Verification and Validation of Temporal Business Rules

Babis Theodoulidis, Petros Alexakis and Pericles Loucopoulos

Information Systems Group
Department of Computation
UMIST P.O. Box 88
Manchester M60 1QD
email: (babis, APetros, pl)@uk.ac.umist.co.sna

Abstract

In this paper the development of an analysis tool for verification and validation of temporal business rules is described. The formalism for expressing the business rules is the Conceptual Rule Language developed as part for the ESPRIT II project TEMPORA.

1. Introduction

The TEMPORA project advocates the use of temporal business rules as the most effective way for business modelling [TEMPORA 1991a; TEMPORA 1991b]. The TEMPORA conceptual modelling formalism consists of three integrated models that capture all the necessary information about the business domain. The Entity Relationship Time (ERT) model deals with the structural aspects, the Process Interaction Diagram (PID) that deals with the behavioral aspects and the Conceptual Rule Language (CRL) that deals with the modelling of business rules that apply to either the structural aspects or the behavioral aspects of the business domain.

In this paper, the development of an analysis tool for the verification and validation of business rules is discussed. The architecture of the overall environment is depicted in figure 1. As shown in this figure, the information captured by the corresponding capture tools is stored in the specification repository. The analysis tools interact with the specification repository in order to provide a full completeness and consistency checking of the business knowledge.

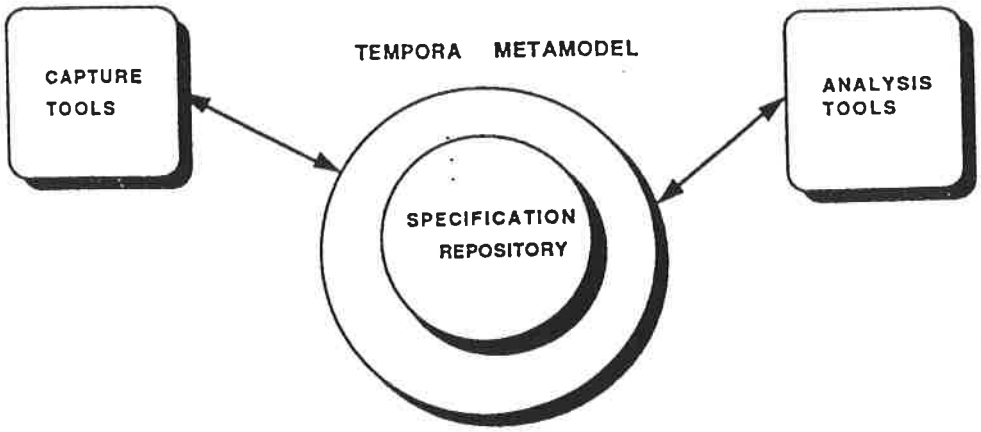


Figure 1. The overall architecture of the system

The specification repository is built on top of the Megalog deductive database management system. In section 2, a brief description of the representation of the metamodel in MegaLog is given. Section 3 overviews the CRL language and discusses its functionality with regard to the modelling of temporal business rules. In Section 4, a detailed description of the consistency and completeness checkings that can be discussed is exemplified. Finally, section 5 concludes the paper and discusses directions for future work.

2. Metamodelling and representation in MegaLog

A metamodel is a formal description of a modelling technique together with axioms that can be used for checking whether the application is well formed or not. A metamodel is specification-independent and time invariant in the sense that it is only developed once for a specific technique and can be using for verifying the various application models developed using this technique [Brinkkemper 1990].

A metamodel should consist of the following:

- The basic concepts of the model that will be used for describing a specific application and their interrelationships. For example, if the candidate model is the Entity-Relationship model then its metamodel should contain the concepts of entity, relationship and attribute.

- Axioms that constraint the handling of the basic concepts so as to ensure the correctness and consistency of the developed specification. For example, if the candidate model is the Entity-Relationship model then its metamodel should contain the axioms such as "An attribute should always be related to an entity" and "Entity names are unique".

- A set of basic functions that describe the possible ways that a metamodel can be instantiated in order to create the application model. For example, if the candidate model is the Entity-Relationship model then its metamodel should contain the function *create_entity*.

The approach described in this paper implements the TEMPORA metamodel in the deductive database management system MegaLog. MegaLog is a programming environment for building Database and Knowledge Base Management systems.

