Cranfield UNIVERSITY

Decision Engineering Report Series
Edited by Rajkumar Roy and Clive Kerr

INTEGRATION OF ONTOLOGY WITH EXPERT SYSTEM: A CASE STUDY

By Andrew Krizhanovsky

February 2005

Cranfield University
Cranfield
Bedfordshire
MK43 OAL
United Kingdom

http://www.cranfield.ac.uk

ISBN 1-861940-96-3

Edited by:

Dr. Rajkumar Roy
r.roy@cranfield.ac.uk

Dr. Clive Kerr
c.i.kerr@cranfield.ac.uk

Enterprise Integration
Cranfield University
Cranfield
Bedfordshire
MK43 OAL
United Kingdom

http://www.cranfield.ac.uk

Series librarian:

John Harrington
j.harrington@cranfield.ac.uk

Kings Norton Library
Cranfield University

Publisher:

Cranfield University

Abstract

This report describes the software prototype for the integration of ontology with expert system: requirements, the design, and the implementation of the prototype. The prototype is based on such tools and platforms as: Protégé, CLIPS, and Perl/Tk. The detail description of these tools and their integration in the prototype are presented in this report.

The prototype uses ontology as knowledge representation model. This made possible to solve the problem of semantic consistency, since ontology define notation for the knowledge domain. Views on ontology specification are shared with Kitamura and Mizoguchi [Kitamura and Mizoguchi, 2004]. They specify ontology as a set of concepts with informal definitions, a set of relations holding among these concepts not limited to hierarchical ones (*is-a* and *part-of*), and a set of axioms to formalize the definitions and relations.

Protégé ontology editor is used to design the ontology, create instances of classes, and set values of attributes. An example of ontology development is presented in the report. The ontology design methodology is adapted, presented and applied.

CLIPS is a powerful inference engine and programming language developed by NASA. An inference engine derives answers from a knowledge base. CLIPS is the unique language because it combines the properties of declarative (e.g. Prolog, LISP) and procedural (e.g. C++, Java, VB) programming languages. In a declarative programming paradigm a set of attributes that a solution should have are specified rather than set of steps to obtain such a solution as in a procedural programming paradigm. So, there are three ways to represent knowledge in CLIPS: rules (declarative way), functions (procedural way), and object-oriented programming.

CLIPS has been designed for integration with other languages such as C and Ada. This feature of CLIPS is used in the prototype to connect CLIPS and Perl/Tk.

Perl/Tk is an interpreted scripting language for making programs with Graphical User Interfaces (GUI). This language is suitable for rapid application development. Perl/Tk has a huge library of ready to use GUI elements. Since Perl/Tk is a part of Perl language, so it is possible to call a Perl function from GUI. It is important here because Perl text manipulation capabilities are used in the prototype.

The main results are the adapted methodology for ontology design, the test ontology building, logic rules written in CLIPS, GUI implemented in Perl/Tk. The prototype development shows that integration of the ontology editor with inference engine based on high level programming language (Protégé, CLIPS, and Perl, respectively) is successful union.

Keywords: *Ontology, Inference engine, Expert system, Protégé, CLIPS, Perl/Tk.*

Table Of Contents

1. Introduction

The purpose of the designed system is to integrate ontology and expert system. The test ontology describes parts of an automobile engine and requirements and constraints between these parts. The main function of the system is to provide a tool that allows users (design engineers) to define engine parts and relations between them, to fill properties of these parts, to test different parts of the model (e.g. compatibility tests).

The prototype has the following requirements:

- *Design support*. The management of ontology data is based on expert knowledge. The system has to present the ability to develop and manage the requirements and constraints.
- *Semantic consistency*. Different users can use different notation, terms, definitions to describe the same concept, while global constraints have to bind them.

The first requirement “design support” is based on expert knowledge and is satisfied in the prototype with the help of the rule-based expert system CLIPS and the ontology editor Protégé. Expert knowledge is stored in the form of an ontology (e.g. parts of engine, properties of these parts) and a set of rules (e.g. compatibility of different parts).

An ontology using solves the problem of “semantic consistency”, since ontology define notation for the given knowledge domain. The prototype uses ontology as knowledge representation model. Views on ontology specification are shared with Kitamura and Mizoguchi [Kitamura and Mizoguchi, 2004]. They specify ontology as a set of concepts with informal definitions, a set of relations holding among these concepts not limited to hierarchical ones (*is-a* and *part-of*), and a set of axioms to formalize the definitions and relations.

The prototype accumulates merits of such tools and platforms as: Protégé, CLIPS, and Perl/Tk. Protégé is an ontology editor with rich visual interface. At the same time CLIPS is a powerful inference engine. It is possible to manage objects and relations of ontology classes and instances (created in Protégé) by CLIPS rules.

The graphical user interface (GUI) gives to user possibilities to work with ontology data and run CLIPS rules and functions. GUI part of the prototype was developed at the basis of Perl/Tk.

The use case diagram (Figure 1) shows that three roles play in the prototype. Fist, an ontology engineer works with Protégé ontology editor. S/he designs the ontology, creates classes, assigns attributes to classes, and defines relations between classes. Second, domain expert (and CLIPS programmer) creates CLIPS rules and functions to treat ontology data. Thus, ontology data and rules are used together in CLIPS to derive answers. Third, user (e.g. designer) works with this ontology-based expert system via GUI, rather than CLIPS command line mode.

