ELSEVIER

Mapping visual to textual knowledge representation

Catholijn M. Jonker[a], Rob Kremer[b], Pim van Leeuwen[a], Dong Pan[b], Jan Treur[a],*

[a]*Department of Artificial Intelligence, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands*
[b]*Software Engineering Research Network, University of Calgary, 2500 University Drive NW, Calgary, Alberta, Canada, T2N 1N4*

Abstract

In this paper, graphical representations for knowledge structures in the design method DESIRE for component-based design of knowledge and multi-agent systems are presented, together with a graphical editor based on the Constraint Graph environment. Moreover, a translator is described which translates these graphical representations to textual representations in DESIRE. The strength of the combined environment is a powerful—yet easy-to-use—framework to support the development of knowledge-based and multi-agent systems. Finally, a mapping is presented from DESIRE to Conceptual Graphs. This provides a unifying perspective on the knowledge representation format of DESIRE and allows the use of conceptual graph tools to specify and maintain the knowledge representation part of a DESIRE specification.
© 2005 Elsevier B.V. All rights reserved.

1. Introduction

Most languages for knowledge acquisition, elicitation, and reasoning result in specifications in pure text format. Textual representation is easier for a computer program to process. However, textual representation is not an easily understandable form, especially for those domain experts who are not familiar with computer programming. Visual representation of knowledge relies on graphics rather than text. Visual representations are more understandable and transparent than textual representations [9].

DESIRE (DEsign and Specification of Interacting REasoning components) is a design method used for component-based design of knowledge and multi-agent systems (see [4] for the underlying principles and [3] for a case study). DESIRE supports designers during the entire design process: from knowledge acquisition to automated prototype generation. DESIRE uses composition of processes and of knowledge to enhance transparency of the system and the knowledge used therein.

Originally, a textual knowledge representation language was used in DESIRE that is based on order sorted predicate logic and some extensions. In particular constructs were introduced for the following:

- Structuring in (hierarchical) components of information structure specifications and of knowledge specifications
- Explicit naming of information structure specification components and knowledge base specification components
- Complex object structures as terms, based on functions.
- The use of meta-descriptions of information structures within (other) information structure specifications and knowledge specifications
- The use of logical relationships (if-then implications) in knowledge specification

Subsequently, a graphical representation method for knowledge structures has been developed, inspired by a number of known graphical formalisms such as semantic networks, terminological formalisms and the conceptual graph formalism [2,7,10,11,12,14]. Finally, starting with [8], the relationship of the graphical representations to conceptual graph formalism [10,11,12,14] was investigated in more depth. One of the outcomes of the current paper is that a translation to the conceptual graph formalism can be made that provides a suitable unifying perspective on the different notations, and makes it possible to use conceptual graph tools to specify and maintain the knowledge

* Corresponding author. Tel.: +20 598 7763; fax: +20 598 7653.

E-mail addresses: jonker@nici.ru.nl (C.M. Jonker), kremer@cpsc.ucalgary.ca (R. Kremer), pand@cpsc.ucalgary.ca (D. Pan), treur@few.vu.nl (J. Treur).

URLs: http://www.nici.ru.nl/~catholj (C.M. Jonker), http://www.cs.vu.nl/~treur

representation part of a DESIRE specification. In some cases, the graphical notation of conceptual graphs has been adapted a bit to obtain a more dedicated graphical format for the needs as discussed above.

Constraint Graphs [6] is a concept mapping 'meta-language' that allows the visual definition of any number of target concept mapping languages. Once a target language is defined (for example, the DESIRE's graphical representation language) the constraint graphs program can emulate a graphical editor for the language as though it were custom build for the target language. This 'custom' graphical editor can prevent the user from making syntactically illegal constructs and dynamically constraints the choices of the user to those allowed by the syntax. Constraint Graph's graphical environment is used to present knowledge in a way that corresponds closely to the graphical representation language for knowledge that is used in DESIRE. A translator is described that bridges the gap between the graphical representation and the textual representation language in DESIRE.

2. Graphical knowledge representation in DESIRE

In this section, both graphical and textual representations and their relations are presented for the specification of knowledge structures in DESIRE [4]. Knowledge structures in DESIRE consist of information types and knowledge bases. In Sections 2.1 and 2.2 graphical and textual representations of information types are discussed. In Section 2.3 representations of knowledge bases are discussed.

2.1. Basic concepts in information types

Information types provide the ontology for the languages used in components of the system, knowledge bases and information links between components. In information type specifications the following concepts are used: sorts, sub-sorts, objects, relations, functions, references, and meta-descriptions. For the graphical specification of information types, the icons in Fig. 1 are used.

A sort can be viewed as a representation of a part of the domain. The set of sorts categorizes the objects and terms of the domain into groups. All objects used in a specification have to be typed, i.e. assigned to a sort. Terms are either objects, variables, or function applications. Each term belongs to a certain sort. The specification of a function consists of a name and information regarding the sorts that
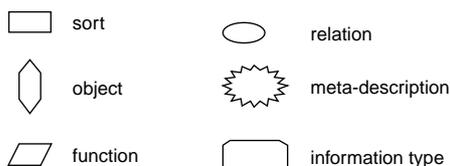
form the domain and the sort that forms the co-domain of the function. The function name in combination with instantiated function arguments forms a term. The term is of the sort that forms the co-domain of the function. Relations are the concepts needed to make statements. Relations are defined on a list of arguments that belong to certain sorts. If the list is empty, the relation is a nullary relation, also called a propositional atom. The information type birds is an example information type specifying sorts, objects, functions and atoms with which some knowledge concerning birds can be specified. The information type is specified in Fig. 2. With information type birds it is, for example, possible to express statements like 'Tweety is of the type that it prefers vegetarian food': is_of_type (tweety, food_preference(vegetarian)).

