# CONSTRAINTS IN A HYBRID KNOWLEDGE REPRESENTATION SYSTEM

*H.W. Guesgen, U. Junker, A. Voss*

Expert Systems Research Group
Gesellschaft fuer Mathematik
und Datenverarbeitung (GMD)
Postfach 1240
D-5205 Sankt Augustin 1, F.R.G.

**Abstract**

In our research group, the hybrid knowledge representation system Babylon has been developed providing formalisms for rules, prolog and frames. Beyond it, we implemented Consat, a system for constraint satisfaction. Since applications of Babylon for process diagnosis, planning etc. required constraints, we integrated Consat into the Babylon environment.

The paper describes the integration of Consat into Babylon, regarding two aspects: First, constraints should be available as another Babylon formalism by using the functional interface of Consat. On the other hand, it is important to have constraints implicitly controlling other Babylon formalisms, for instance, in order to keep the system's database consistent. While with respect to the first point, the paper describes work already finished, the second form of integration is work in progress.

**Keywords**

Constraints, hybrid knowledge representation, integration.

## 1. Introduction

Constraint networks are evolving as another style of knowledge representation beside rules, logic, frames, etc. in many domains such as qualitative reasoning and planning. A hybrid knowledge representation system called Babylon has been developed in our group at GMD Sankt Augustin [Forschungsgruppe Expertensysteme 1985], [diPrimio, Brewka 1985], that - so far - provides rules, prolog, and frames. However, Babylon is open to integrate any other formalisms which can be achieved by simple extensions of the system. Currently, in our group a planning system, a knowledge acquisition system [Diederich et al. 1986], and a system for process diagnosis are being developed, all of which depend on the availability of some constraint mechanism (cf. Molgen [Stefik 1981] as a planning system using constraints, and [Davis 1984] and [deKleer, Williams 1986] as diagnosis systems using constraints). Thus, there is an urgent need to extend Babylon to a constraint formalism.

Independently from Babylon but with the purpose of later integration, a constraint system called Consat [Guesgen 1986] has been developed. Consat allows to define hierarchical networks of domain independent constraints. The variables in a network can be assigned sets of values that are propagated to compute the maximal locally consistent assignment, the globally consistent assignments, or to detect an inconsistency in the original assignment.

Due to the open-endedness of Babylon, integration of Consat as another formalism was straightforward. As a result, rules, prolog goals, frame instances with their slots and behaviors can be used in the definition of constraints. Vice versa, rules, prolog clauses, frames and behaviors can use constraint networks for the computation of relations or the detection of inconsistencies with respect to some given values.

Apart from having constraints symmetrically to the other formalisms, we would like to use them asymmetrically as watching over the other formalisms:

Babylon has a dynamic database containing frame instances, items produced by rules, and facts asserted by prolog. Insofar as these data may be interpreted as values of certain variables, the variables could be identified with the variables in some constraint network. Now if all data changes are implicitly passed to the constraint network, propagated therein, and the results are implicitly returned to the database, we use the constraint network to watch the data of the other formalisms.

This asymmetric use of constraints saves much explicit programming needed to maintain the consistency of the database. It is particularly attractive in a hybrid system where very different types of data as produced from the different formalisms have to be maintained consistent.

So far we know only one other hybrid system that provides constraints as a real knowledge representation formalism. It is the Socle system that combines constraints with a frame formalism [Harris 1986]. Other systems like ART, KEE, or Knowledge Craft use constraints only limited to the scope of single frame instances in order to express context sensitive restrictions on their slot values (cf. [Richer 1986]).

The integration of Consat in Babylon has been completed with respect to the symmetric case and has been delivered as an experimental feature of Babylon release 2.0 in January '87. Immediately afterwards, we will implement the asymmetric integration for which currently only a restricted prototype with an interface to the frame formalism has been implemented [Junker 1986].

## 2. Two systems as a starting point

Before discussing the integration of Consat into Babylon we will consider the two systems separately in some more detail.

