The Redux' Server

Charles Petrie
MCC Enterprise Integration Division
3500 West Balcones Center Drive
Austin, TX 78759
petrie@mcc.com

Abstract

Redux' is a subset of the full REDUX model[7]. The latter performs problem solving. In contrast Redux' does not and acts only as a decision maintenance server. It takes objects of types defined in an ontology of decision components and maintains dependencies between them. Redux' is domain-independent. The dependency relationships are maintained on the basis of proposition type and not content, except for some string matching. Redux' servers are proposed as a mechanism for federating heterogeneous design agents by encapsulating their design decisions within a simple model and providing coordination services, especially for design revision. This proposal is described within the context of the SHADE and PACT projects.

1 Introduction

The SHADE and PACT[10] projects take a federating[6, 9] approach to the problem of coordinating distributed design. Individual software systems, used by the people to accomplish their part of the design, are idiosyncratic and may not work with each other.1 Since it is usually impractical if not impossible to impose a unifying structure on all systems, the most reasonable solution is to provide a framework that allows systems to coordinate as needed, while allowing them to be unchanged locally. This should allow, for example, subcontractors to develop their sub-designs and coordinate with each other over a wide-area network.

An important objective is to reduce the semantic unification required to connect two software systems. Each system will say something about its domain that the other system should "understand" to some degree. For instance, one CAD system configuring components and another CAD system designing mechanical platforms may have to agree at least that a motor shaft is round and requires support. But the former system may not need to worry about the shaft weight and the latter may not need to know about motor voltage. So it is possible to minimize how much the two systems have to agree upon the semantics of the terms that each formally represents.

The PACT approach is for participating agents to communicate through a language, KQML[3], that specifies a small set of performatives, such as assert, retract, or query. This isolates the predicates on which the various agents must agree, and the degree to which the semantics are common. In an example from [2], one agent mentions the domain-specific predicates closed-form and pmx in an assertion encapsulated in a KQML message using the domain-independent performatives interested-in. This use of the performatives identifies at least two points at which participating agents must perform semantic unification, rather than trying to unify complete models. The general principle is that a standard communications language can help minimize semantic unification because it allows agents to specify the connection points between them.

However, the KQML performatives have weak semantics. No inferences can be drawn from the message types. An interested-in message says something about when messages should be sent to whom. But there is no theory of message types that would allow distributed agents to make value-added inferences.

What one needs is a model of interactions among systems that minimizes the domain knowledge required for agents to cooperate. REDUX[7] is a general model of design and planning that emphasizes the propagation of the effects of change. We propose that it can be used as a framework for communication between systems. To that end, we have extracted a subset, Redux', of the general model and show how it can be used to encapsulate systems in a way that pro-

---

1The general version of the problem of getting heterogeneous systems to cooperate in a larger task is known as Enterprise Integration.

---
vides significant functionality in return for a small requirement for formal structuring. If the requirement presented here is not small enough, then perhaps no general model is adequate for the problem of cooperating distributed design and planning.

The Redux theory is formally described below in Section 3. This formal description is intended for conversion to Ontolingua[5] or some other KIF[4]-based system for portability. The informal idea is that Redux provides a theory for determining the effects upon decisions of changing conditions and the making and revision of other decisions.

Redux is presented here as a server. Clients send it messages about decisions and related types of objects. These messages are sent either by the user or by daemons in application code. When selected lines of application code are executed, such daemons send the appropriate messages to the server. The output is other messages, representing the change propagation, sent to the user, or receptor daemons. This is shown schematically in Figure 1.

![Figure 1: Decision Server](image)

The input messages contain strings that are used by the server for object matching, but otherwise the server does not understand anything about the application domain. The result is that client applications are encapsulated by the Redux object types, of which there is (currently) a small number.

We describe here how such a service could be used to coordinate distributed work, as in SHADE and PACT. Instead of a central server, we propose that each user has a local copy of Redux and these local copies talk to each other, perhaps through PACT facilitators. In effect, each copy of Redux becomes an “agent in a box” for each user.² We begin with an example, loosely adapted from PACT and SHADE.