Note that being able to express a statement does not mean that the statement is true, it could be false.

2.2. Compositionality of information types

Compositionality of knowledge structures is important for the transparency and reusability of specifications. In DESIRE two features enable compositionality with respect to information types: information type references, and meta-descriptions. By means of information type references it is possible to import one (or more) information type(s) into another. For example, information type birds above can be used in an information type that specifies an extended language for specifying knowledge that compares birds.

Example 1

```
information type compare_birds
  information types birds;
    relations     same_type:           BIRD * BIRD;
end information type
```

A more complex example shows how a number of information types can be composed and extended in one new information type: in the information type flying objects the three information types are combined (Fig. 3). In Fig. 4, it is shown how in such a composed information type also additional concepts can be specified.

Textually, the information type flying objects is specified as follows:

Example 2

```
information type flying_objects
  information types birds, insects, machines;
  sorts
      WORLD_OBJECT;
  sub-sorts
      BIRD, INSECT, MACHINE: WORLD_OBJECT;
  objects
      ariane_5            : MACHINE;
      pingu               : BIRD;
      hindenburg          : WORLD_OBJECT;
```

sort		relation
object		meta-description
function		information type

Fig. 1. Information types: legenda.
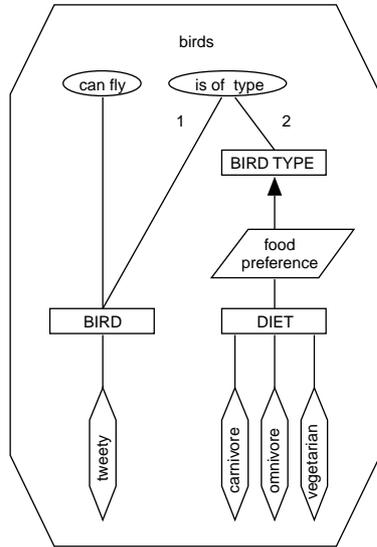
```
information type birds

sorts                          BIRD,

                               DIET,

                               BIRD_TYPE;

objects

    tweety :                   BIRD;

    carnivore,

    omnivore,

    vegetarian :               DIET;

functions

    food_preference :   DIET–> BIRD_TYPE;

relations

    can_fly:                   BIRD;

    is_of_type :      BIRD * BIRD_TYPE;

end information type
```

Fig. 2. Information type: birds.

```
relations
   zeppelin,
   balloon,
   flies          : WORLD_OBJECT;
end information type
```

The second feature supporting compositional design of information types is the *meta-description* representation facility. The value of distinguishing meta-level knowledge from object level knowledge is well recognized. For meta-level reasoning a meta-language needs to be specified. It is possible to specify information types that describe the meta-language of already existing languages. As an example, a meta-information type, called about birds, is constructed using a meta-description of the information type birds (Fig. 3). The meta-description of information type birds connected to sort BIRD ATOM ensures that every atom of information type birds is available as a term of sort BIRD ATOM. Using information type about birds it is possible to express that it has to be discovered whether bird Tweety can fly (to_be_discovered(can_fly (tweety))) (Fig. 5).

2.3. Knowledge bases

Knowledge bases express relationships between, for example, domain specific concepts. Reasoning processes use these relationships to derive explicit additional information. Knowledge elements represent statements about the domain that are considered true. A statement can be unconditional (a *general fact*) or conditional (a *rule*). Assume, as an example, that it fior the control of a chemical process it is a fact that the action to increase the incoming mass flow has to be performed. This fact is represented graphically in Fig. 6.

The same icons are used as were used to construct information types. As can be seen the function increase is

Fig. 3. Composition of flying objects by references.
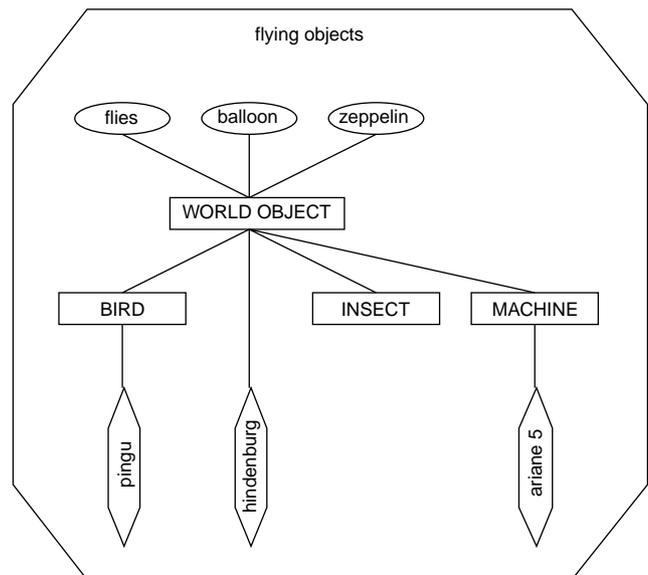

Fig. 4. Flying objects.
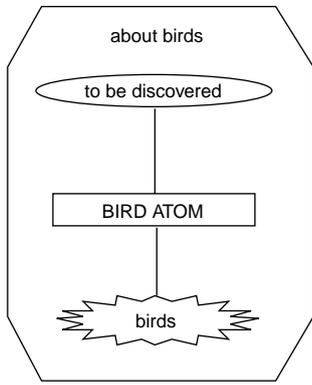
Fig. 5. **Meta-descriptions:** about birds.

applied to the object incoming mass. As explained before, a function combines its arguments to a term of its destination sort. This is made explicit by the arrow to an object, which is named by the term created by the function. The relation to be performed has one argument, in this case the term created by applying the function increase to the object incoming mass. The complete statement in concise form is: to_be_performed(increase (incoming_ mass)). Using this statement as a general fact means that the statement is considered to be true.