### 2.1. The constraint system Consat

Consat is a domain-independent constraint satisfaction system [Guesgen 1986] allowing to propagate arbitrary symbolic values.

A constraint establishes a relation between certain variables. In Consat, the relation may be defined *extensionally* by enumerating all tuples. The following example is extracted from the description of traffic lights with two fires:

```
(defconstraint
    (:name inverse-state)
    (:type primitive)
    (:interface fire1 fire2)
    (:relation (:tuple (on off)) (:tuple (off on)))).
```

In Consat, constraints can also be defined *intensionally by functional expressions* for all variables in terms of the other variables as in the following example:

```
(defconstraint
  (:name sum)
  (:type primitive)
  (:interface addent1 addent2 sum)
  (:relation
    (:pattern (addent1 addent2 (+ addent1 addent2))
        :if (constrained-p addent1 addent2))
    (:pattern (addent1 (- sum addent1) sum)
        :if (constrained-p addent1 sum))
    (:pattern ((- sum addent2) addent2 sum)
        :if (constrained-p addent2 sum)))).
```

The condition following :if ensures that, e.g., (+ addent1 addent2) is computed only if the arguments are assigned a set of definite values. Thus, the functional patterns may be conditioned. In extreme cases, the condition may play the role of a *characteristic predicate* for the relation:

```
(defconstraint
    (:name greater)
    (:type primitive)
    (:interface x y)
    (:relation (:pattern (x y) :if (< x y)))).
```

Constraints can be composed to hierarchical networks of constraints. Figure 1 sketches the traffic lights of the crossing between 5th avenue and 32nd street.

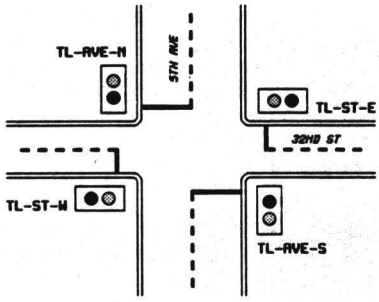


Fig. 1: Crossing between 5th avenue and 32nd street.

The corresponding constraint network can be defined using eight variables, one for each fire:

```
(defconstraint
    (:name traffic-lights)
    (:type compound)
    (:interface
        tl-ave-n-red tl-ave-n-green
        tl-ave-s-red tl-ave-s-green
        tl-st-e-red tl-st-e-green
        tl-st-w-red tl-st-w-green)
    (:constraint-expressions
        (inverse-state tl-ave-n-red tl-ave-n-green)
        (inverse-state tl-ave-s-red tl-ave-s-green)
        (inverse-state tl-st-e-red tl-st-e-green)
        (inverse-state tl-st-w-red tl-st-w-green)
        (same-state tl-ave-n-red tl-ave-s-red)
        (same-state tl-st-e-red tl-st-w-red)
        (inverse-state tl-ave-n-red tl-st-e-red))).
```

Constraints have been used to filter sets of values assigned to the variables (cf. Waltz's filtering in scene analysis [Waltz 1972]), and to compute new values from given ones as, e.g., in electrical circuit analysis [Stallman, Sussman 1977]. Consat supports both aspects by allowing sets of values to propagate in two modes (terminology is taken from [Voss 1987]):

1. Given the definition of a constraint network and an initial assignment of value sets to the variables of the network, the system can compute the *maximal locally consistent assignment* by local propagation. For example, the expression

```
(satisfy traffic-lights :locally
    :with tl-ave-n-green = 'on)
```

evaluates to

```
((tl-ave-n-red off) (tl-ave-n-green on)
 (tl-ave-s-red off) (tl-ave-s-green on)
 (tl-st-n-red on) (tl-st-n-green off)
 (tl-st-s-red on) (tl-st-s-green off)).
```

This is a unique assignment which describes the situation already shown in figure 1.

If one of the variables is assigned the empty set as a result of local propagation, the network is inconsistent with respect to the initial assignment.