Megalog integrates a knowledge base with a logic programming language to provide large scale persistent storage of knowledge in a way that it can be efficiency accessed and processed by logic programs. Some features of Megalog are [Orsfield et al 1990]:

- Efficient retrieval of knowledge for any size of knowledge base.

- Deductive rules can be stored in the database.
- Complex structures and lists are valid data types for the database.
- Database transparency. When making a query the user does not have to specify whether the data is stored in main memory or in the database.
- The database can be queried with either set-oriented or single-tuple operations. Backtracking can be used to navigate through the database and extract all solutions to a goal.
- Incorporates standard Prolog syntax and semantics.
- Provides a large range of built-in predicates.

There are a number of reasons why Megalog was selected to implement the domain metamodel. These are as follows:

- Metamodelling is a rule intensive process. This means that it is rich in consistency axioms and constraints and that consequently, an appropriate formalism such as Megalog, is best suited for the explicit representation of model syntax and semantics.
- Metamodelling axioms are explicitly represented in Megalog and this provides the flexibility to easily maintain and enhance the metamodel by updating these axioms without the need to revise large chunks of metamodelling knowledge.
- An executable application model is of great help with respect to prototyping for verification and validation purposes. Megalog provides an appropriate platform for building executable specifications. In addition, in the context of TEMPORA, executable specifications facilitate the task of consistency and completeness checking of business rules.
- Megalog provides a natural platform for implementing TEMPORA applications. This means that the transformation of a domain specification to implementation constructs is a straightforward process that can be automated. On the contrary, employing a relational or object-oriented implementation platform complicates the transformation of the specification.

When using Megalogs' knowledge base to express the TEMPORA metamodel the syntax of a deductive relation is as follows:

relation _name <==> {Attribute_name1, Attribute_name2, Attribute_nameN}.

where

- The Relation_Name can be either an atom or a variable depending on whether the relation is to be permanent or temporary.
- Each Attribute_Name is an atom. It is preceded by a + when the attribute is to be included in the key.
- The attribute type and field length do not need to be specified, as Megalog allocates memory as required.

The domain metamodel in MegaLogs' Knowledge Base (MKB) is as follows ship is as follows:

- *relationship* <==> *[+rname1, +card1, +rname2, +card2,+nameA, +nameB, +concept_type, + time_mark, +relation_type]*
- *business_rule* <==> *[+rule_id, +description, +associates_to, +motivated_by, + motivates, +refers_to, +invokes_process, +rnumber, +formal_rule]*
- *entity_class* <==> *[+full_name, +symbol_text, +category, +minnumof_entities, +maxnumof_entities, +time_mark, +concept_type, +description, +identified_by, +refbyassocto_rules].*
- *value_class* <==> *[+full_name, +symbol_text, +category, +type_definition, +description, +relationshipvalue_classes, +used_by].*
- *process* <==> *[+process_id,+full_name, +symbol_text, +description, +information_requirements].*
- *pll_rule* <==> *[+process_id, +process_fullname, +If_part, +Then_part].*
- *signal* <==>*[+name,+type, +subtype, +from, +to].*

The *entity_class*, *value_class* and *relationship* predicates refer to the ERT. The *process*, *signal* and *pll_rule* predicates refer to the PID. Finally the *business_rule* predicate refers to the CRL. For the ERT schema given in figure 2, the relationship between the entity class EMPLOYEE and the entity class PROJECT can be expressed as follows:

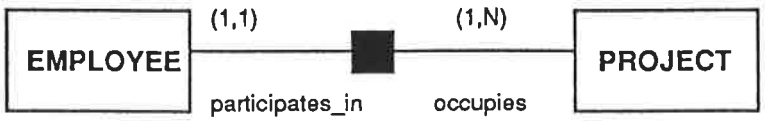


Figure 2. An ERT schema

relation("participates_in","(1,1)","occupies","(1,N)","EMPLOYEE","PROJECT","non_derived","N","Inter")

A number of metamodel axioms have been implemented in order to provide a complete syntax checking of the metamodel. These checks ensure for example, that the attributes of each relation must contain a set of predefined values. For example, the Time mark attribute of Entity must be one of the (S,M,H,D,Mo,Y,T), otherwise the syntax checker will reply :"The time mark must be one ..."