The statement to_be_performed(increase(incoming_ mass)). is not always true. For example, it is only true if it has been observed that the pressure is low and the temperature is
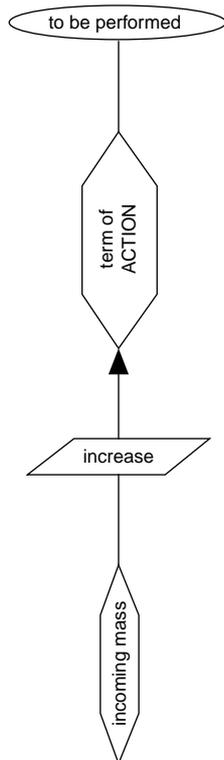
Fig. 6. General fact: the incoming mass flow has to be increased.
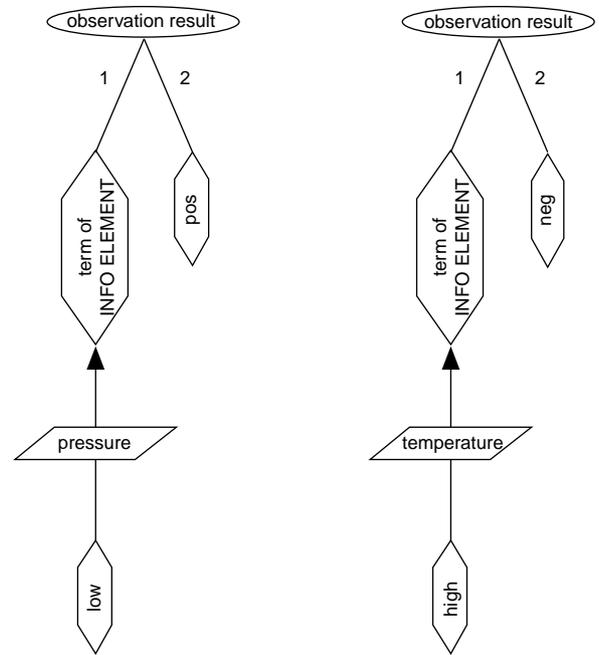
Fig. 7. Conditions.

not high. These two statements, called the conditions, are presented in Fig. 7. The same statements in concise form are

observation_result(pressure(low),pos) and obser-vation_ result(temperature (high),neg).

If these two conditions are true, then as a consequence the general fact depicted in Fig. 6 is also true. To graphically represent this implication, the two conditions are to be composed, the result of that composition (the *antecedent*) is the first argument of the implication, the *consequent* is the second. The true statement (rule) to be represented is of the form:

(condition one AND condition two) IMPLIES consequent

The logical connective used in this statement is the binary relation: IMPLIES. Two sorts are introduced: the sort
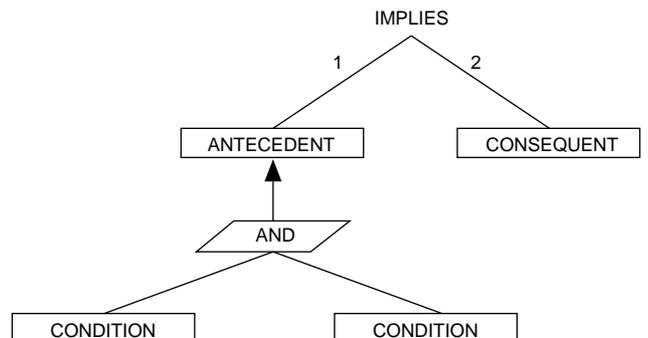
Fig. 8. Schema for implication.

ANTECEDENT and the sort CONSEQUENT. The first argument of the logical relation IMPLIES is of sort ANTECEDENT, the second of sort CONSEQUENT. The first argument is a complex term:

condition one AND condition two

This term contains the logical connective AND which can be expressed by introducing the function AND and the sort CONDITION. The function AND has two arguments of sort CONDITION which it combines into a term of sort ANTECEDENT. The graphical representation of the knowledge element (antecedent one AND antecedent two) IMPLIES consequent can be found in Fig. 8.