2. Given a network and an initial assignment to the variables of the network, the system can compute one, a specific number or all *globally consistent assignments* by local propagation, tentative assumptions and backtracking. If such an assignment does not exist, the initial assignment following the keyword :with has been inconsistent and nil is returned. In the preceding example, the globally consistent solution is identical with the locally consistent solution, because all variables are uniquely determined. In general, a locally consistent assignment is only an approximation containing the globally consistent solution (cf. [Freuder 1978]).

Summarizing, we observe that Consat supports a purely functional view: After computing the final assignments for the variables, all values are forgotten. In contrast to [Steele 1980], there is no way to restrict and propagate values in the network incrementally. As a consequence, no reason maintenance system is needed in Consat. With regard to the integration of Consat in Babylon, we wanted to avoid double work: In Babylon, we can use frame instances to record the state of the network.

### 2.2. The Babylon philosophy of integration

Babylon is a knowledge representation system that is open to integrate arbitrary knowledge representation formalisms [diPrimio et al. 1986],[Christaller et al. 1986]. This open-endedness is achieved by having separate interpreters for each formalism which do not know from one another. Instead they know a meta interpreter that is responsible for all communications.

So far, Babylon provides a frame formalism with multiple inheritance and active values, prolog, and a

rule formalism without variables but with optional forward and backward chaining strategies.

In order to integrate a new formalism like constraints or a database language, we just have to provide an interpreter for the new formalism and inform the meta interpreter in a so-called processor mixin about the new types of Babylon expressions to be encountered in the other formalisms.

## 3. Integration of Consat in Babylon

Planning to integrate constraints in Babylon, there was an obvious solution offered by Babylon's open-endedness to new formalisms. We will first discuss this solution of adding constraints as another formalism before we turn to an alternative where we have constraints silently controlling the other formalisms.

### 3.1. Constraints as another Babylon formalism

According to section 2.2, we can integrate the constraint formalism into Babylon by

- establishing the Consat interpreter as a new language processor,
- introducing satisfy expressions calling Consat's two propagation functions as new types of Babylon expressions.

Now, we can use constraints in rules, prolog clauses, and behaviors of frames. For example, we could write a rule reacting to the inconsistent assignment detected by the traffic-lights network from section 2.1:

```
(($and (not (satisfy traffic-lights :globally
                    :with tl-ave-n-green = 'on
                          tl-st-e-green = 'on))
        ...further conditions...)
  ($execute (handle-inconsistency))).
```

The consistency check is a meaningful application of constraints in rules. Besides, in the prolog formalism we can bind the result of constraint propagation to prolog variables for further processing in that formalism:

```
(is _va (satisfy traffic-lights :locally
                :with tl-ave-n-green = 'on)
(candidate _va   _tl1 = tl-st-e-red
                 _tl2 = tl-st-e-green)
```

In the first clause, the variable _va is bound to the maximal locally consistent assignment, which is used in the candidate clause to assign the elements of the value sets successively to the specified prolog variables. In our example, the candidate clause results in binding _tl1 to on and _tl2 to off. If backtracking occurs, the candidate clause would result in fail, since the maximal locally consistent assignment is unique.

Beside using constraints in other formalisms, we can use the other formalisms in order to define constraints. In particular, we can use slots and behaviors of frames in the functional expressions of the :pattern construct, and prolog goals as conditions following the keyword :if.

### 3.2. Constraints watching other Babylon formalisms

The integration of constraints as a new Babylon formalism is not satisfactory due to the functional interface of Consat. Thus, it is difficult to impose any constraints on the slot values of a frame instance (as e.g. in ART, KEE, or Knowledge Craft), because these constraints should be attached permanently to the slots and they should be invoked implicitly whenever a slot value is about to change. For example, we would like to define a frame traffic-light with two slots for the fires and the inverse-state constraint keeping them consistent.