### 2 Example

In the PACT/SHADE planar manipulator redesign scenario, the motor used is discovered to be inadequate for a new load specification. A larger motor is substituted. The change is annotated in a journal. Somehow, change notifications must be sent to other engineers. To quote from [10],

One is for the manufacturing engineer responsible for milling the frame for the manipulator. The new motor has a larger shaft diameter, requiring a larger hole in the mounting bracket. The size of the hole must be increased in the CAD specification, and process planning tool invoked to verify that the specified bore hole is still within manufacturability limits.

The notification problem is determining who needs to know of what changes. SHADE proposes to solve the notification problem by allowing the individual users to write “relevance theories” that can infer that an agent should be notified of a given change. An example of such a theory is

```plaintext
if a component is replaced, then the features of the replacement part (weight, cost, etc.) potentially change.
```

REDUX provides a relevance theory that generalizes much of what might otherwise have to be repeated in specific theories.

### 2.1 REDUX Model

We illustrate the REDUX model with the PACT planar manipulator example. In what follows, we use the notation that ENG-n is an engineer, or an engineering software system, and RDX-n is the local copy of Redux, which is a server implementing a partial REDUX model. We will refer to the latter as computational “agents”.

We start with the power engineer who chooses a motor for the artifact. As shown in Figure 2, given a goal G1 of something like Choose Motor for planar manipulator PL-1, engineer ENG-1 makes a choice of motor-1. The result of this decision, say D11, is an assignment of motor-1 to the “motor slot” of the design and perhaps a subgoal, say G2; the design of the encoding electronics required for such a direct drive motor.

In addition, a decision may have a contingency associated with it. For example, in this case, the possible unavailability of the motor is an unexpected future

²The knowledge acquisition problem is difficult: a phased implementation is suggested in [9].
event that would automatically invalidate the choice of motor.

Decisions are also said to have an optimality. This depends upon the validity of the decision rationale: why one operator was picked over the others in a conflict set. This reasoning is generally domain-specific. Suppose there was one other possible choice, motor-2, and motor-1 was chosen because it was cheaper. This rationale may or may not be recorded. If it were, it would consist of the costs for the two motors and is indicated by the bold arrow in Figure 2.

This decision by ENG-1 is defined by the goal, the contingency, the rationale, the assignment, and the subgoal and is recorded in REDX-1 as a set of dependencies around the decision object D11. The engineer (or program) would generate such a decision by sending a message to the Redux' server. While we do not specify the command language here, an example message would be:

```
MAKE-DECISION
  goal: 'choose motor for planar manipulator'
  assignment: 'motor is catalog number 701'
  subgoal: 'design d-drive encoder for 701'
  contingency: 'catalog number 701 unavailable'
  rationale: ['cost of 701 is $200', 'cost of 702 is $250']
```

Notice that the strings encapsulated by the Redux' types (formally specified in Section 3) are arbitrary, and need not be as structured as suggested by the figures. The human or application program is free to say whatever makes sense, with the caveat that these strings may need to be unified with those of other agents, so that the simpler, the better.

2.2 Revision Services

The REDUX model determines what pieces of the domain model are used for what change propagation, and, thus, what revision services to provide to the design engineer. If REDX-1 learns that motor-1 is unavailable, then decision D11 becomes invalid and several inferences follow from the REDUX model. The assignment of this motor (and any other resulting from this decision) becomes invalid. Any constraint violations in which they participated become moot. The goal G1 is no longer reduced: ENG-1 should be notified of this effect so that the goal may be placed back on the problem solving task agenda. The choice of motor-1 is no longer an option when this task is reconsidered.