As shown in figure 3, the metamodelling information is classified into two categories, formal and informal. By the definition of "formal", it is meant that the information must be provided always and in addition, it should satisfy all the model axioms. For example, a formal metamodelling information for the ERT model is that "Each entity must have a name". On the other hand "informal" means not obligatory in the sense that this information is not necessary but only desirable, for the definition of the application model. This type of metamodelling information is usually a piece of text which defines related information like the name of the application developer, the date of update etc.

	entity class	value class	business rule	process	pll_rule	relation	signal
Obligatory	full_name symbol_text minimum numof_ entities maximum numof_ entities time_mark concept_type identified_by refbyassocto _rules	full_name symbol_text category type_definition relationship value_classes used_by	rule_id category associates_to motivated_by motivates refers_to invokesprocess formal rule	id full_name symbol_text information_ requirements	rule_order process_id pfull_name formal_rule	name1 card1 name2 card2 nameA nameB concept_type time_mark relation_ type	name type subtype from to
Mandatory	description_ inEnglish	description_ inEnglish	description_ inEnglish	description_ inEnglish			

Figure 3: Table of the informal and formal metamodelling information for the domain models

The syntax checker provides also *existence checkings*. For example, a relation or its unique identifier (name or id number) that already exists in the repository can not be stored. For example, consider the following metamodelling functions:

insert_entity("Employee", "Employee", "1","N","T", "An employee..","has NAME","R28").

message:"This entity already exists in this database"

insert_pll("2", "P2.4", "Register@payment", "If register...").

message:"There is no process with this id in the database"

The syntax checker provides also existence checkings. For example a relation or its unique identifier (name or id number) that already exists in the repository can not be stored [Alexakis and Theodoulidis 1992].

3. The Conceptual Rule Language

The CRL language provides the means for controlling the behaviour of the business domain in terms of rules. The CRL rules are classified into categories according to their semantics. As shown in figure 4, the following categories of CRL rules are distinguished:

- *Constraint rules* which are concerned with the integrity of the ERT components. They are further subdivided to static constraint rules which are expressions that must hold in every valid state of an ERT database and transition constraint rules which are expressions that define valid state transitions in an ERT database. An example of an static constraint rule might be 'The number of employees working in a department must be less than 100 at all times'. An example of a transition constraint rule might be 'The salary of an employee must never decrease'.

- *Derivation rules* which are expressions that define the derived components of the ERT model in terms of other ERT components including derived components. There must be exactly one derivation rule for each such component. As the constraint rules, derivations rules are also subdivided to static derivation rules and transition derivation rules depending on whether the derived ERT component is timestamped or not. An example of a static derivation rule might be 'A supplier is the cheapest supplier for a particular product if his offer for this product has the minimum price'. An example of a transition derivation rule might be 'A customer is the best customer of this month if the total amount of his orders placed this month is the maximum'.