The entire implication for the consequent to_be_performed(increase(incoming_mass)) can be found in Fig. 9. In concise textual form the knowledge element in Fig. 9 is expressed as:
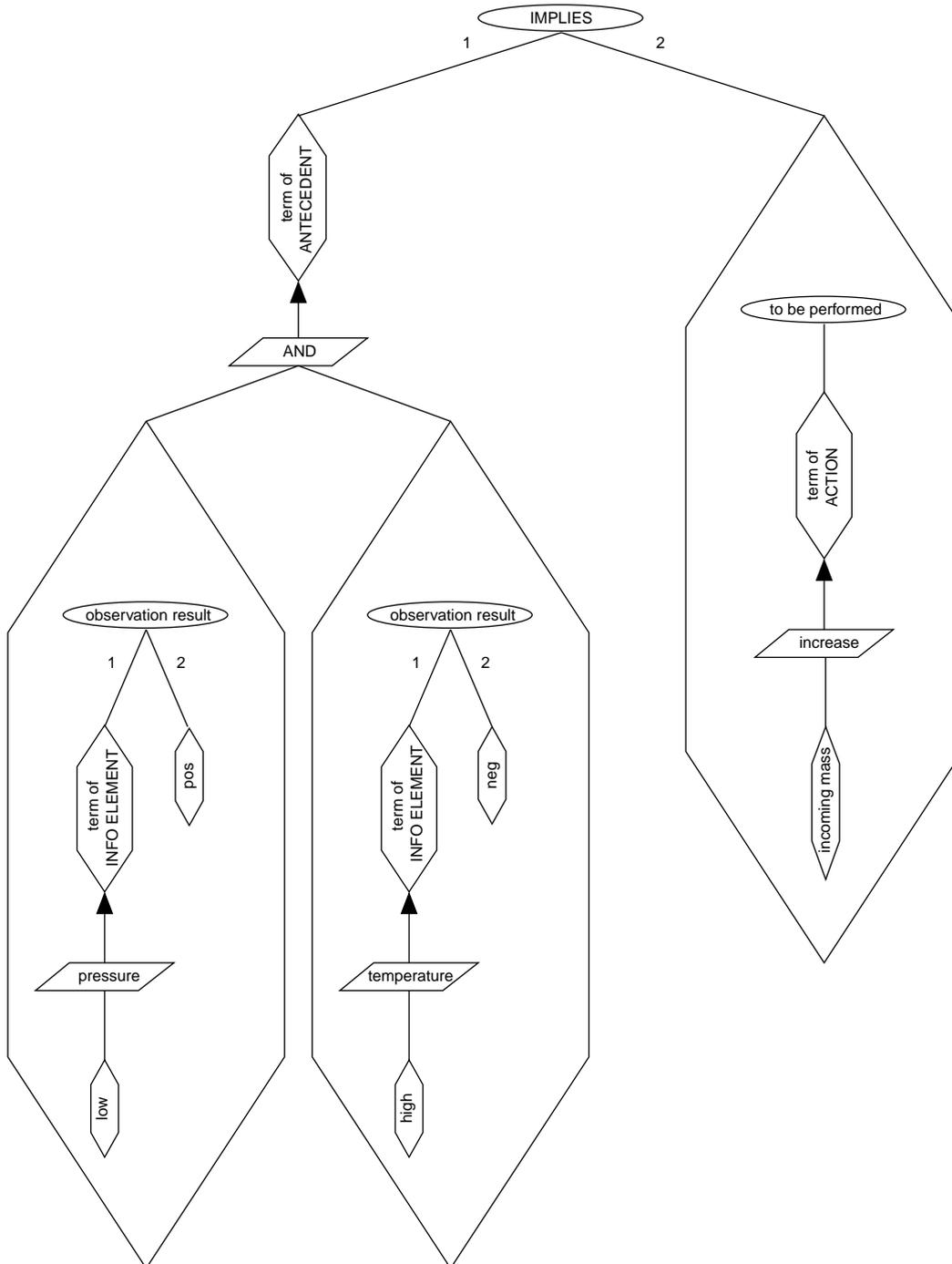
Fig. 9. Kowledge base element of mass control knowledge.

if observation_result(pressure(low),pos)

and observation_result(temperature(high),neg)

then to_be_performed(increase(incoming_mass))

Finally, a knowledge base can reference several other knowledge bases. The knowledge base elements of knowledge bases to which the specification refers are also used to deduce information (an example has been omitted). Notice that this graphical representation is comparable to the one that can be used in conceptual graphs. Representing more complex logical statements in graphical form is always not easy; for this problem no new perspectives were found.

3. Constraint graphs

Constraint graphs is a concept mapping 'meta-language' that allows one to visually define any number of target concept mapping languages. Once a target language is defined (for example, the DESIRE knowledge representation language) the constraint graphs program can emulate a graphical editor for the language as though it were custom build for the target language. This 'custom' graphical editor can prevent the user making synactically illegal constructs. Furthermore, the editor dynamically constraints user choices to those allowed by the syntax.

In order to accommodate a large number of visual languages, constraint graphs must make as few assumptions about concept mapping languages as possible. To this end, constraint graphs defines only four base components: node, arc, context, and isa (Fig. 10). Nodes and arcs are mutually exclusive, where nodes are the vertices from graph theory, and arcs interconnect other components, and are analogous to edges in graph theory. Both nodes and arcs may (or may not) be labeled, typed, and visual distinguished by color, shape, style, etc. Contexts are a sub-type of node and may contain a partition of the graph. Isa arcs are a sub-type of arc and are used by the system to define the sub-type relation: one defines one component to the be a sub-type of another component merely by drawing an isa arc from the sub-type to the supertype.

Futhermore, the generality requirement of constraint graphs dictates that arcs are not always binary, but may also be unary or of any arbitrary arity greater than 1 (i.e. trinary and *n*-ary arcs are allowed). For example, the
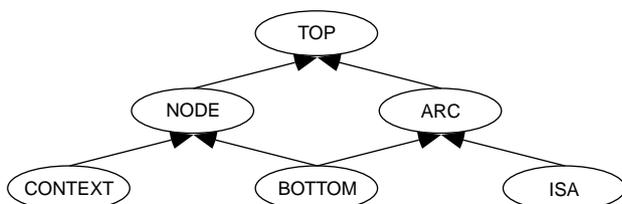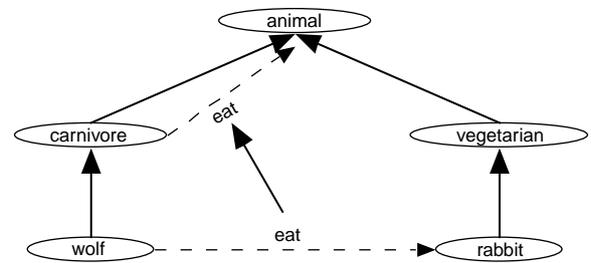

Fig. 11. An example Constraint Graphs definition.

between relation puts a trinary arc to good use. Constraint graphs arcs may interconnect not only nodes but other arcs as well. This is not only useful, but necessary because all sub-type and instance-of relations are defined using an isa arc, arcs between arcs are required to define the type of any arc. Finally, within constraint graphs no hard distinctions are made between types and instances, but rather, the object-delegation model [1] is followed where any object can function as a class or type.