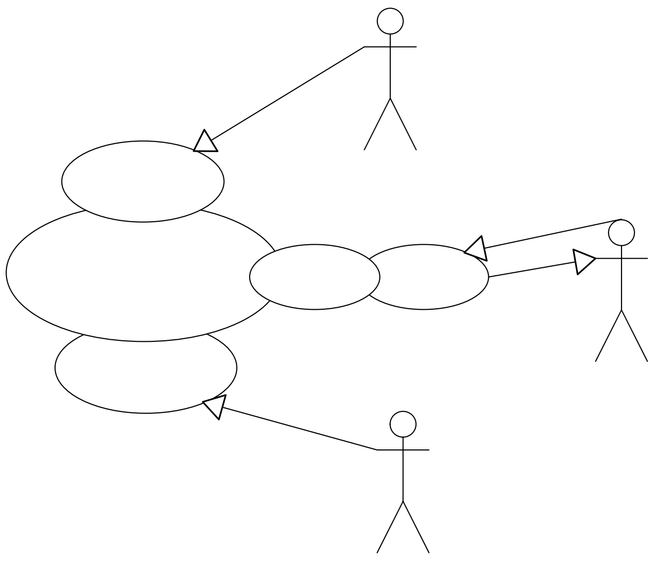


Figure 1: The prototype use case diagram

Thus the using of CLIPS and Protégé in the prototype defines several roles of users (ontology engineer, user/designer, domain expert) which work with:

- ontology classes, attributes of classes and relations between classes in Protégé (ontology engineer);
- rules, functions, and messages in CLIPS (domain expert);
- instances of ontology classes, values of attributes in Protégé (user/designer);
- special requests (search, test, etc.) using objects of classes Component and Requirement in the prototype (user/designer).

The next section introduces platforms used in the prototype: CLIPS and Perl/Tk. Third section presents the prototype — from project requirements and the adapted ontology design methodology till GUI details. Forth section discusses a future work.

2. Prototype Prerequisites

This section describes several of used tools (CLIPS, Perl, and Perl/Tk) in the prototype development.

2.1. CLIPS — Expert System

CLIPS is an expert system tool developed by the Software Technology Branch (STB), NASA/Lyndon B. Johnson Space Center [Giarratano, 2002].

There are three ways to represent knowledge in CLIPS:

- *Rules*, which are primarily intended for heuristic knowledge based on experience.

- *Deffunctions and generic functions*, which are primarily intended for procedural knowledge.
- *Object-oriented programming*, also primarily intended for procedural knowledge.

Software could be developed using only rules, only objects, or a mixture of objects and rules.

CLIPS has also been designed for full integration with other languages such as C and Ada. In fact, CLIPS is an acronym for C Language Integrated Production System. Rules and objects form an integrated system too since rules can pattern-match on facts and objects. In addition to being used as a stand-alone tool, CLIPS can be called from a procedural language, perform its function, and then return control back to the calling program. Likewise, procedural code can be defined as external functions and called from CLIPS.

2.2. Rapid Application Development: Perl & Perl/Tk

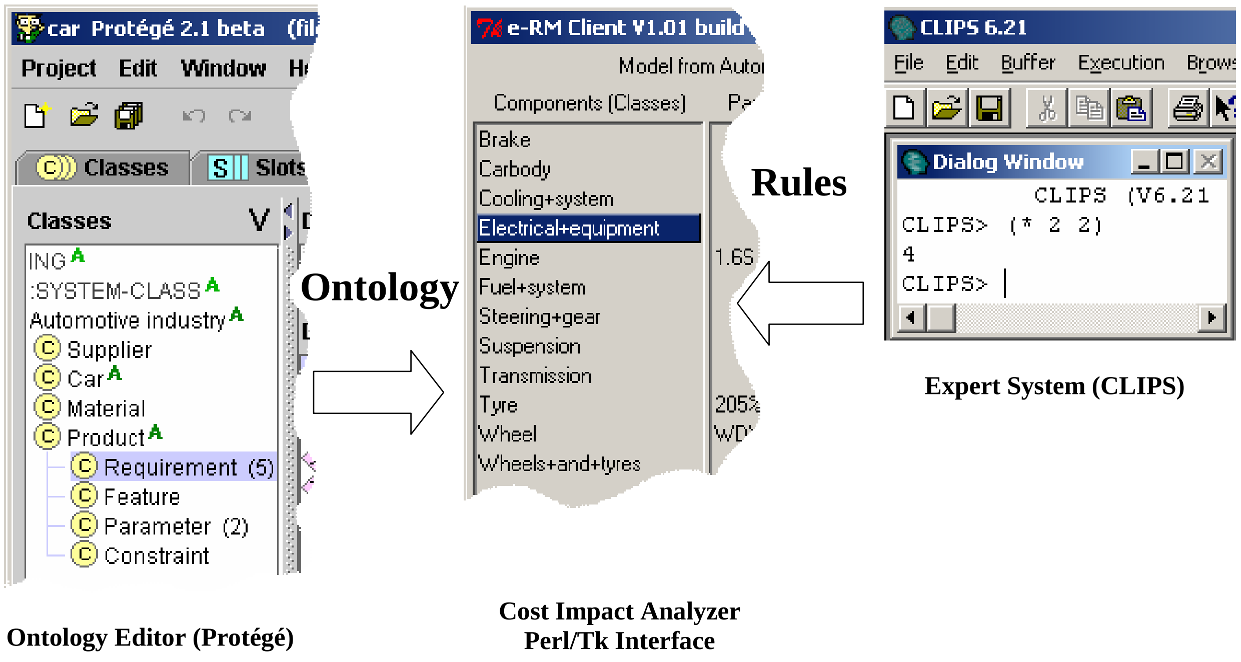
Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more.

The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). Its major features are that it's easy to use, supports both procedural and object-oriented (OO) programming, has powerful built-in support for text processing, and has one of the world's most impressive collections of third-party modules [Kirrily, 2003].

Perl/Tk (also known as pTk or ptk) is an interpreted scripting language for making widgets and programs with Graphical User Interfaces (GUI). (Examples of widget programs [not necessarily written in perl/Tk] include xterm, xclock, most web-browsers, etc.) [Laird, 2000].

