset $\mathcal{L}_{BP}$.

The theory of amalgamation can be developed for any logic-based language. The language that is the foundation for building an amalgamated language is called a *base language*. In our context, the base language for amalgamation is HiLog. Our next definition defines the atom of an amalgamated language.

Definition 11: Let D be a *BP*-term and let α be a HiLog atom. Then $\alpha{:}[D]$ is called an *amalgamated atom*.

The amalgamated formula is defined in the usual way, as it was defined in definition 2. The difference is that for amalgamated formulas we use amalgamated atoms.

So, as we have seen, the syntax of amalgamation is very simple. There are amalgamations added to atoms of the base language. The formulas of an amalgamated language are built in the same way as the formulas of a base language are generated. To make this language useful, we need to define the semantics of this language.

The truth of an amalgamated atom is defined with respect to the local programs BP_i. An amalgamated atom of form $\alpha{:}[i]$ is true iff an atom α is true in a local program BP_i. That is obvious. But, how can we define the truth of an amalgamated atom $\alpha{:}[D]$ where is D, $|D|>1$? That depends on the strategy we use. If we use a pessimistic strategy, then $\alpha{:}D$ is true if $\alpha{:}[i]$ is true for all $i \in D$.

If, on the other hand, we use a naïve strategy, then $\alpha{:}[D]$ is true if there is at least one $i \in D$ such $\alpha{:}[i]$ is true.

Now we know how to interpret almost all the amalgamated formulas.

Now we can formalize them. For the moment we shall exclude the symbol s from the *BP*-language. That symbol will be treated separately later.

Definition 12: Let $\mathbf{M}_i$ be a semantic structure for local databases BP_i, $i=1,\dots, n$. Then semantic structure for the language $\mathcal{L}_{Amal}$ is n-tuple $\mathbf{M}=<\mathbf{M}_1,\dots,\mathbf{M}_n>$.

Now, we can define an amalgamated interpretation as

Definition 13: 1. For $i\in\{1,\dots,n\}$ $\mathbf{M}\models\alpha:[i]$ iff $\mathbf{M}_i\models\alpha$

2. For $D\in\{1,\dots,n\}$ $\mathbf{M}\models\alpha:[D]$ iff for all $i\in D$ is $\mathbf{M}_i\models\alpha$.

We will now have the semantics for pure amalgamation, based on the pessimistic strategy. But, we want to define another program that contains the amalgamated formulas, and represents a way of amalgamating local programs. We shall call that program a *mediator* or *supervisor*. In the mediator we can define way to amalgamate the sources that differ from default strategy, in our case the pessimistic strategy.

Because of the mediator in definition 9 we introduce the symbol s. Generally, that symbol is like any other amalgamation symbol. But, it is not necessary to examine all the amalgamated formulas with this symbol. We shall define a restricted class of formula that is sufficient to represent all these defined in the mediator.

At first, we can exclude all those atoms that have an amalgamation that contains s or any other symbols. Namely, we shall define the axiom for that as:

$$\alpha:([s]\cup D) = \alpha:[s].$$

That means that we shall examine only those formulas that have atoms whose amalgamation is the form set $2^{\{1,\dots,n\}}\cup\{s\}$.

Now, it is normal in all logic-based programming languages to restrict the class of formulas that a program can contain to clauses. So, local databases will contain HiLog clauses, and the mediator will contain amalgamated clauses, i.e. clauses made from amalgamated atoms.

We will now introduce the so-called *amalgamation axiom scheme*. The local program BP_i defines the atoms with an amalgamation i, and the mediator defines the atoms with an amalgamation s. Now we have to define the way to resolve the problem of truth for atoms that have an amalgamation that contains more than one symbol. In order to simplify the programming of an amalgamated system, we need to define the set of intristic axioms, that is built into the system, and make a base for amalgamated deduction. These are axioms that define the pessimistic strategy.