To illustrate some of the above points, Fig. 11 shows a simple definition. Here, the fat, directed arcs are the constraint graphs isa arcs and define carnivore and vegetarian to be sub-types of animal, wolf as a sub-type (or instance-of) of carnivore, and rabbit as a sub-type (or instance-of) of vegetarian. Furthermore, the eat binary relation (dashed arc) is defined and starts on carnivore and terminates on animal. These terminals are important: the components at the terminals constrain all sub-types of eat to also terminate at some sub-type of carnivore and animal respectively. The second eat arc is defined (by the fat isa arc between it's label and the first eat arc's label) to be a sub-type of the first eat arc. It is therefore legally drawn between wolf and rabbit, but the editor would refuse to let it be drawn in the reverse direction: the eat definition says that rabbits can't eat wolves.

4. Implementation of the translator

In Constraint Graphs, three basic types of objects exist: nodes, arcs and contexts. The elements of the language to be expressed in the Constraint Graphs' environment therefore need to be mapped onto these basic types. Table 1 below shows the mapping between DESIRE's knowledge elements and nodes, arcs and contexts.

Constraint Graphs allows the user to further constrain the language definition in by, for example, restricting the shapes and connector types of the nodes and arcs the language elements are mapped onto. In our case, we restrict the shape of node Sort to a rectangle, and the shape of Object to a diamond. Furthermore, sub-sorts, meta-descriptions and relations will be represented as directed labeled arcs, where the label takes the shape of an ellipse. Moreover, functions will be depicted as directed labeled arcs as well, but the label

Fig. 10. The base type lattice for Constraint Graphs.

Table 1
Mapping between DESIRE and constraint graphs

Object	Sort	Subsort	Meta description	Function	Relation	Information type	Knowledge base
NODE	NODE	ARC	ARC	ARC	ARC	CONTEXT	CONTEXT

will be a parallelogram. Finally, information types and knowledge bases are mapped onto contexts, and the shape of these contexts will be the default: a rectangle.

Fig. 12 below gives an impression of a specification of the DESIRE information type birds (compare to Fig. 2) in Constraint Graphs.

Every mature engineering discipline has handbooks to describe successful solutions for known problems. There now exists a software design patterns literature, e.g. [5], describing successful solutions to common software problems. Industrial experience has proven that patterns are a valuable technique in software engineering problem-solving discipline. Not only do patterns capture successful experience, they also help improve communication among designers. They can help new developers avoid traps and pitfalls that traditionally can only learned by costly experience. This section uses patterns to detail some aspects of the translator implementation. The translator was implemented with Borland C++ under Windows NT. To make the program more portable, only ANSI C++ syntax is used. The implementation details that do not relate to design patterns are omitted here.

The intent of the Interpreter pattern is to represent the grammar of a language and interpret sentences in the language [5], pp. 243–255). The Interpret pattern represents each grammar rule as a class. Symbols on the right-hand side of grammar rule are instance variables of the class. The TerminalExpression implements an Interpret method associated with terminal symbol in the grammar. The NonterminalExpression implements the Interpret method for nonterminal symbol in the grammar. Typically the Interpret method of NonterminalExpression is implemented by calling the Interpret methods of its subexpressions. The Interpret method takes Context as an argument. The Context provides information global to the interpreter. What the Context should contain totally depends on what the Interpret method intends to do.

For the translator, the class hierarchy for the Interpreter pattern has a common abstract class Expression. Expression declares a pure virtual Interpret method, which will be inherited and implemented by all its concrete subclass. It has two direct subclasses: MapExpression and DesireExpression. These two classes are also abstract classes. They act as the base classes of Constraint Graph and DESIRE object hierarchies respectively. All Constraint Graph expression nodes are subclasses of MapExpression; all DESIRE expression nodes are sub-classes of DesireExpression.

For TerminalExpression, the implementation of the representing class is simple and straightforward. Besides the attributes and methods needed for normal functioning, it must implement the virtual Interpret method inherited from base class. The Interpret method will interpret the corresponding terminal symbol that the class represents. For example, DESIRE has a grammar rule defining variables:

<variable::= <variable_name ':' <sort_name

This is a terminal expression. This grammar rule was modeled as class DesireVariable shown in Listing 1.

Listing 1: Class Definition of DesireVariable

```
class DesireVariable: public DesireEx-
pression {
    public:
        ...
        int Interpret(Context);
        ...
    protected:
        String varName;
        String sortName;
};
```

This class has two instance variables, variable_name and sort_name, which correspond to the symbols appearing on the right-hand side of its grammar rule. It also implements the Interpret method declared in its parent class. The Interpret method checks whether an expression is valid according to the Context. For variables, a variable is legal if the sort is defined. Through the Context, one can check whether
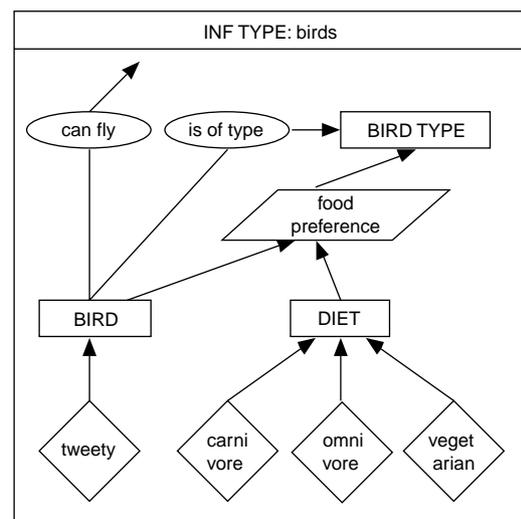

Fig. 12. Example of a DESIRE information type represented in Constraint Graphs.

a symbol is defined and the type of the symbol. The Interpret method can be defined as:

Listing 2: Interpret Method of DesireVariable Class

```
int DesireVariable::Interpret(Context c)
{
  if(c.defined(sortName) && c.typeOf
  (sortName) == 'sorts')
      return 1;
  else
    return 0;
}
```

In the above code, defined and typeOf are methods defined in Context that checks whether a symbol is defined and what its type is.