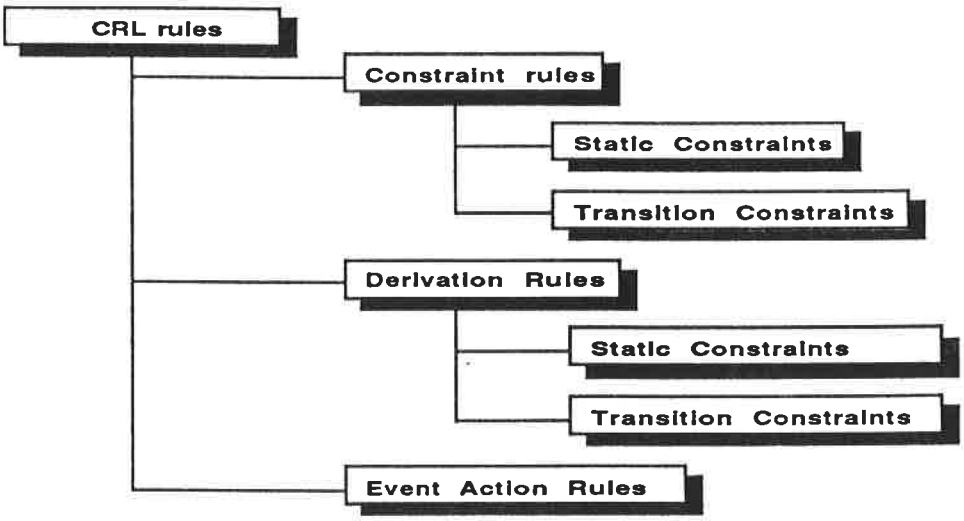


Figure 4. Classification of CRL rules

- *Event-Action rules* which are concerned with the invocation of procedures. In particular, action rules express the conditions under which procedures are considered fireable i.e., a set of triggering conditions and/or a set of preconditions that must be satisfied prior to their execution. An example of an action rule might be 'When the stock of a product falls below the reorder quantity level specified for this product then execute the reorder procedure immediately'.

The constraint and derivation rules can be further subdivided as shown in figure 4, to static and transition rules. The static constraint rules can be further subdivided as shown in figure 5 into:

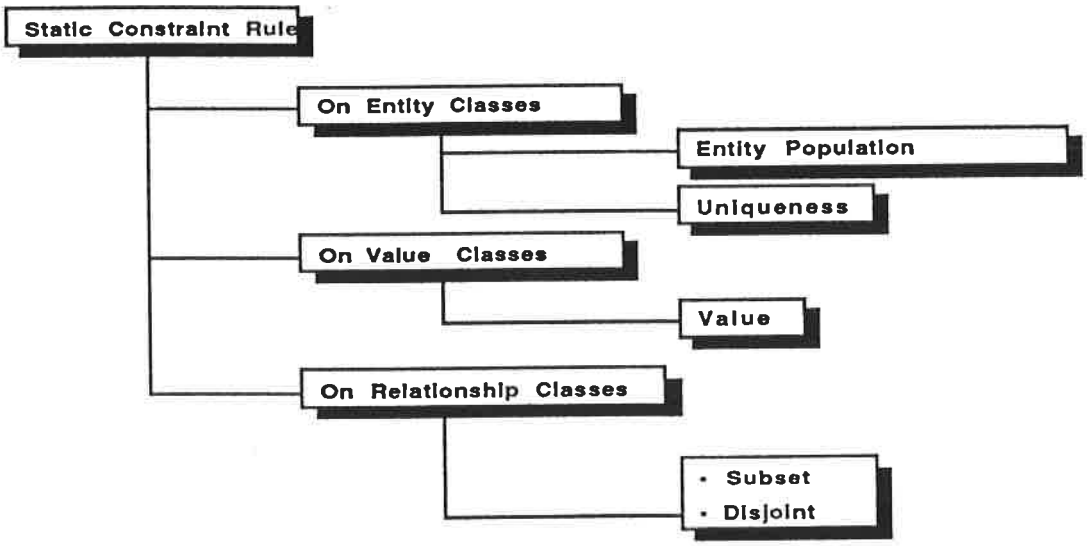


Figure 5. Classification of Static constraint rules

Each subcategory has its own syntax and semantics and as result, different possible errors have to be checked for during verification and validation.

4. Verification and Validation

Verification and validation mechanisms play a major role in any systems development approach. Their purpose in the information system development lifecycle is to ensure the development of the right system for the right problem. More specifically,

- The purpose of verification is to ensure that a specification is complete and formally correct [Bubenko 1988].
- The purpose of validation is to ensure that a specification really reflects the user needs and his/her intended statements about the application [Bubenko 1988].

From these we can conclude that verification means to check the syntax and semantics of a specification, whereas validation is the effort to ensure that the specification does what it is supposed to.

Verification can be divided into to types: syntax checking and checking of semantic consistency. In some cases, it may appear hard to distinguish the two kinds of checking. In general however, the verification of a model consist of the following:

- Ensuring usage of legal combinations of symbols, that is, check if the primitives (eg: entities, or processes, etc.) used, belong to the specific model. This is part of the syntax checking.

- Ensuring usage of legal combinations of symbols, that is, make sure that more complex constructs are formed from legal primitives and by applying the grammar rules of the specific model. This is also considered syntax checking.

The second process yields a model constructed by legal expressions. It does not check whether these legal expressions have been correctly put together in forming the specification. In order to do this, one has to apply semantic verification which goes beyond the detection of erroneous application of the grammar. Semantic checking is concerned with checking whether there are any contradictions or redundancies in the specification and comes very closer to the process of validation.