3. Prototype

The prototype accumulates merits of such tools and platforms as: Protégé, CLIPS, and Perl/Tk (Figure 2). The advantage of Protégé ontology editor is that the result ontology could be saved in the format readable by CLIPS. So CLIPS inference engine can manage objects and relations of ontology classes and instances created in Protégé. User can work with CLIPS rules and ontology data via the graphical user interface (GUI) based on Perl/Tk module.



```
        CLIPS (V6.21
CLIPS> (* 2 2)
4
CLIPS> |
```

Figure 2: Software prototype scheme

3.1. Example of the Test Ontology Development

This section describes the step by step creation of test ontology for automobile engine. Protégé editor [Protégé, 2000] was used.

There are some fundamental rules in ontology design methodology proposed by [Noy et al., 2001]. They can help to make design decisions in many cases.

- There is no one correct way to model a domain — there are always viable alternatives. The best solution almost always depends on the application that you have in mind and the extensions that you anticipate.
- Ontology development is necessarily an *iterative process*.
- Concepts in the ontology should be close to objects (physical or logical) and relationships in your domain of interest. These are most likely to be nouns (objects) or verbs (relationships) in sentences that describe your domain.

The ontology design methodology adapted from [Noy et al., 2001] consists of the following steps:

1. Determine the domain and scope of the ontology;
2. Formulate ontology competency questions;
3. The work with classes via an ontology editor:
   - o Define the classes and the class hierarchy;
   - o Define the properties of classes;
   - o Create instances of classes.

The step "reusing existing ontologies" of the original methodology is skipped since author did not find any available ontology related to automobile engines.

The first step of methodology is very important for the prototype, since scope of ontology and competency questions are the basis (1) to understand what objects needed in the ontology and (2) to define what functions and rules should be implemented in inference engine.

3.2.1. Step 1. Determine the Domain and Scope of the Ontology

First we define ontology domain and scope. That is, answer several basic questions:

- Q: What is the domain that the ontology will cover?
- A: It is an automobile engine: parts, properties, and models of engines.

- Q: For what we are going to use the ontology?
- A: The main idea is to integrate the ontology with inference engine in order (1) to propagate changes via constraints, (2) to define problem domain requirements for ontology classes, (3) check a satisfaction of requirements.

- Q: Who will use and maintain the ontology?
- A: They are engine designers and ontology engineers. Engine designers are ontology users. Ontology engineers maintain the ontology.

The answers to these questions may change during the ontology-design process, but at any given time they help limit the scope of the ontology model.

3.2.2. Step 2. Formulate Ontology Competency Question

One of the ways to determine the scope of the ontology is to sketch a list of questions that an ontology-based knowledge base should be able to answer, *competency questions*.

In the domain of automobile engine, the following are the possible competency questions:

1. Is it possible to change a value of some attributes without affecting any others? List possible components and constraints which bind these conflict components with changed attribute.
2. What parts form the component X? or What other components are bound with the component X (via some constraints)?
3. Does it perform all the requirements related to some model? What requirements are violated?

4*. What components (properties) need to be changed (to be affected) in the car model if the component X is changed in the way Y? (e.g. What attributes is changed in Toyota model if the engine 1GZFE is replaced by 5VZFE?).

Judging from this list of questions, the ontology will include the information on various engine models (and component) characteristics. It is need information about compatibility of components. The constraints expressing value of one property via others are necessary.

During development of ontology with the help of Protégé, the several rules and functions (Table 1) were developed answering successfully to competency questions. These rules and functions were implemented by means of a tool for building expert systems [CLIPS, 2003].

Table 1: Rules answering to ontology competency questions (CQ)

CQ	Name	Description
1, 2	List-Requirements-for-Component	List all requirements related to the Component
1, 2	List-Tied-Components	List all components related to the Component through requirements (indicate level of connectivity) (recursive function)
2	print-parts	Print all parts formed the assembly model
3	List-Model-Requirements	List all requirements related to assembly model
3	test-req	Test satisfaction of the requirement

First column (CQ) in Table 1 shows the number of competency question solved by corresponding CLIPS' function. Names and descriptions of implemented CLIPS functions (or messages, or rules) are presented in columns "Name", and "Description". The following name convention is used in CLIPS: message handlers — lowercase (e.g. test-req), function and rules — Title-Case (e.g. List-Tied-Components).

It should be mentioned that the first three competency questions can be answered with the help of the current version of prototype, but the prototype cannot answer yet for the forth question marked by the asterisk.