For NonterminalExpression, as described in Gamma, et al. [5]: 'one such class is required for each rule

R::=R1R2…..Rn

in the grammar'.'maintains an instance variable of type AbstractExpression for each of the symbols R1 through Rn in the grammar'. 'implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn' [5], p. 246.

The second point, maintains an instance variable of type AbstractExpression for each of the symbols R1 through Rn in the grammar, deserves further explanation. At the first glance, the sentence may seem that the authors were advocating using class AbstractExpression as instance variables types. But further investigating shows that is not what the authors means.

Note the use of the word type, instead of class before AbstractExpression. This should be taken to imply that the instance variable is some subtype of AbstractExpression, not precisely AbstractExpression. In class-based languages (e.g. C++), subclassing is subtyping [1]. By subsumption, a value of type A can be viewed as a value of a supertype B. So, if c' is a subclass of c, then an instance of class c' is an instance of class c. A subtype can be used in any place where a supertype can be used. Since any subclass type is of its base class type, the instance variables type can be the type of any subclasses of AbstractExpression. For example, consider the following grammar rule of DESIRE.

4.1. Grammar rule of knowledge base in DESIRE

```
knowledge_base ::= knowledge base <kb_name
                   [knowledge_base_interface]
                   [knowledge_base_reference]
                   knowledge_base_contents
                   end knowledge base.
```

Knowledge_base_interface has been modeled as class DesireKBInterface. Knowledge_base_reference has been modeled as DesireKBRef. And Knowledge_base_contents has been modeled as DesireKBContent. How does one model Knowledge_base? Of course, all the instance variables can be subsumed to DesireExpression, one could use the DesireExpression class as the type of all instance variables. Doing so has no run-time effect. But it has the consequence of reducing static knowledge about the true type of objects. So subtypes are used as the instance variable types if the instance type information is evident from the grammar syntax. The above grammar rule is implemented as in Listing 3.

Listing 3: Class Definition of DesireKB

```
class DesireKB: public DesireExpression {
public:
  ...
  Interpret(Context);
  ...
protected:
  string name;
  DesireKBInterface* kbInterface;
  DesireKBReference* kbReference;
  DesireKBContent kbContent;
};
```

Each R1, R2,…, Rn in the grammar rule is maintained as an instance variable of a specific subclasse type. These can be treated as type DesireExpression or Expression by subsumption if necessary. This approach is advantageous for the following reasons:

- The static object types of symbols in NonterminalExpression are made obvious. It is easier to relate classes to grammar rules.
- It is more type-safe. Since the types of instance variables are all of a specific subclasses type, not the generic AbstractExpression, there is no need to get the instance types at run time. While using these instance variables, there is no need to use dynamic_cast< to get their actual types dynamically.
- Statically specifying instance variable's type can also ensure objects of the wrong type cannot be set/added to the NonterminalExpression. Therefore, the instance of NonterminalExpression will not contain wrong types of instance variables. The creation of NonterminalExpressions will be less error-prone.

The Interpret method for NonterminalExpression calls the Interpret method of instance variables representing R1 through Rn. For example, to check whether a symbol is valid, the Interpret method of class DesireKB can be implemented as in Listing 4.

Listing 4: Interpret Method of DesireKB

```
int DesireKB::Interpret(Context c)
{
    int ret =1;
    if(kbInterface)
        ret = ret && kbInterface-Inter
        -pret(c);
    if(kbReference)
        ret = ret && kbReference-Inter
        -pret(c);
    ret = ret && kbContent.Interpret(c);
    return ret;
}
```

5. Relation to conceptual graphs

In this paper, graphical notations for knowledge in DESIRE are presented, as well as a translator which translates specifications of these notations in a graphical environment called Constraint Graphs to the textual DESIRE representation. Having this graphical interface brings the knowledge modelling in DESIRE closer to other well-known knowledge representation languages, such as Conceptual Graphs [10–12] because a dedicated interchange procedure could be added to the software. The relation between Conceptual Graphs and predicate logic is well-known. The fact that DESIRE is vased on order sorted predicate logic, and the possibility to represent different meta-levels of information within DESIRE, ensures that all knowledge represented in Conceptual Graphs can also be represented in DESIRE. In this section a translation from representations in DESIRE to representations in Conceptual Graphs is defined.

A conceptual graph is a finite, connected, bipartite graph, which consists of two kinds of nodes: concepts and conceptual relations. Concepts are denoted by a rectangle, with the name of the concept within this rectangle, and a conceptual relation is represented as an ellipse, with one or more arcs, each of which must be linked to some concept. Fig. 13 below shows an example conceptual graph, representing the episodic knowledge that a girl, Sue, is eating pie fast.

When comparing conceptual graphs with the graphical notations for DESIRE, many similarities become apparent. For instance, DESIRE's relations are denoted by ellipses, like conceptual relations, and sorts appear as rectangles, like concepts. Other elements however, are harder to translate to a Conceptual Graph notation. Table 2 provides an overview of the translation of DESIRE elements to Conceptual Graphs. Part of this is discussed in some detail.