In addition, subgoal G2 becomes invalid. If this had been distributed as a subtask to electrical engineer ENG-2, then REDX-1 would notify REDX-2 that G2 was no longer valid. If it had not yet been reduced, REDX-2 would notify ENG-2 to remove it from the task agenda. If G2 had been reduced by, say, decision D21, then D21 becomes suboptimal and its subgoals invalid. Any assignments resulting from D21 are still valid, but the problem solver is informed that it should consider retracting D21.3

Alternatively, suppose that motor-1 continues to be available, but the costs of the two motors change. REDX-1 will notify ENG-1 that decision D11 may have become suboptimal and should be reevaluated. This is a local action and would not affect any other agent. Thus, the case of possible loss of optimality is treated by REDUX very differently than the loss of decision validity.4

Redux' will also detect domain-independent cases of optimality loss. For instance, if a goal becomes invalid, then the decision reducing it becomes suboptimal.

3It is easy to modify the REDUX theory so that D21 would automatically become invalid. Application studies by Juergen Pauldokai and Helmuth Ritzer at the Universität Kaiserslautern indicate that this stronger condition is sometimes desirable.

4However, in the general case, this metadecision should not be automatic. For instance, the effect of the suboptimal decision may only be to add a flange, but undoing the decision will mean undoing much design work.
timal, regardless of the decision rationale. There is also a special case of domain-independent optimality involving backtracking that is in which the REDUX model is especially useful. This is discussed below in Section 2.4.

2.3 Notification and Connection

To complete the PACT example, there is another engineer, ENG-3 that is designing the connecting rod between the motor and the manipulator. Let the goal of doing so be G3. Suppose that to make a decision reducing this goal, ENG-3 has to know the shaft diameter of the motor ENG-1 has selected because there must be a hole in the connecting rod to receive the motor shaft. PACT describes a reasonable way for these agents to proceed, once some humans have determined which agents need to talk and have manually performed the required semantic unification. In PACT, ENG-3 asks ENG-1 to tell it the identity of the motor selected when it is chosen. Then, ENG-3 uses this information to look up the shaft diameter in a parts catalog, maintained by yet another agent.

There is another general principle that is useful here: requests for information should be recorded and used for notification upon revision of the information. This is the basic principle behind the distributed truth maintenance system of [1]. In this case, the fact that ENG-3 needed the motor identity from ENG-1 implies that ENG-1 should always tell ENG-3 if the motor identity changes. There is no requirement for a domain-specific SHADE relevance theory that if a component changes, its features may change. ENG-3 does not have to go to the trouble of stating such an interest. The PACT architecture should have ENG-1 notify ENG-3 automatically whenever the motor identity changes.

There are several possible ways that the above notification principle could be structured in REDUX. The motor identity could be recorded as part of the rationale for decision D31 that reduces G3 into other goals and assignments. If D11 is ever invalided, then RDX-1 identifies the assignments affected. At least one of these assignments is the motor identity, motor-1. RDX-1 then notifies RDX-3 which informs ENG-3 that decision D31 to drill a particular size hole in the connecting rod may be suboptimal and need to be retracted.

For this distributed case however, let us use a dependent assignment. In this example, the size of the hole in the connecting rod is an assignment associated with D31 that must be valid to satisfy goal G3 as before. But unlike the usual case, this assignment depends for its validity on D11. The two Redux agents, RDX-1 and RDX-3, create a virtual link between this assignment and D11, as if that decision had made two assignments instead of one. Such a link would create a stronger dependency between the two systems. Now, if D11 became invalid, ENG-3 would be notified that goal G3 is no longer reduced or satisfied, as shown in Figure 3, because of the invalid dependent assignment.

2.4 Optimality and Backtracking

So far, the example covers the simple case in which ENG-1 retracts the choice of motor-1 and ENG-2 and ENG-3 are subsequently notified. This was the case described in [2] because the first motor choice was determined to be inadequate for the load. In REDUX, the inadequacy of the motor would be represented as a constraint violation. ENG-1 tells RDX-1, possibly through an application daemon, to reject D11 with a reason consisting of the propositions of the motor assignment, the constraint, and the load assertion. Because G1 is no longer reduced, RDX-1 prompts ENG-1 to rechoose a motor. When ENG-1 chooses, say, motor-2, RDX-1 records the decision rationale as dependent upon the rejection of D11. The new decision, D12, is now propagated just as D11 was, notifying ENG-3 of the new hole size of the connecting rod.