Definition 14: The *amalgamation axiom scheme* is a set of clauses of the form $B:[D]\leftarrow \bigwedge_{\emptyset\subset D'\subset D} B:[D']$ where $D\subseteq\{1,\dots,n\}$ or $D=[s]$.

By introducing the axiom scheme from definition 14 into the system, we can symplify the mediator program because the programmer only has to write those clauses that differ from the chosen pessimistic strategy. With these built-in axioms we obviate the writing of approximately of 2^n clauses.

But, we shall restrict the class of queries that the system can process. As we said before, we will presume that the user does not know the structure of an amalgamated system. So, the only queries that the user can ask are the queries to the mediator, because the mediator is an interface between the user and the sources. So, queries that the users ask will consist of atoms whose amalgamation will be $[s]$. In this case, the user could write queries in pure HiLog, without any amalgamations, and then before processing a query, amalgamations can be added to all the atoms in a query. On the one hand, that obviates the user from writing amalgamations in their query, but on the other hand this restricts those clauses necessary to be defined in the mediator only to those that have the head with amalgamation $[s]$. Those kind of clauses are called *s-amalgamated clauses*. They are introduced in the next definition:

Definition 15: The *S-amalgamated clause* is the clause with the head of the form $\alpha{:}[s]$.

We can make a demand such that all the clauses in the mediator should be *s*-amalgamated clauses now.

Now we shall make the connections between the local programs BP_i, written in pure HiLog, and the mediator program written in amalgamated HiLog.

Definition 16: Let H be a HiLog program that represents the local database BP_i. Let C be a HiLog clause, $C \in H$. Let C have the form

$$\alpha{:-}\alpha_1, \ldots, \alpha_n, \text{not } \alpha_{n+1}, \ldots, \text{not } \alpha_{n+m}.$$

Then we can define an *amalgamated transformation* AT(C) as the amalgamated clause

$$\alpha{:}[i]{:-}\alpha_1{:}[i], \ldots, \alpha_n{:}[i], \text{not } \alpha_{n+1}{:}[i], \ldots, \text{not } \alpha_{n+m}{:}[i].$$

We also define an *amalgamated transformation* of the HiLog program H as $\mathrm{AT}(H)=\{\mathrm{AT}(C) : C \in H\}$

We can now define a *local* for interpretation given in the local database.

Definition 17: Let $\mathbf{M}^{loc}$ be a local semantic structure of the local database BP_i. This means that the semantic structure defines truth in the local database. Then the *local* of $\mathbf{M}^{loc}$ is the set of amalgamated structures $\{\mathbf{M}$: for $\mathbf{M}$ and for every ground atom α we have $\mathbf{M} \models \alpha{:}[i]$ iff $\mathbf{M}^{loc} \models \alpha\}$.

This definition gives the semantics of the amalgamated transformation. Now it is interesting to see if this transformation preserves the truth of atoms, and formulas. That means that we want to see that clause c is true in H iff AT(c) is true in AT(H). We will now take two theorems from [2].

Theorem 1: Let BP_i be a HiLog program. If $\mathbf{M}^{loc} \models BP_i$ then for all $\mathbf{M}$ from the local of $\mathbf{M}^{loc}$ with respect to BP_i satisfies $\mathbf{M} \models \mathrm{AT}(BP_i)$.

Theorem 2: Let BP_i be a HiLog program and $\mathbf{M}^{loc}$ be the local structure such that there exists structure **M** in the local of $\mathbf{M}^{\mathbf{loc}}$ with respect to BP_i that is a model for $\mathbf{AT(BP_i)}$. Then $\mathbf{M}^{\mathbf{loc}}$ is a model for $\mathbf{BP_i}$.

Proofs of these two theorems are given in [2] but with small differences in notation. These two theorems are very important in amalgamation theory because they show us that an amalgam that is going to be defined preserves the semantics of all the local databases.