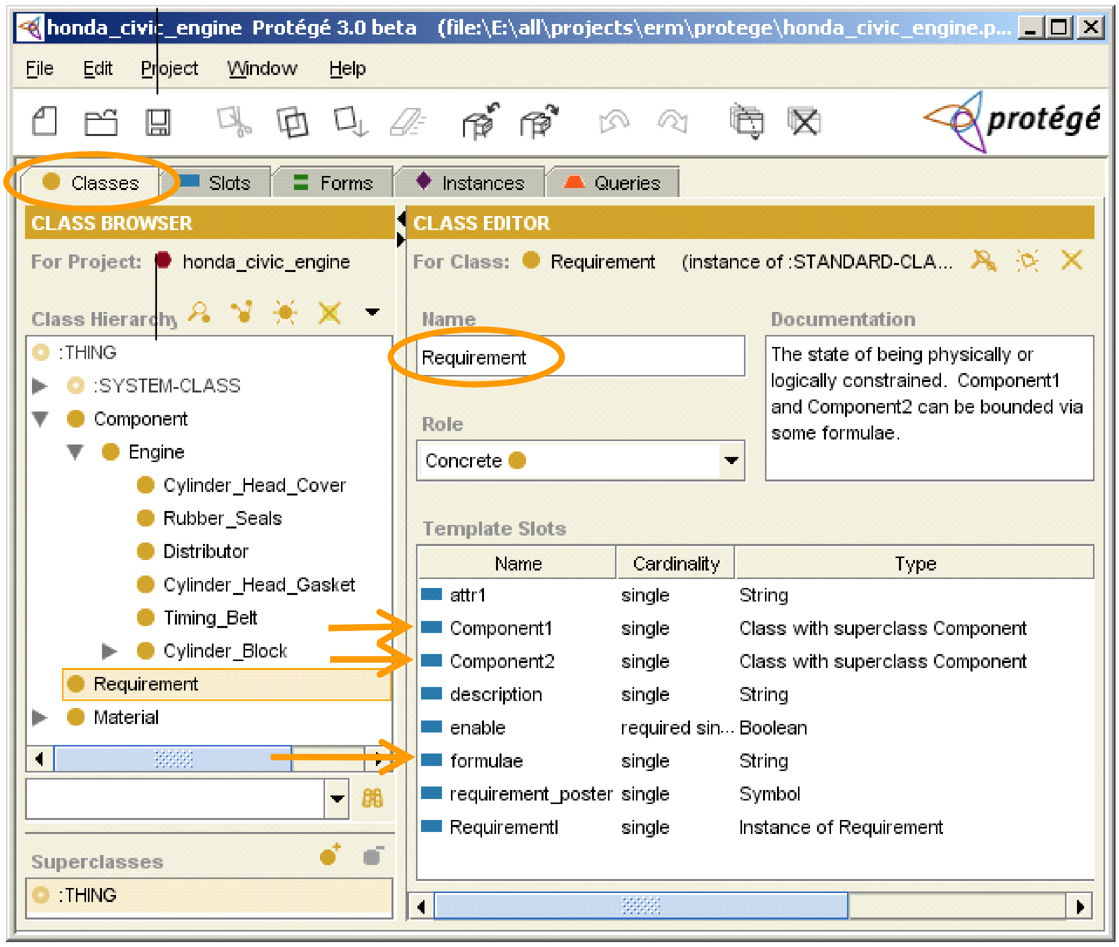
3.2.3. Step 3. The Work with Classes

Comments on data are usually much more helpful than on algorithms [Pike, 1989]. The most important classes (i.e. data here) designed in Protégé and used in CLIPS are Components and Requirements. The test ontology is designed to be used with an expert system, since it consists of data (class Components, parts of engine here) and functional constraints (class Requirements).

Class Component describes parts of an engine, e.g. cylinder block, distributor, timing belt, etc. Class Requirement describes design requirement imposed by designer or customer (property "requirement_poster" has value "designer" or "customer" and shows an importance of the requirement). Attributes "Component1", "Component2" refer to Component objects bound by Requirement object (Figure 3).

There are two types of requirements: qualitative and computable. All information about qualitative requirement is stored in "description" attribute (e.g. "Cylinder block's dimensions affect on engine form."). Computable (quantitative) requirements are described via "formulae" attribute. For example, the formulae "(<= Engine.weight

130)” in CLIPS’ syntax means that maximum allowed weight of engine is 130 kg (Figure 4). These computable requirements could be executed in CLIPS to check their satisfaction/ violation. The question about automatic propagation of component changes via requirement is still open.



For Project: honda_civic_engine

:THING
:SYSTEM-CLASS
Component
Engine
Cylinder_Head_Cover
Rubber_Seals
Distributor
Cylinder_Head_Gasket
Timing_Belt
Cylinder_Block
Requirement
Material

For Class: Requirement (instance of :STANDARD-CLA...

Name: Requirement

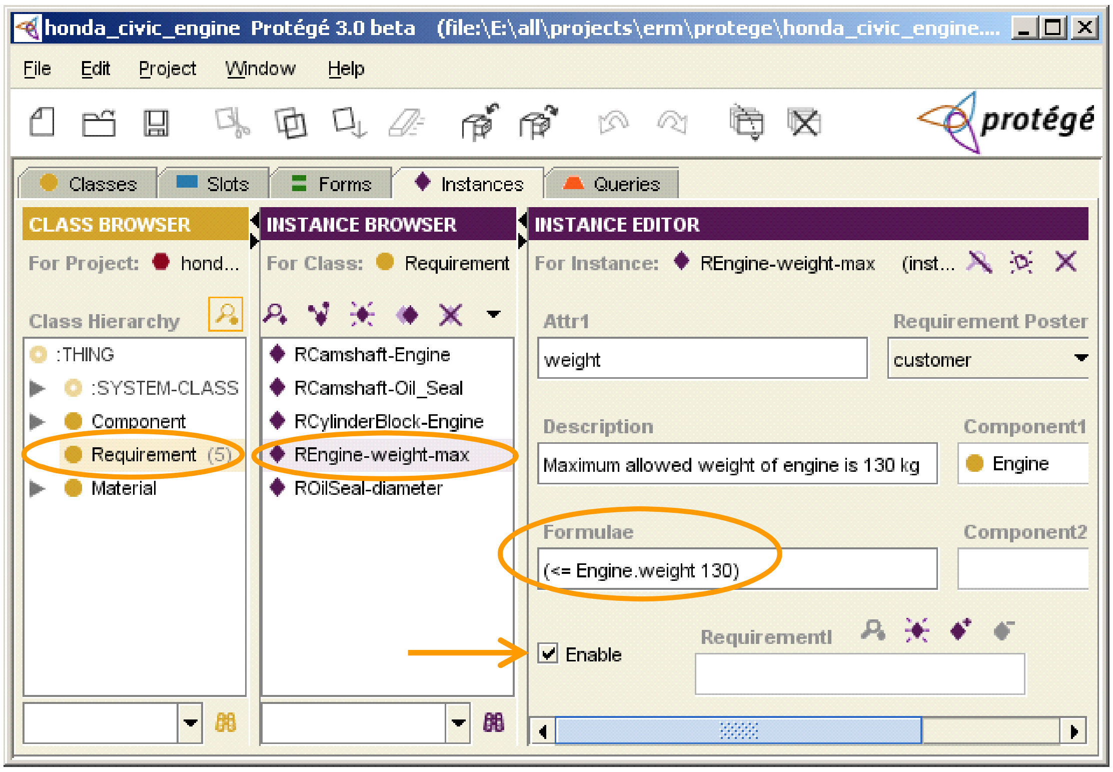
Role: Concrete

Documentation: The state of being physically or logically constrained. Component1 and Component2 can be bounded via some formulae.