But this example can be made much more interesting, and possibly realistic, by supposing not that motor-1 was determined to be inadequate, but that it conflicted with some other decision. Suppose ENG-1 worries about the ability to actually manufacture parts and had decided (say in decision D41) to use some exotic metal for the connecting rod that can not accommodate the right shape and hole size for motor-1.

In the REDUX model, this is represented as a constraint violation. Global constraint satisfaction is a problem for which Redux servers offer little help. Constraints are that which are violated by assignments, but it is up to the problem solvers to detect constraint violations and say which assignments need to be rejected in order to resolve the violation. Redux

---

5Decreasing the human involvement in this process is a very difficult problem not addressed here.
6Unless the decision was already committed, perhaps by drilling a limited supply of material.
7This is an extension of the basic REDUX model of [7].
8G3 no longer has a supporting decision as defined in Definition 4 in Section 3.
9Rejection reasons are formally described in Definitions 5 and 6 in Section 3.
servers only retract the necessary decisions and propagate the consequences. Semantic unification of variables and values in assignments is the crucial problem and we can only assume that it has somehow been done here.

Some engineer must give some Redux a statement that some set of assignments are in conflict. The Redux servers can indicate which decisions are responsible and what the effects of rejecting any of them. Suppose in this case, the engineers agree to reject the current hole size with a command that combines the conflict and the resolution sets:

\[
\text{CONSTRAINT-VIOLATION-RESOLUTION}
\]

\[
\text{conflicts: } \{\text{``motor hole size is 3.2mm''}, \text{``connector rod is titanium''}\}
\]

\[
\text{culprits: } \{\text{``motor hole size is 3.2mm''}\}
\]

Because of the dependent assignment, RDX-I rejects decision \(D_{11}\), with motor-I. When ENG-I (perhaps sometime later) chooses motor-2, he may or may not include the choice of exotic metal (say, titanium) explicitly in the rationale for this new decision, but the above command will cause RDX-I to do so. The new decision \(D_{12}\) automatically includes in its rationale the rejection of \(D_{11}\). This rejection itself has a reason dependent on the choice of exotic metal, derived from the statement of the constraint violation. Both of these reasons are produced as part of the Redux model. This is explained in more detail in Section 3.3.

Now we have the special case of domain-independent optimality. The Redux servers may not really understand anything about the manufacture of the artifact, but dependencies have been established between the optimality of \(D_{12}\) and the validity of decision \(D_{41}\). If ENG-I ever rejects the use of the exotic metal, then RDX-I and RDX-1 conspire to notify ENG-I that decision \(D_{12}\) is now suboptimal and there is now an opportunity of improving the design by using motor-I.

The decision connections here are shown in Figure 4. The bold arrows represent reasons. The one from \(D_{41}\) to \(D_{11}\) represents the reason for rejecting \(D_{11}\), and the one from \(D_{11}\) to \(D_{12}\) the reason for the optimality of \(D_{12}\). When \(D_{41}\) becomes invalid, there is no longer a good reason for the rejection of \(D_{11}\), and thus no longer a good rationale for \(D_{12}\), which then becomes suboptimal.

However, ENG-I need not see all of these connections. If ENG-I is human, then an appropriate interface, such as one illustrated in Figure 5, may be provided. Such an interface would alert ENG-I to the opportunity and explain the reason for it to as deep a level as desired.

We now give the formal description of the model that performs the services in this example. The model consists of an ontology and a theory: a set of entailments that constrain the type relations as well as the
values of the attributes of instances of the types in the ontology.

3 Theory

All input to Redux' are commands with arguments that are objects of types, or alternatively, classes, defined below.

3.1 Ontology

Here we define seven Redux' types of objects. Each object o has an associated type T, denoted by o:T. There are slots (binary relations) associated with each class, which map the objects to values consisting of other objects.

The four types that carry special significance for Redux' are:

- <decision>,
- <goal>,
- <reason>, and
- <assignment>.