We could extend this example to a crossing frame with four slots for the four traffic lights (and another slot to remember the time of the last switch). To keep the traffic lights consistent, we would like to attach traffic-lights-constraints similar to the traffic-lights constraint in section 2.1 to all instances of the frame.

```
(defframe traffic-light
   (slots (red-fire - :possible-values (:one-of on off))
          ...further slots...))

(defframe crossing
   (slots (tl-ave-n -
              :possible-values (instance-of traffic-light))
          ...further slots...))

(attach-constraints
 (:name traffic-lights-constraints)
 (:attachments
  (:for-all tl = (instances-of traffic-light)
       (inverse-state (tl red-fire) (tl green-fire)))
  (:for-all xing = (instances-of crossing)
       (same-state ((xing tl-ave-n) red-fire)
                   ((xing tl-ave-s) red-fire))
       (same-state ((xing tl-st-e) red-fire)
                   ((xing tl-st-w) red-fire))
       (inverse-state ((xing tl-ave-n) red-fire)
                      ((xing tl-st-e) red-fire)))))
```

Now, given the crossing 5th-ave-32nd-st from figure 2.1 as an instance of the crossing frame, what do we expect from attaching the traffic-lights-constraints? We could write some rules to regulate the traffic lights of the crossing depending on the cars in the street and in the avenue, and depending on the time elapsed since the last switch, for example:

```
(rule4 ($and (car-in-ave 5th-ave-32nd-st)
             (car-in-st 5th-ave-32nd-st)
             (green-phase 5th-ave-32nd-st avenue 1))
       ($execute
          (<- 5th-ave-32nd-st :let-thru 'street)))
```

The rule uses the prolog goals car-in-ave, car-in-st and green-phase, which reflect the situation in the avenue and in the street, respectively. When the rule fires, the expression (<- 5th-ave-32nd-st :let-thru 'street) is evaluated, executing the behavior :let-thru, which translates its parameter street into the traffic light tl-st-e. If the green fire is not yet on, it is switched, and all other fires are reset to undetermined, thereby triggering the traffic lights constraints. The new values - on for green-fire of tl-st-e and undetermined for all other fires - are propagated. The result, a definite state on or off for each fire, is assigned implicitly to all fires of all traffic lights in our crossing, so that a new consistent state is obtained.

In the context of Babylon, the idea of using constraints to watch the data of another formalism should be generalized in a way that is open to arbitrary other future formalisms of Babylon. The key of such a generalization is to introduce the new concept of a ***Babylon variable***. A Babylon variable is a construct in some Babylon formalism that can be meaningfully associated with a value. For example, a Babylon variable in the rule formalism could be any text like "Peter is sick" that can be asserted or

negated in the database and thus can be interpreted as a Boolean variable. A Babylon variable in prolog could be any fact that can or cannot be derived from the current database and thus can also be interpreted as a Boolean variable.

Allowing arbitrary Babylon variables in the attach-constraints construct, we can impose constraints between very different data emerging from the different formalisms.

Having attached a constraint network to certain Babylon variables, demons must be installed for each of them. This task should be left to the interpreters of the variables' formalisms according to the idea that the individual interpreters should not know each other. Whenever a Babylon variable shall be changed, its demon must inform the constraint interpreter - of course via the meta interpreter. The constraint interpreter then must collect the current values of all Babylon variables that are in the same network as the one to be changed. This is again achieved via the meta interpreter. The collected values are propagated to compute the maximal locally consistent assignment. This assignment determines a set of values for each Babylon variable in the network which is returned - via the meta interpreter - to the interpreters of the variables' formalism. Processing these values is again left to the individual interpreters since different reactions may be meaningful in different formalisms. For example, reactions to an empty assignment may be to suppress the intended change of values or to ask the Babylon user. Or, if local propagation returns not a simple value but a set of definite values, this set may simply be forgotten or stored in some appropriate place: In the case of frames, the set of values may be stored in the :possible-values property, which is available for each slot.