Template Slots

Name	Cardinality	Type
attr1	single	String
Component1	single	Class with superclass Component
Component2	single	Class with superclass Component
description	single	String
enable	required sin...	Boolean
formulae	single	String
requirement_poster	single	Symbol
Requirementl	single	Instance of Requirement

Figure 3: The class Requirement in the hierarchy of the test ontology “engine” in Protégé. The most important (for the expert system) attributes (“Component1”, “Component2”, and “formulae”) of the class Requirement are marked by arrows. They bound the class Requirement with Component one



honda_civic_engine Protégé 3.0 beta (file:\E:\all\projects\erm\protege\honda_civic_engine....

CLASS BROWSER
For Project: ● hond...
Class Hierarchy
:THING
:SYSTEM-CLASS
Component
Requirement (5)
Material

INSTANCE BROWSER
For Class: ● Requirement
RCamshaft-Engine
RCamshaft-Oil_Seal
RCylinderBlock-Engine
REngine-weight-max
ROilSeal-diameter

INSTANCE EDITOR
For Instance: REngine-weight-max (inst...

Attr1	weight
Requirement Poster	customer
Description	Maximum allowed weight of engine is 130 kg
Component1	● Engine
Formulae	(<= Engine.weight 130)
Component2	
Enable	✓
RequirementI	

Figure 4: The attributes of the instance “Rengine-weight-max” of the class Requirement are presented in the right pane. This is an example of computable Requirement with “formulae” attribute in CLIPS syntax

Integration of CLIPS language and Protégé visualization (Figure 4) allows to code different types of constraints used in the prototype:

- One-argument requirements: the quantitative constraint *REngine-weight-max* depends on one attribute *weight* of one component *Engine*, so the formula consists of only one variable *Engine.weight:*

```
([REngine-weight-max] of Requirement
    (description "Maximum allowed weight of engine is 130 kg")
    (Component1 Engine)
    (attr1 "weight")
    (formulae "(<= Engine.weight 130)"))
```

- Two-argument requirements: the qualitative constraint *RCylinderBlock-Engine* depends on two components *Cylinder_Block* and *Engine*. There is no any formula since this requirement is qualitative one:

```
([RCylinderBlock-Engine] of Requirement
    (Component1 Cylinder_Block)
    (Component2 Engine)
    (description "Cylinder block's dimensions affect on engine form."))
```

- Complex requirement is a requirement depending on other requirement. These complex requirements allow to define relations between requirements (i.e. integrate requirements) and to define more than two arguments. For example, the requirement *RCamshaft-Oil_Seal* depends on the requirement *ROilSeal-diameter*:

```
([RCamshaft-Oil_Seal] of  Requirement
    (attr1 "diameter")
    (Component1 Camshaft)
    (Component2 Oil_Seal)
    (description "Diameter of camshaft and oil seal have to be equal.")
    (formulae "(eq Camshaft.diameter Oil_Seal.diameter)")
    (RequirementI [ROilSeal-diameter]))
```

3.2. Implementation of CLIPS Rules and Functions

The Figure 5 presents the listing of CLIPS' function which search requirements related to component. This function traverses all instances of class Requirement with the help of simple cycle (line 05). If Requirement object "?req" is enabled (line 08) and one of two pointers Component1 or Component2 is equal to the sought component "?x" (lines 09-10) then the description of this requirement (line 14) and names of these components (lines 15-16) will be printed.

```
01 ;;;* List all requirements related to Component X *
02 ;;;* (x is a class, e.g. Engine, Wheel)          *
03 ;;;*******************************************
04 (deffunction List-Requirements-for-Component (?x)
05   (do-for-all-instances
07       ((?req Requirement))
08       (if (and (eq ?req:enable TRUE)
09              (or (eq ?req:Component1 ?x)
10                 (eq ?req:Component2 ?x)
11              )
12          )
13        then
14          (printout t ?x " constraint: " ?req:description
15             " Component1:" ?req:Component1
16             " Component2:" ?req:Component2
17             crlf)
18       )
19   )
20 )
```

Figure 5: Listing of CLIPS' function List-Requirements-for-Component

The result of this function is presented in the Figure 6. In first case the function found two constraints for the component "Engine", and one constraint for the component "Wheel" in second case.

```
CLIPS> (List-Requirements-for-Component Engine)
Engine constraint: Maximum allowed weight of engine is 130 kg
Component1:Engine Component2:nil
Engine constraint: Engine defines fuel system Component1:Fuel+system
Component2:Engine
FALSE
CLIPS> (List-Requirements-for-Component Wheel)
Wheel constraint: Diameter of wheel have to be less than 40 cm.
Component1:Wheel Component2:nil
FALSE
CLIPS>
```

Figure 6: Log of interactive execution of function List-Requirements-for-Component in CLIPS environment

3.3. Graphical User Interface

3.4.1. Data Flow: Binding CLIPS and Perl Via CAPE

When CLIPS functions were written then the next task was to organize an access to these functions using Perl/Tk graphical interface. Since there is no direct access from CLIPS to Perl/Tk, the module CAPE was used.

CAPE system (written by Robert Inder [Inder, 1999]) binds Perl with CLIPS. During the prototype development, CAPE source code was modified to run CAPE under Win32 (originally CAPE works in UNIX) and to integrate Perl/Tk GUI with CAPE command line system. Since CAPE system is distributed under GPL license, all CAPE modifications are made available [CAPE_Tk, 2004].

Sockets and process pipes (for interprocess communication in Unix, Windows) helps to connect different parts (GUI and CLIPS) and different programming languages (Perl and C++) of the multithreaded application. The following definitions explain the sockets and pipes [Microsoft, 2003], [FOLDOC, 2005], [HTD, 2005], [Webopedia, 2005]:

- *Socket*. The Berkeley Unix mechanism (a software object) 1) for creating a virtual connection between processes, 2) for connecting an application with a network protocol. A program can send and receive TCP/IP messages by opening a socket and reading and writing data to and from the socket. This simplifies program development because the programmer need only worry about manipulating the socket and can rely on the operating system to actually transport messages across the network correctly. Note that a socket in this sense is completely soft — it's a software object, not a physical component. The socket has associated with it a socket address, consisting of a port number and the local host's network address.