The major reason behind errors detected by verification is limited knowledge and ad hoc use of the selected models. The usual case is that designers subjectively apply the modelling techniques they have in order to develop a specification and the result is that they usually come up with inconsistent and/or incomplete schemata. This situations is even more frequent when more than one modelling techniques are used. In such case the risk of producing inconsistent schemata is very high and the designer should detect these inconsistencies before proceeding to the next design phase. In order to minimise the informal and heuristic application of a specific modelling formalism for the development of specifications and assist the process of developing models of high quality, knowledge about the formalism should be made explicit and formal. If the modelling languages are themselves formalised, then the inevitable subjectivity of modelling [Falkenberg 1989] can be restricted as much as possible.

The chosen approach, as described in section 3, is by means of metamodelling in order to accomplish the above specified requirements.

As far as validation is concerned on the other hand, there seem to be as many approaches to validation as there are developers, thus ad hoc methods are widely used. The validation effort is very much depended on experience and luck.

The scope of validation embraces both methods and tools. In particular, the following kinds of support can be envisioned [Buberiko 1988]:

- paraphrasing of graphical specifications, logic-based rules and constraints in natural language (NL).

- generation of abstractions and abstracts of (parts of) specifications.

- support of reasoning about the specifications eg: answering "what if X?", "why X" type of queries directed at an existing specification.

- animation or simulation (or symbolic execution) of (parts of) a specification.
- prototyping facility which interprets (parts of the executable specifications).
- assistance in classification and concept formation (requires domain knowledge).

With regard to the CRL language, the following checkings for temporal business rules are performed [Alexakis & Theodoulidis 1992]:

i) *Redundancy checking* in order to deduce whether any business rule is the logical consequence of one or more business rules.

For example, assume that we have the following rules:

R_1: if A_1 then X_1
R_2: if A_2 then X_2
R_3: if A_1 and A_2 then X_1 and X_2

In this case, R_3 is redundant.

ii) *Conflict checking* in order to deduce whether two or more business rules have conflicting outcomes.

For example, assume that we have the following rules:

R_1: if A then Z_1
R_2: if A then Z_2

In this case, R_1 and R_2 are conflicting rules.

iii) *Subsumption checking* in order to deduce whether two rules have the same outcome but the one has additional premises.

For example, assume that we have the following rules:

R_1: if A_1 and B_1 then X_1
R_2: if A_1 then X_1

In this case, R_2 is subsumed by R_1.

If we follow a decision table-like approach for consistency checking of the rule base, then the complexity of the resolution algorithm will be exponential [Nguyen et al 1985]. That is highly inefficient. A medium size case study, as in [TEMPORA 1991c], will have about sixty business rules that must be checked. Checking rule by rule amounts to 2^{60} checkings which is clearly not acceptable.

The approach discussed in this paper deals with that problem by trying to decompose the rule base according to the corresponding business rule categories. Following this approach the search space of the resolution algorithm is greatly reduced since we only deal with

business rules that refer to the same conceptual object e.g., 'take all the value constraints that refer to the value class SALARY'.

Business rule categories are distinguished by *keyword detection*. For example, event-action rules should always have a *When* part that defines the triggering condition whereas derivation rules have a dinstinct *is-derived-as* expression [Theodoulidis et al, 1991b].

The constraint rules are further subdivided by keyword detection in their syntax or ERT component recognition (bold letters indicate detected keywords). For example:

Value constraint

<value_class_name> <role_name> <static_class_ref> <compar_operator> <arithm_expr>

or

<value_class_name> <role_name> <static_class_ref> <compar_operator> <string>

Population constraint

number_of (<static_class_ref> {set_operator><set_expr>}) <compar_operator> <arithm_expr>

Subset relationship constraint

<static_class_ref>{<set_operator><set_expr> **is_subset_of** <set_exp>

Disjoint relationship constraint

<static_class_ref>{<set_operator><set_expr> **is_disjoint_from** <set_exp>

Uniqueness constraint

<static_class_ref>{<set_operator><set_expr> **is_identified_by** <set_exp>

An example of how the checkings for the event-action rules take place follows. The syntax of the event-action rules is

WHEN {Trigger}
<External signal> |
<Internal signal> |
<Clock condition>|
<ERT expression>|

IF {Condition}
<static_class_ref>{<set_operator><set_expr>

THEN {Process }

Those rules that have same **WHEN** parts and those rules that have same **THEN** parts are grouped and the appropriate tables are created. These tables will be used in order to detect inconsistency. The following checkings are proposed to take place for event-action rules.