## 4. Conclusion

We have presented two systems, the constraint satisfaction system Consat and the hybrid knowledge representation system Babylon. There are two compatible ways of integrating Consat into Babylon: Firstly, as another knowledge representation formalism beside rules, prolog, and frames, and secondly, as a mechanism watching over the other formalisms. The integration is realized according to the philosophy of Babylon, so that the next formalism to be added to Babylon - say a database language - can at once use constraint expressions as a special type of Babylon expressions or can be watched by attached constraint networks.

We already mentioned that, so far, we only know of one other hybrid system that provides constraints as another knowledge representation formalism. But in the Socle system [Harris 1986], constraints are combined only with a frame formalism, not with rules or prolog as in Babylon. The constraint component of Socle differs from Consat in having networks with a state so that constraints can be stepwise restricted and retracted. This rises much more problems since both, the state of the network and that of the frame objects, have to be kept consistent. Integrating Consat into Babylon was easier due to the purely functional interface of Consat. On the other hand, being integrated into Babylon, Consat profits from all features offered in Babylon. For example, constraint networks can be made permanent by attaching them to permanent data from other formalisms, in particular to frame instances. As another example, a reason maintenance component, which is being planned for Babylon, will also be available for Consat in Babylon.

The integration of Consat in Babylon has already been accomplished for the symmetric case and is available since January 1987 in release 2.0 of Babylon. The asymmetric case has been implemented prototypically with an interface to the frame formalism only [Junker 1986]. Implementation of the full feature will start in January'87, parallel to the implementation of the process diagnosis system which we consider as a major application test of our integration of constraints in Babylon.

## 5. Acknowledgements

We thank all members of our group for their contributions to this paper. Special thanks to Ursula Bernhard who read former versions of the paper.

## 6. References

T. Christaller, E. Gross, B.S. Mueller, E. Rome, J. Walther. Softwareentwurf und Realisierung des Expertensystem-Werkzeugs BABYLON mit Hilfe objektorientierter Programmierung. Proceedings of the GI-86, 1986.

R. Davis. Diagnostic Reasoning Based on Structure and Behavior. Artificial Intelligence 24 (347-410), 1984.

J. Diederich, I. Ruhmann, M. May. KRITON: A Knowledge Acquisition Tool for Expert Systems. Proceedings of the AAAI-Workshop: Knowledge Acquisition for Knowledge Based Systems, 1986.

Forschungsgruppe Expertensysteme. BABYLON-Referenzhandbuch. GMD, 1985.

E. C. Freuder. Synthesizing Constraint Expressions. Communications of the ACM, Vol.21, No.11 (958-966), 1978.

H.W. Guesgen. CONSAT - Foundations of a System for Constraint Satisfaction. GMD, 1986.

D.R. Harris. A Hybrid Structured Object and Constraint Representation Language. Proceedings of the AAAI-86 (986-990), 1986.

U. Junker. Integration des Constraint-Systems CONSAT in das Werkzeugsystem BABYLON. GMD, 1986.

J. deKleer, B.C. Williams. Diagnosing Multiple Faults. Proceedings of the AAAI-86.

F. diPrimio, G. Brewka. Babylon: Kernel System of an Integrated Environment for Expert Systems Development and Operation. Proceedings of the Fifth International Workshop on Expert Systems and their Applications, Avignon, 1985.

F. diPrimio, G. Brewka, K. Wittur. Integration of Formalisms in Babylon. GMD, 1986.

M.H. Richer. An Evaluation of Expert System Development Tools. Expert Systems, Vol.3, No.3, 1986.

R. M. Stallman, G. J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. Artificial Intelligence 9 (135-196), 1977.

G. L. Steele. The Definition and Implementation of a Computer Programming Language Based on Constraints. MIT, 1980.

M. Stefik. Planning with Constraints (MOLGEN: Part 1). Artificial Intelligence 16 (111-140), 1981.

A. Voss, H. Voss. A Uniform View on Local Constraint Propagation Methods. Proceedings of the KIFS-87, 1987.

D. L. Waltz. Generating Semantic Descriptions from Drawings of Scenes with Shadows. MIT, 1972.