- *Pipe*. One of Unix's buffers which can be written to by one asynchronous process and read by another, with the kernel suspending and waking up the sender and receiver according to how full the pipe is. A pipe connects two processes so one's output can be used as the other's input. A pipe is available in Windows too.

The part of the prototype written in Perl consists of two components: client (Perl/Tk GUI) and server (interaction between CLIPS and CAPE). Client and server are communicated via sockets and can be run on different computers, although they should know IP-addresses of each other. In its turn, client consists of two threads (communicated by pipes): (1) the thread responsible for GUI, and (2) the thread sending and getting messages from the server.

Every user request generates (from GUI) the following sequence of actions (Figure 7). Client GUI sends the message (via pipe inside client) from client’s visual interface thread to clients’ communication thread (this interaction implemented in Perl code). Clients’ communication thread sends the message to CAPE server via socket (Perl code). CAPE sends the message to CLIPS (C++ code). Then CLIPS engine executes rules and function using loaded rules and the test ontology. It returns message to CAPE server (C++ code), to client via socket (Perl code), to client visual interface thread via pipe (Perl code). Client prints result to user and updates elements of GUI.

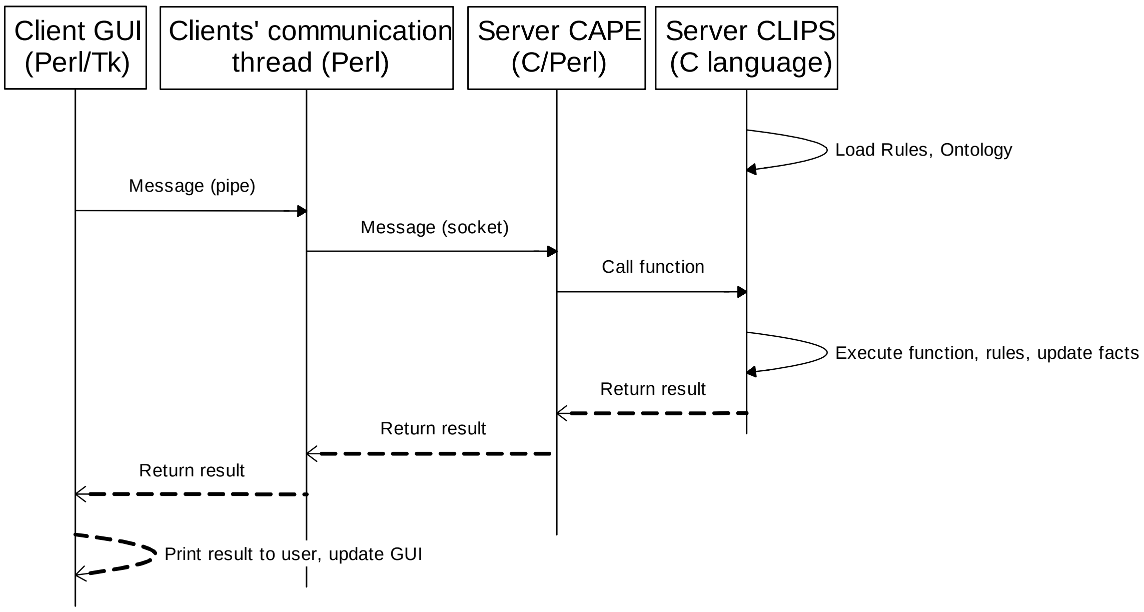
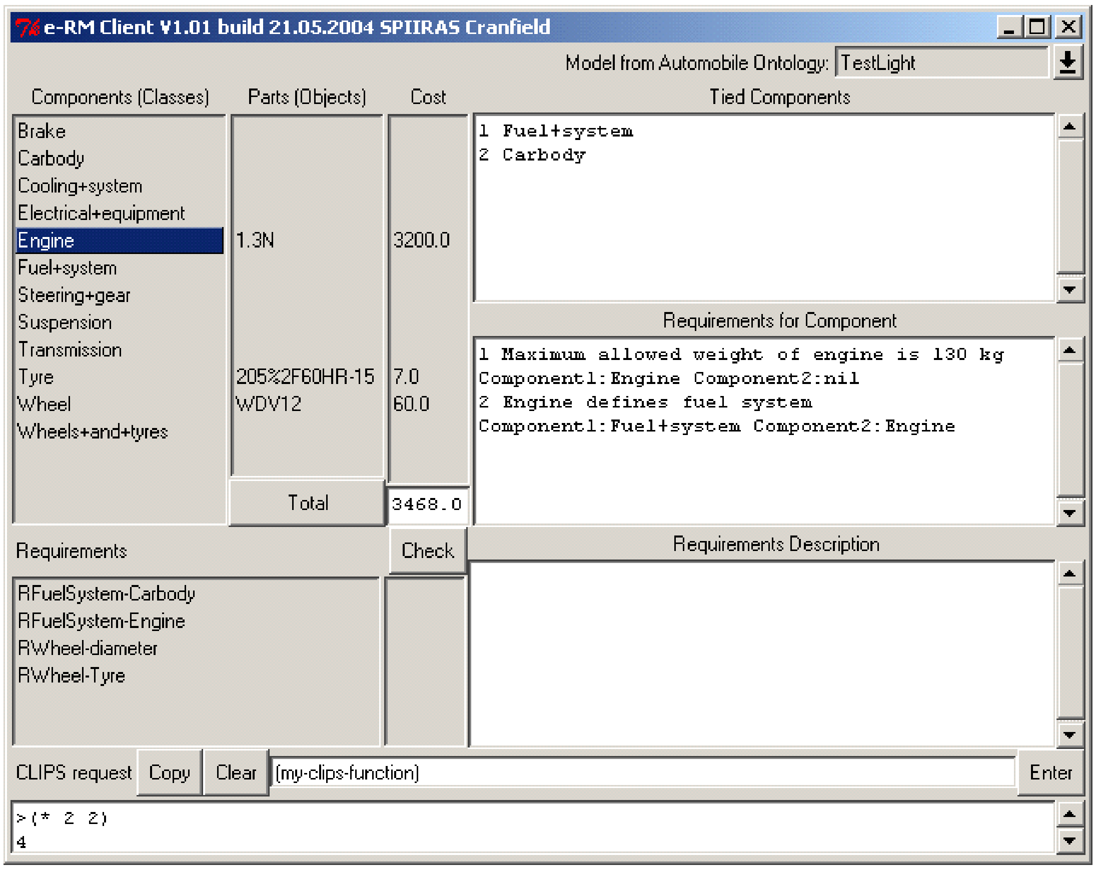