- Trigger existence -> For WHEN part
- Process existence ->For THEN part
- ERT components existence -> For IF part when it exists
- WHEN A1 $\wedge$ A2 THEN X -> Error ; Conjunction is not allowed in When part
- Same WHEN parts, same IF parts (in terms of ERT expressions), same THEN parts -> Redundant rules
- Same THEN parts and additional constraints in WHEN and/or IF parts -> Subsumed rules

For example, consider the following event-action rules.

i]	WHEN Alpha	IF Ena and Dyo	THEN Omega
ii]	WHEN Alpha	IF Ena and Tria	THEN Ypsilon
iii]	WHEN Bita	IF Dyo and Tessera	THEN Omega
iv]	WHEN Gamma	IF Ena and Tessera	THEN Fi
v]	WHEN Epsilon	IF Dyo and Tria and Ena	THEN Omega
vi]	WHEN Alpha	IF Ena	THEN Fi
vii]	WHEN Delta	IF Tessera	THEN Ypsilon
viii]	WHEN Zita	IF Pente and Dyo	THEN Psi
ix]	WHEN Epsilon	IF Ena and Tessera	THEN Psi
x]	WHEN Gamma	IF Tria and Tessera	THEN Fi

The following tables will be created:

- Same **WHEN** part :

• table1	[i,ii,vi]	--> Alpha
• table2	[iv,x]	--> Gamma
• table3	[v,ix]	--> Epsilon
• table rest	[iii,vii,viii]	--> Bita,Delta,Zita

- Same **THEN** part :

• table a	[i,iii,v]	--> Omega
• table b	[ii,vii]	--> Ypsilon
• table c	[iv,vi,x]	--> Fi
• table d	[Viii,iv]	--> Psi
• table rest	[]	

The two groups of tables will be compared. When find a common pair of rules the IF part will be examined in order to decide whether the rules are redundant or subsumed.

table 1 ∩ table a --> ∅
table 1 ∩ table b --> ∅
table 1 ∩ table c --> ∅
table 1 ∩ table d --> ∅
table 2 ∩ table a --> ∅
table 2 ∩ table b --> ∅
table 2 ∩ table c --> [iv,x]
table 2 ∩ table d --> ∅
table 3 ∩ table a --> ∅
table 3 ∩ table b --> ∅
table 3 ∩ table c --> ∅
table 3 ∩ table d --> ∅

The rules iv and x might be redundant or subsumed. That depends on the IF part. The following cases are distinguished:

- If Ena = Tria --> Redundant
- If Ena ∈ Tria --> Subsumed
- If Tria ∈ Ena --> Subsumed
- If Tria ≠ Ena --> One can say that there is no problem. A warning message might be displayed.

The IF part, also called precondition, is the most complicated part of a rule, in terms of analysing and tracing its semantics. The are a number of possible ways to express an ERT expression and the Analysis tool must understand and formalise the information given in order to create a unique way of representing each rule. Let for example examine the expression:

EMPLOYEE works_for DEPARTMENT [has Department_name where Department_name = "Computation"] has Salary where Salary = "9000"

EMPLOYEE has Salary where Salary= "9000" works_for DEPARTMENT [has Department_name where Department_name = "Computation"]

The representation of these rule parts in a parse tree is shown in figure 6. This representation results in an easy way of understanding the semantics of each rule:

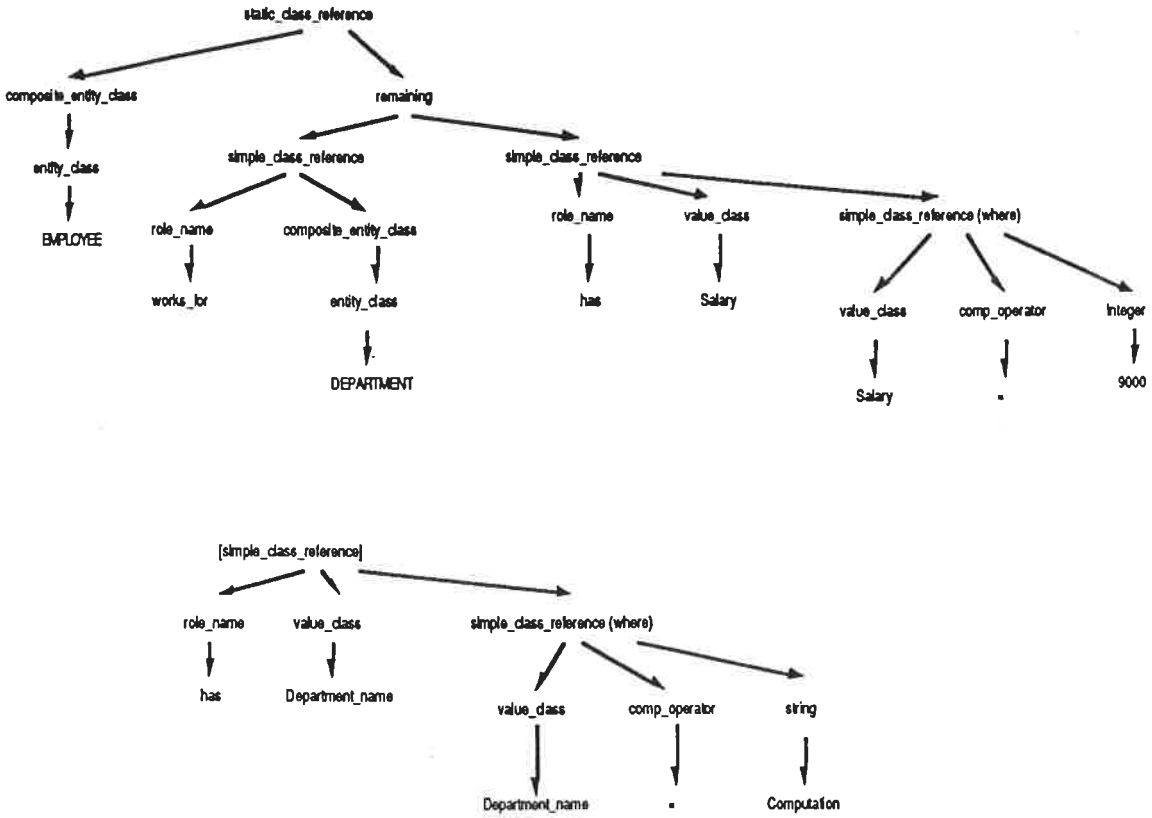


Figure 6. The parse tree of the precondition of an Event - action rule

In this parse tree every component of the Event-acion rule is a unique subtree. This means that *if the order of the rule components is changed and all the subtrees are still the same in a different order* then the semantics of this rule are the same. If one and only one subtree has been changed then the meaning of the rule is different.

A second example of how the checkings for value constraints take place is given below. The important point in Value constraint rules, apart from their ERT expression part (static_class_ref), is the comparison part. Here the **range** (>9000 and <10000) that are defined in each rule have to be examined. When comparing two rules the **intersection** of these spaces (Figure 7) is of crucial importance in order to decide whether they are conflicting, redundant or subsumed.

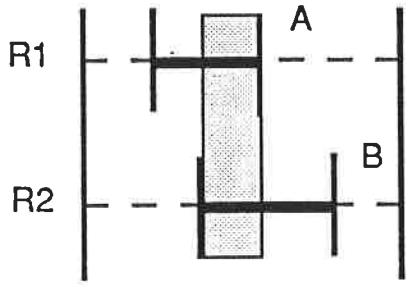
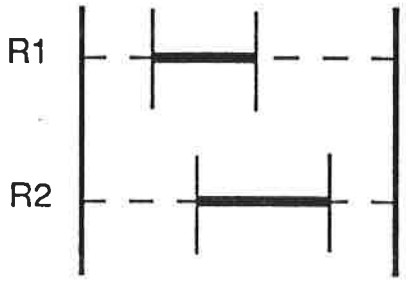


Figure 7. The intersection between spaces A and B ($A \cap B$)

Like in Event-Action rules the appropriate tables should be created and the checkings that have to be done are:

- For arithmetic and string comparison part:

  - Two rules with same Value class and same (identical) comparison part.