Definition 18: Let $BP_1,\dots,BP_n$ be local databases that are written as HiLog programs. Let S be the mediator written in amalgamated HiLog. Then we define *amalgam* as

$$S \cup \bigcup_{i=1}^{n} AT(BP_i) \cup amalgamation\ axioms$$

An amalgam represents all clauses that are defined in the amalgamated system in one way or another. In other words, an amalgam represents all the clauses that we use for deduction in an amalgamated system.

5. CONCLUSION

Throughout this paper we have introduced a language for the amalgamation of heterogeneous independent data sources. A logic-based language called amalgamated HiLog was proposed as a language that is appropriate for defining an amalgamated system and the relations in it. It was shown that this language is correct and that the semantics of this language preserves the semantics of the local databases.

Let us sum up a way to define clauses and queries in a system. Local sources are wrapped by their wrappers, so we can assume that local sources are HiLog programs. Local sources are connected to a mediator, a program written in amalgamated HiLog. The connection between the right interpretation of those components is assured by theorems 1 and 2. After this there is another transformation between amalgamated and pure HiLog. A user writes his queries in pure HiLog, and for every atom in his amalgamated version with amalgamation, the [*s*] query is automatically created. So we use an amalgamated transformation to transform a query, and use theorems 1 and 2 to assure that we get the right transformation.

REFERENCES

[1] S. Adali and V.S. Subrahmanian. Intelligent caching in heterogeneous reasoning and mediator systems. *Proceedings of Second International Conference on Building and Sharing of Very Large-Scale Knowledge Bases*. IOS Press, Twente, The Netherlands, 1995, pp. 247-256.

[2] W. Chen, M. Kifer and D.S. Warren. HiLog: a foundation for higher-order logic programming, *Journal of Logic Programming*. Vol. 15, No. 3, 1993, pp. 187-230.

[3] A. Lovrenčić. *Amalgamacija baza znanja*. M.Sc. thesis, Faculty of Organization and Informatics, Varaždin, 1999.

[4] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina and J. D. Ullman. A query translation scheme for rapid implementation of wrappers. *Proceedings of the Conference on Deductive and Object-Oriented Databases,* 1995, pp. 247-256.

[5] V. S. Subrahmanian. Amalgamating knowledge bases. *ACM Transactions on Database Systems*. Vol. 19, No. 3, 1994, pp. 291-331.

[6] T. Swift T, W. Chen and D. S. Warren. Operational Semantics for SLG Evaluation. Part 1: Modularly Stratified Programs, Draft, technical paper, manuscript, SUNY, Stony Brooks, 1994.

Received: 15 October 1999
Accepted: 10 December 1999

Alen Lovrenčić

AMALGAMACIJA BAZA ZNANJA KORIŠTENJEM LOGIČKOG JEZIKA VIŠEG REDA

Sažetak

Povećanjem broja izvora podataka koji se koriste pri odlučivanju dolazi do potrebe za integracijom svih izvora, čime se ostvaruje jedinstvena reprezentacija svih podataka. Izvori podataka koji se integriraju mogu biti interni (lokalne baze podataka, datoteke itd.) i eksterni (Internet , itd.). Svaki od izvora može imati različit upitni jezik. Upitni jezici izvora mogu imati različitu sintaksu i semantiku, ali i ekspresivnost. Ove razlike mogu uzrokovati različite probleme. Najčešći problemi koji se javljaju jesu nekonzistencija podataka s raznih izvora, parcijalnost podataka i konflikti među izvorima. Stoga je potrebno definirati jezik koji će biti dovoljno moćan da razriješi te i druge probleme koji mogu nastati pri amalgamaciji izvora znanja. U ovom se članku pokazuje način rješavanja gore navedenih problema korištenjem logičkog jezika HiLog kao jezika medijatora sustava. Također, u radu se obrađuju različite strategije rješavanja problema amalgamacije heterogenih izvora znanja te implementacija tih strategija u HiLogu.

Ključne riječi: integracija, baze znanja, logičko programiranje.