5.1. Objects

Objects in DESIRE are instances of a sort. In Conceptual Graphs (CG) these instances are represented by individual concepts, i.e. concepts with an individual marker following the concept name. For example, the object tweety of sort BIRD in DESIRE is represented by [BIRD: tweety] in CG. Also anonymous individuals can be translated, e.g. the DESIRE variable X:BIRD is translated into [BIRD: *x] of CG which means that it is known that an individual of type BIRD exists, but it is unknown which individual.

5.2. Functions

In DESIRE, functions group sorts together by mapping them onto another sort. Functions can be regarded to be sub-types of a general CG concept FUNCTION, which takes one or more arguments and produces a result. In DESIRE functions act as a named placeholder for an object of its result, in which the argument(s) and the name of the function ensure the placeholder's uniqueness. Function food pre-ference, for example of Fig. 2, can be represented by the following Conceptual Graph:

[DIET] < -(ARG) < -[food_preference]-(RSLT)-[BIRD_TYPE].

5.3. Relations

Relations in DESIRE can be classified according to their arity. This arity determines the mapping to Conceptual Graphs. 0-ary relations in DESIRE will have to be translated to concepts; concepts in Conceptual Graphs form a graph in itself, like nullary relations form a DESIRE atom in DESIRE. Relations with an arity greater than zero can be translated into either a conceptual relation with the same arity or a combination of a concept and (an)other conceptual relation(s). For example, the relation between: space *
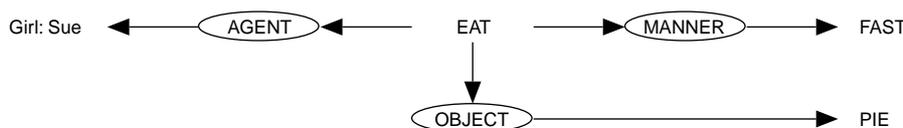

Fig. 13. An example Conceptual Graph.

Table 2
DESIRE, Constraint Graphs, and Conceptual Graphs

Desire element	Graphical equivalent in constraint graphs	Equivalent in conceptual graphs
Object	Diamond	Individual concept
Sort	Rectangle	Generic concept
Sub-sort	Rectangle connected to super-sort by instance-of arrow	Type hierarchy of concepts
Meta-description	Dashed arrow from information type to sort	Conceptual relation -(METALEVEL)-
Function	Parallelogram	Concept FUNCTION
Relation	Ellipse	Conceptual relation or concept and conceptual relation(s)
Information type	Context-box labeled SIG	Context
Knowledge base	Context box labeled KB	Context
Antecedent	Context-box labeled ANT	Context
Consequent	Context-box labeled CONS	Context
NOT-context	Context-box labeled NOT	Negative context
Information type reference to information type	Arrow between information types	Context enclosed in another context
Knowledge base reference to KB	Arrow between knowledge base contexts	Context enclosed in another context
KB reference to information type	Arrow from kb to information type	Comparable to first three and last component in a canon
Rule	Arrow labelled 'implies' between antecedent and consequent	Conceptual relation-(IMP)-

brick * brick in DESIRE could be translated into the following Conceptual Graph:

```
[SPACE] - (BETW) - [BRICK]
                  -[BRICK]
```

This graph is a triadic relation, which could be read as 'a space is between a brick and a brick'. Relation is_traveling_from_to: person * origin * destination however could be translated into the graph

```
[TRAVEL]-
        (AGNT)-[PERSON]
        (ORG)-[ORIGIN]
        (DEST)-[DESTINATION]
```

5.4. Sub-sorts

In DESIRE, hierarchical relations between sorts are allowed. Sub-sorts in DESIRE correspond to the type hierarchy of concepts in Conceptual Graphs. In Conceptual Graphs, hierarchies of both concepts and conceptual relations are possible, but these hierarchical is-a relations are kept in a separate semantic net from other relations that exist in the domain.

5.5. Information types, knowledge bases, antecedents, consequents and not-boxes

DESIRE's information types, knowledge bases, antecedents, consequents and not-contexts can be regarded as contexts in Conceptual Graphs. Although, the graphical DESIRE notation uses a different icons to represent these

contexts, these contexts can be represented by labelled rectangles in Conceptual Graphs, where the labels of these rectangles denote the type of the context (information type, knowledge base, antecedents, consequents, and not-boxes).

5.6. Information type and knowledge base references

In DESIRE, mechanisms exist to enable compositionality of information types and knowledge bases: information type- and knowledge base references, see Figs. 3 and 6. In Conceptual Graphs, contexts that contain other contexts represent this contain-relation by enclosing context-boxes in other context-boxes. Therefore, the graphical DESIRE notations for this compositionality can be translated to Conceptual Graphs notation by placing information types in information types and knowledge bases in knowledge bases.

5.7. Meta-descriptions

Another, different relation exists between information types: the meta-description. A information type A in DESIRE is said to contain a meta-description of a information type B if information type A can be used to specify as terms the atoms that can be specified using the vocabulary of information type B. This means that information type A allows for expressing statements that are at a meta-level with respect to the language defined in information type B. The meta-description relationship is expressed in the graphical notation as a connection, between the meta-described information type B to a sort in the meta-level information type A. This object-meta-level relation between information types, adopted from the area of meta-level architectures, provides a powerful

expressive means to model reflective reasoning patterns. In Conceptual Graphs, this relationship between information types A and B could be expressed as the conceptual relation [Information type B]-(METALEVEL)-[Information type A], with the intended meaning that information type A is at a meta-level with respect to information type B.