Figure 7: User request processing: message passing from GUI to CLIPS engine

3.4.2. GUI Design and Implementation

This section describes design and implementation of the GUI element "CLIPS_Frame1". All elements of GUI presented in Figure 8.



e-RM Client V1.01 build 21.05.2004 SPIIRAS Cranfield

Model from Automobile Ontology: TestLight

Components (Classes)	Parts (Objects)	Cost
Brake		
Carbody		
Cooling+system		
Electrical+equipment		
Engine	1.3N	3200.0
Fuel+system		
Steering+gear		
Suspension		
Transmission		
Tyre	205%2F60HR-15	7.0
Wheel	WDV12	60.0
Wheels+and+tyres		
	Total	3468.0

Tied Components

```
1 Fuel+system
2 Carbody
```

Requirements for Component

```
1 Maximum allowed weight of engine is 130 kg
Component1:Engine Component2:nil
2 Engine defines fuel system
Component1:Fuel+system Component2:Engine
```

Requirements

RFuelSystem-Carbody
RFuelSystem-Engine
RWheel-diameter
RWheel-Tyre

(my-clips-function)

```
>(* 2 2)
4
```

Figure 8. The prototype GUI

The goal of the element "CLIPS_Frame1" is to provide an interface to CLIPS. The scheme of this element consists of three buttons ("Copy", "Clear", "Run"), the text field "CLIPS_entry" and the label "CLIPS request" (Figure 9). Indeed, user can write a request to CLIPS engine typing the text of request in the text field "CLIPS_entry" (e.g. the request "(my-clips-function)" in Figure 10). To start user request processing (Figure 7) user press the button "Run". The result of CLIPS calculations are printed in the text area located below the label "CLIPS request" (Figure 8). This result could be copied to clipboard (button "Copy") or erased from GUI (button "Clear").

Figure 9 presents the scheme of GUI element using the following notation:

- Type of Tk widget is in square brackets, e.g. [Label], [Button];
- Name of widget control is marked by apostrophe, e.g. 'CLIPS_entry';
- CLIPS function is enclosed in brackets, e.g. (DoAndGetOutput);

```
 ┌─ CLIPS_frame1 ──────────────────────────────────────────────────────────────┐
 │'CLIPS request:'│ 'Copy'  │ 'Clear'  │ 'CLIPS_entry'     │ 'Run' [Button]  │
 │[Label]         │[Button] │ [Button] │ (DoAndGetOutput)  │   or  [Enter]   │
 └────────────────┴─────────┴──────────┴───────────────────┴─────────────────┘
```

Figure 9: Scheme of GUI component "CLIPS_frame1"

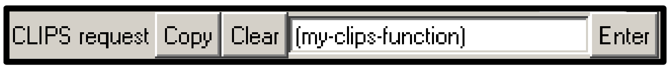
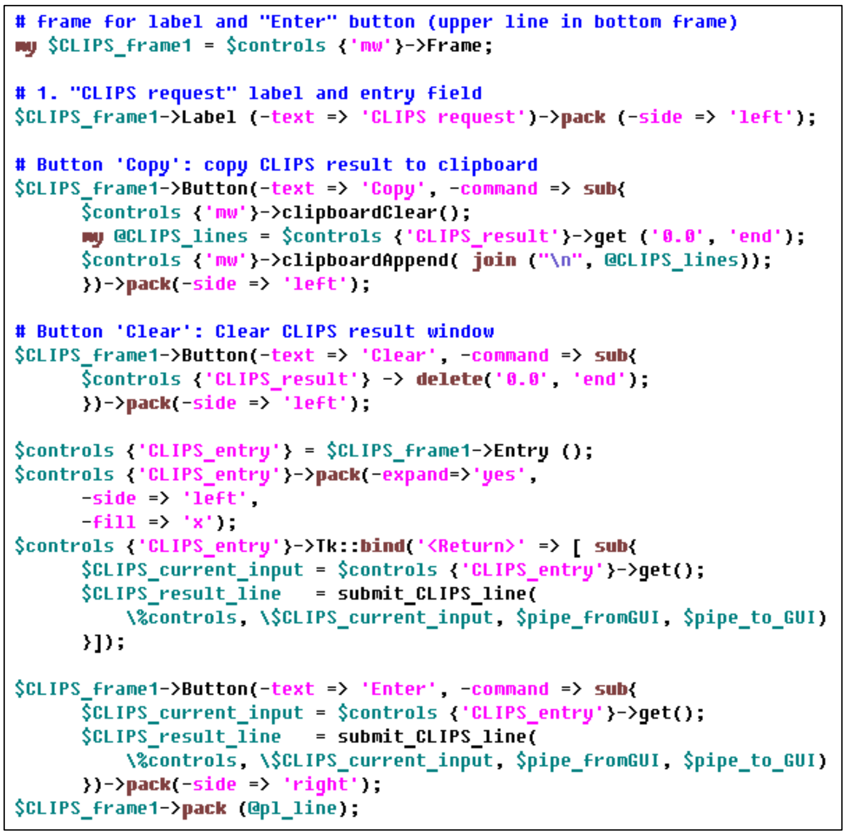