Domain-specific computations are captured as decisions that reduce goals by making assignments and new goals. Reasons are captured for decisions and their rejections.

There are also three more generic types of objects. The first generic type is <string>. Domain-specific knowledge may be represented as strings, which are then encapsulated in Redux' object attributes, or slots. String matching is assumed as a primitive operation and is indicated by “==”. The second generic type is
<proposition> limited to predicates defined in Definition 5. The third generic type is that of datum, which has but one slot, datum-value, with a single value of type string.

Decisions have the following slots:

<table>
<thead>
<tr>
<th>Slot Name</th>
<th>Cardinality</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;name&gt;</td>
<td>single</td>
<td>string</td>
</tr>
<tr>
<td>&lt;objective&gt;</td>
<td>single</td>
<td>goal</td>
</tr>
<tr>
<td>&lt;contingency&gt;</td>
<td>multiple</td>
<td>string</td>
</tr>
<tr>
<td>&lt;rationale&gt;</td>
<td>multiple</td>
<td>reason</td>
</tr>
<tr>
<td>&lt;assertion&gt;</td>
<td>multiple</td>
<td>assignment</td>
</tr>
<tr>
<td>&lt;new-goal&gt;</td>
<td>multiple</td>
<td>goal</td>
</tr>
<tr>
<td>&lt;rejection&gt;</td>
<td>multiple</td>
<td>reason</td>
</tr>
<tr>
<td>&lt;dependent&gt;</td>
<td>multiple</td>
<td>assignment</td>
</tr>
</tbody>
</table>

The relation name is a one-to-one function between decisions and the set of all strings. The following predicates take decisions as single arguments:

- <valid>,
- <optimal>,
- <rejected>,
- <best>,
- <committed> and
- <retracted>.

These predicates and the rest below in this section are defined in Section 3.2.

The type goal has the following slots:

<table>
<thead>
<tr>
<th>Slot Name</th>
<th>Cardinality</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;consequent&gt;</td>
<td>single</td>
<td>string</td>
</tr>
<tr>
<td>&lt;super-goal&gt;</td>
<td>multiple</td>
<td>goal</td>
</tr>
<tr>
<td>&lt;sub-goal&gt;</td>
<td>multiple</td>
<td>decision</td>
</tr>
</tbody>
</table>

The relation consequent is a one-to-one function between goals and the set of all strings.

The following predicates take goals as single arguments:

- <valid-goal>,
- <reduced>, and
- <satisfied>.

The type assignment has only one slot, <variable-value>, with a single value of type string. The predicate <valid-asg> takes assignments as a single argument. The type reason has just one slot, <conditions>, which is a set of propositions. The predicate <case> takes reasons as a single argument.

In order to discuss how decisions relate to one another, certain slot values are required so that the decisions are sufficiently defined.

**Definition 1 (Decision Completeness)** A decision is completely defined just when:

- There exists a unique value for name.
- There exists exactly one value for objective.
- There must be at least one value for assertion or new-goal.

### 3.2 Dependencies

The following describes the theory that defines relations among elements of the ontology that depend upon slot values and so determine Redux services. We begin with some convenient notation. We say there is a database S of datum strings such that \( s \in S \Leftrightarrow \exists d : d . \text{datum} \) such that \( \text{datum-value}(d) \).

A decision \( D \) is admissible iff there is no \( c \in S \) that matches one of its contingencies (no datum matches a contingency) and no contingency matches an assignment of any decision.

**Definition 2 (Decision Admissibility)**

\[ \forall D : \text{decision}, \text{admissible}(D) \Leftrightarrow \forall c : \text{string} \text{ such that}\]

- contingency(D, c), \( c \notin S \land \exists A : \text{assignment} \text{ such that}\]

  \( \text{variable-value}(A, c) \).

The second part of admissibility requires that no assignments resulting from any one decision invalidate the admissibility of any decision; i.e., proof of admissibility should not depend on any assignment. All conflicts should be represented in constraints. Thus decisions monotonically extend problem solving. Non-monotonicity results from either changes in the world that invalidate admissibility or as a response to a constraint violation that causes retraction.