5.8. Knowledge base to information type reference

The graphical DESIRE notation for a knowledge base referencing an information type is a connection from the knowledge base to that information type. This connection indicates that the knowledge base, which contains facts and rules that hold in the application domain, uses that information type as the vocabulary to express those facts and rules. One could argue that a information type can be compared to the first three parts of a cannon [11], p. 96, which is a set of four components used to derive canonical graphs: a type hierarchy (sorts and sub-sorts), a set of individual markers (objects), a conformity relation (the sorts the objects belong to) and a finite set of conceptual graphs (the graphs that are true in the domain). The knowledge expressed in knowledge bases would then conform to the fourth component of the canon: the set of graphs that are true in the domain.

5.9. The implies arrow

The last candidate for comparison is the labeled arrow 'implies', which connects the antecedent and consequent of a rule in the graphical DESIRE notation. This arrow can be translated into a relation 'IMP', a logical operator denoting the implication between propositions [11], p. 147. 'IMP' is defined as follows:

relation IMP(x,y) **is** [*x] [*y] (NEG)-[[*x] (NEG)-[[*y]]].

The parameter symbols *x and *y are used to denote the conference relations between elements in the expression. 'IMP' could be read as: there exists an x and a y and it is not true that both x is true and y is not true.

6. Conclusion

In this paper, graphical representations for knowledge structures in DESIRE [3,4] have been presented, together with a graphical editor based on the Constraint Graph environment [6]. Moreover, a translator has been described which translates these graphical representations to textual representations in DESIRE. This software environment can be regarded as a graphical design tool for knowledge in DESIRE, an interface which offers many advantages to a textual interface. First, Constraint Graphs can be used to specify knowledge structures, allowing the user to work with a mouse, pull-down menu's and windows instead of typing the specification conform the textual DESIRE syntax.

Second, the graphical representation of knowledge structures (supported by the software environment for Constraint Graphs) offers a clear visual representation, facilitating communication between domain expert and knowledge engineer in the development process. The strengths of the Constraint Graphs environment as an easy to use representation tool in combination with the DESIRE environment allows for a powerful framework to support the development of knowledge based or multi-agent systems.

From a historical perspective, for the development of the graphical notations for the DESIRE knowledge representation, inspiration has been obtained from a number of different graphical formalisms available in the literature, such as semantic networks, terminological structures and conceptual graphs; e.g. [2,7,10,11,12,14]. Moreover, ideas from meta-level architectures [13] have been used. Based on these sources of inspiration a dedicated graphical language has been developed. As a source of inspiration; e.g. [13]. The research reported in this paper brings the graphical representations bring DESIRE closer to one knowledge representation language, Conceptual Graphs [10,11,12,14], by defining a mapping from DESIRE to Conceptual Graphs (the other direction was already covered). This has led to a more unifying perspective on the DESIRE knowledge representation formalism, and opens the possibility to use DESIRE in connection to the knowledge representation tools of the conceptual graph community.

References

[1] M. Abadi, L. Cardelli, A Theory of Object, Springer, New York, 1996.
[2] Brachman, J. Ronald, R.E. Fikes, H.J. Levesque, Krypton: a functional approach to knowledge representation, IEEE Computer 16 (10) (1983) 67–73.
[3] F.M.T. Brazier, B. Dunin-Keplicz, N.R. Jennings, J. Treur, Formal specification of multi-agent systems: a real-world case in: V. Lesser (Ed.), Proceedings of the First International Conference on Multi-Agent Systems, ICMAS-95, MIT Press, Cambridge, MA, 1995, pp. 25–32. [Extended version in: International Journal of Cooperative Information Systems, M. Huhns, M. Singh, (Eds.), special issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems, vol. 6, 1997, pp. 67–94].
[4] F.M.T. Brazier, C.M. Jonker, J. Treur, Principles of component-based design of intelligent agents, Data and Knowledge Engineering 41 (2002) 1–28.
[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Mass, 1994.
[6] R. Kremer, Constraint Graphs: A Constraint Graphs Meta-Language, PhD Dissertation, Department of Computer Science, University of Calgary, 1997.
[7] Lehmann, Fritz (Eds.), Semantic Networks in Artificial Intelligence, Pergamon Press, Oxford, 1992 [Also published as a special issue of Computers and Mathematics with Applications 23:6–9].
[8] U. Moeller, M. Willems, CG-DESIRE: Formal Specification Using Conceptual Graphs in: B.R. Gaines, M.A. Musen (Eds.), Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop KAW-95, Calgary (1995), pp. 25/1–25/20.

[9] J.T. Nosek, I. Roth, A Comparison of Formal Knowledge Representation Schemes as Communication Tools: Predicate Logic vs. Semantic Network, International Journal of Man-Machine Studies 33 (1990) 227–239.

[10] S. Polovina, J. Heaton, An introduction to conceptual graphs, AI Expert (1992) 36–43.

[11] J.F. Sowa, Conceptual Structures: Information Processing in Mind and Machine, Addison-Wesley, Reading, Mass, 1984.

[12] J.F. Sowa, D. Dietz, Knowledge Representation: Logical, Philosophical, and Computational Foundations, Brooks/Cole, Belmont, CA, 1999.

[13] R.W. Weyhrauch, Prolegomena to a theory of mechanized formal reasoning, Artificial Intelligence Journal 13 (1980) 133–170.

[14] Wolff, Karl Erich Pfeiffer, Heather D. Delugach, Harry S. (eds.) Conceptual Structures at Work: Proc. Of the 12th International Conference on Conceptual Structures, ICCS 2004. Lecture Notes in Computer Science, vol. 3127. Springer Verlag, 2004.