Figure 10: Screenshot of GUI component “CLIPS_frame1”

The sequence of figures (Figure 7, Figure 9, Figure 10) presents the same GUI element from different points of view: the scheme of element, screenshot of implemented element, and the Perl/Tk code explaining the features of this element to Perl interpreter, respectively.



```
# frame for label and "Enter" button (upper line in bottom frame)
my $CLIPS_frame1 = $controls {'mw'}->Frame;

# 1. "CLIPS request" label and entry field
$CLIPS_frame1->Label (-text => 'CLIPS request')->pack (-side => 'left');

# Button 'Copy': copy CLIPS result to clipboard
$CLIPS_frame1->Button(-text => 'Copy', -command => sub{
      $controls {'mw'}->clipboardClear();
      my @CLIPS_lines = $controls {'CLIPS_result'}->get ('0.0', 'end');
      $controls {'mw'}->clipboardAppend( join ("\n", @CLIPS_lines));
      })->pack(-side => 'left');

# Button 'Clear': Clear CLIPS result window
$CLIPS_frame1->Button(-text => 'Clear', -command => sub{
      $controls {'CLIPS_result'} -> delete('0.0', 'end');
      })->pack(-side => 'left');

$controls {'CLIPS_entry'} = $CLIPS_frame1->Entry ();
$controls {'CLIPS_entry'}->pack(-expand=>'yes',
      -side => 'left',
      -fill => 'x');
$controls {'CLIPS_entry'}->Tk::bind('<Return>' => [ sub{
      $CLIPS_current_input = $controls {'CLIPS_entry'}->get();
      $CLIPS_result_line    = submit_CLIPS_line(
          \%controls, \$CLIPS_current_input, $pipe_fromGUI, $pipe_to_GUI)
      }]);

$CLIPS_frame1->Button(-text => 'Enter', -command => sub{
      $CLIPS_current_input = $controls {'CLIPS_entry'}->get();
      $CLIPS_result_line    = submit_CLIPS_line(
          \%controls, \$CLIPS_current_input, $pipe_fromGUI, $pipe_to_GUI)
      })->pack(-side => 'right');
$CLIPS_frame1->pack (@p1_line);
```

Figure 11: Perl/Tk code of GUI component “CLIPS_frame1”

4. Concluding Remarks

This report describes the software prototype for the integration of ontology with expert system: requirements, the design, and the implementation.

The main achievements are the following: standalone application was developed. The following components of this application were designed and implemented: CLIPS’ rules and functions, and Perl/Tk visual interface. The test ontology describing automobile engine was developed (via Protégé ontology editor) and tested in the prototype.

The adapted ontology design methodology is proposed and used to create the test ontology. It consists of the following steps:

1. Determine the domain and scope of the ontology;
2. Formulate ontology competency questions;
3. The work with classes via an ontology editor:
    - Define the classes and the class hierarchy;
    - Define the properties of classes;
    - Create instances of classes.

The main idea is to integrate an ontology with an inference engine in order (1) to propagate changes via constraints, (2) to define problem domain requirements for ontology classes, (3) check a satisfaction of requirements.

This work shows the solution of the following important issue. On the one hand, designers needed in rich and simple interface in order to store, edit, integrate information (about automobile engines here). It was done with the help of the Protégé editor. On the other hand, expert system (with logic programming capabilities) is needed to describe complex requirements and various dependencies between objects (created by designers in an ontology editor) and to manipulate these objects. CLIPS (stable and fast expert system) proved in the prototype to be suitable for these purposes.

Several roles of users working with the prototype are defined. Their interactions with the system are presented with the help of UML diagrams. Another UML diagram (with the detail description of interprocess communication) helps to describe message passing from GUI to CLIPS engine. Different types of constraints with examples are presented and described in the report.

References

CAPE_Tk (2004) URL http://whinger.narod.ru/soft/clips_tk/index.html
CLIPS 6.21. A tool for Building Expert Systems. URL http://www.ghg.net/clips/CLIPS.html
FOLDOC (2005) Free On-line Dictionary of Computing URL http://foldoc.doc.ic.ac.uk
Giarratano, J.C. (2002) CLIPS User's Guide Version 6.20. http://www.ghg.net/clips/download/documentation
GraphViz (2004) http://www.research.att.com/sw/tools/graphviz
Gruninger, M., Atefi, K., & Fox, M.S. (2000) Ontologies to Support Process Integration in Enterprise Engineering. Computational and Mathematical Organization Theory, 6(4):381-394.
HTD(2005) High-Tech Dictionary URL http://www.computeruser.com
Inder, R. (1999) CAPE: Extending CLIPS for the Internet. In Knowledge-Based Systems V13 pp 151-157. URL http://cape.sourceforge.net
Kirrily, R. (2003) perlintro — a brief introduction and overview of Perl. URL http://www.perl.com

Kitamura, Y., Mizoguchi, R. (2004) Ontology-based systematization of functional knowledge. Journal of Engineering Design, Taylor & Francis, Vol. 15, Number 4, 327-351. URL http://www.ei.sanken.osaka-u.ac.jp/pub/kita/

Laird, C. (2000) Perl/Tk FAQ URL: http://www.cpan.org/authors/id/C/CL/CLAIRD/ptkFAQ.html

Microsoft: (2005) Unix Application Migration Guide (Patterns & Practices) URL http://www.willydev.net/descargas/prev/unix.pdf

Noy, N. F. & McGuinness, D. L. (2001) Ontology Development 101: A Guide to Creating Your First Ontology. Knowledge Systems Laboratory, URL http://protege.stanford.edu/publications/ontology_development/ontology101.pdf

Protégé-2000. (2005) URL http://protege.stanford.edu

Pike, R. (1989) Notes on Programming in C. URL http://www.lysator.liu.se/c/pikestyle.html

Webopedia (computing dictionary) (2005) URL http://www.pcwebopaedia.com/