  - The one has additional constraints :
    R1: X <role_name> A1 <comp_operator> T
    R2: X <role_name> A1 ∧ A2 <comp_operator> T --> **Subsumed**

  - The one has more information :
    R1: X <role_name> A1 <comp_operator> T
    R2: X <role_name>A1 ∨ A2 <comp_operator> T -->**Redundant**

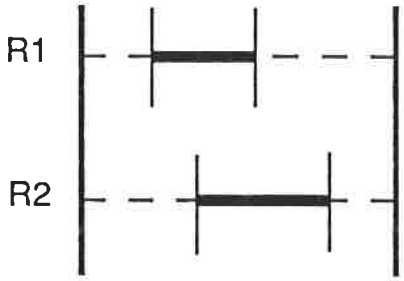
- For arithmetic comparison part only:

  - Two rules with same Value class.

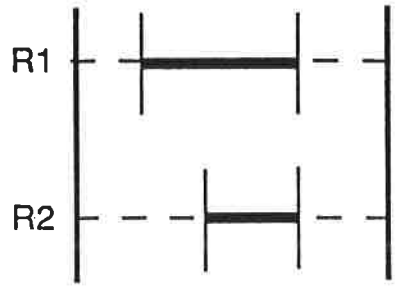
  - Same ERT part and different comparison part
    (T1 ∩ T2 = ∅)
    R1: X <role_name> A1 <comp_operator> T1
    R2: X <role_name> A1 <comp_operator> T2 --> **Conflicting**

  - The one has more information in such_as part
    R1: X <role_name> A1 <comp_operator> T1
    R2: X <role_name> A1∨ A2 <comp_operator> T2 -->

  i] T1 ∉ T2 --> **Conflicting**



  ii] T1 ∈ T2--> **Redundant**



- For string comparison part:

  •Two rules with same Value class.

  • Same ERT part and different comparison part
    R1: X <role_name>A1 <comp_operator> T1
    R2: X <role_name>A1 <comp_operator> T2 --> **Conflicting**

  • The one has more information in such_as part
    R1: X <role_name>A1 <comp_operator>T1
    R2: X <role_name>A1v A2 <comp_operator>T2 --> **Conflicting**

5. Conclusions

The current work is to complete the implementation of the Analysis Tool. The future work is to detect additional conditions that need to be checked in order to create a system that can trace all the possible common user errors. For example transition rules where the notion of time has to be detected and analysed.

References

[Alexakis and Theodoulidis 1992] Alexakis ,P., Theodoulidis, B., *The TEMPORA Deductive Repository Metamodelling and Validation Support* , In Proc. Workshop on Next Generation CASE Tools, NGCT'92, Manchester.

[Brinkkemper 1990] Brinkemper, S., *Formalism of Information Systems Modelling*, Katholieke Universiteit te Nijmegen, Thesis, 1990.

[Bubenko 1988] Bubenko, J. A.,*Selecting a Strategy for Computed-Aided Software Engineering (CASE)*, Syslab Report Nr 59,1988.

[Nguyen et al 1985] Nguyen, T.A., Perkins, W.A., Laffey,T.J., Perora,D., *Checking an Expert System for Consistency and Completeness*,Proceedings of 91h IJCAI, Vol1, 1985.

[Orsfield et al 1990] Orsfield, T.,Boca, J., Dahmen, M., *MegaLog User Guide*, 1990.

[TEMPORA 1991a] TEMPORA Consortium, *Concepts Manual*,1991.

[TEMPORA 1991b] TEMPORA Consortium, *Implementation Manual*,1991.

[TEMPORA 1991c] TEMPORA Consortium, *Swedish Post Case Study*,1991.

[Theodoulidis et al 1991a] Theodoulidis, C., Loucopoulos, P., Wangler B., *A Conceptual Modelling Formalism For Temporal Database Applications,* Information Systems, Vol 16, No 4, 1991.

[Theodoulidis et al 1991b] Theodoulidis, C., Loucopoulos, P., Wangler B., *The Entity Relationship Time Model and the Conceptual Rule Language,* 10th Conference on the Entity Relationship Approach, San Mateo, CA, October 1991.