Complete decisions are valid unless they have been retracted (not possible if they have been committed) or a contingency has occurred.

**Definition 3 (Decision Validity)**

The following are the validity relations for completely defined decisions:

---

3 Alternately, we could define a predicate, say <Told>, that takes strings as a single argument. The proposition Told(x, string) is true just when \( \exists d : \text{datum} \) such that \( \text{datum-value}(d) \), The difference is only in notation.
Goals are reduced, and ultimately satisfied, by decisions. Thus, reduction and satisfaction depend upon decision validity.

Definition 4 (Goal Satisfaction/Reduction)
The following are the reduction and satisfaction relations for goals:

- $\forall G, G': \text{goal, \hspace{1cm}} \text{supergoal}(G') \land \text{subgoal}(G G') \iff \exists D: \text{decision such that} \hspace{1cm} \text{objective}(D G) \land \text{new-goal}(D G') \land \text{valid}(D)$.

- $\forall G: \text{goal, valid-goal}(G') \iff \exists G: \text{goal such that} \hspace{1cm} \text{subgoal}(G G') \land \text{valid-goal}(G) \land \{ \text{consequent}(G' g) \land \text{valid-goal}(g) \} \in S$.

- $\forall A: \text{assignment, valid-asg}(A) \iff \exists D: \text{decision such that} \hspace{1cm} \text{assertion}(D A) \land \text{valid}(D)$.

- $\forall G: \text{goal, \hspace{1cm}} \text{decision, supporting-decision}(G D) \iff \text{objective}(D G) \land \forall A: \text{assignment such that} \hspace{1cm} \text{dependent}(D A), \text{valid-asg}(A) \land \forall A: \text{assignment such that} \hspace{1cm} \text{dependent}(D A), \text{valid-asg}(A)$.

- $\forall G: \text{goal, reduced}(G) \iff \exists D: \text{decision such that} \hspace{1cm} \text{supporting-decision}(G D) \land \forall G': \text{goal such that} \hspace{1cm} \text{new-goal}(D G'), \text{valid-goal}(G')$.

- $\forall G: \text{goal, satisfied}(G) \iff \exists D: \text{decision such that} \hspace{1cm} \text{supporting-decision}(G D) \land \forall G': \text{goal such that} \hspace{1cm} \text{new-goal}(D G'), \text{satisfied}(G')$.

Notice that the satisfaction entailment does not depend upon the validity of a decision but upon the status of its results. A decision may not be valid. But if its assignments are valid and the subgoals are satisfied by other decisions, then the decision is virtually valid; the effect is the same as if it were. Notice also that this entailment holds for a decision with no subgoals and only assignments. This is the leaf node for goal satisfaction.

Decision optimality depends upon having a valid reason for the decision rationale and none for the rejection. The conditions\footnote{This is negation by failure.} value of a reason consists of a set of propositions, treated like a conjunct. There are only three kinds of propositions that may be in this set. One is that some assignment is valid-asg. Another is that some decision is rejected. The other is that some string is in $S$.

**Definition 5 (Reason Validity)** $\forall R: \text{reason, case}(R) \iff \forall \rho \in \text{conditions}(R)$, \hspace{1cm} $\{ \rho = \text{valid-asg}(A: \text{assignment}) \lor \rho = \text{rejected}(D_k) \lor \rho = \exists s \in S \} \land \rho$.

The rationale or rejection being the case then determines decision optimality.

**Definition 6 (Decision Optimality)** The following are the optimality relations for completely defined decisions:

- $\forall D: \text{decision, rejected}(D) \iff \exists R: \text{reason such that} \hspace{1cm} \text{rejection}(D R) \land \text{case}(R)$.

- $\forall D: \text{decision, best}(D) \iff \exists R: \text{reason such that} \hspace{1cm} \text{rationale}(D R) \land \text{case}(R)$.

- $\forall D: \text{decision, optimal}(D) \iff \text{committed}(D) \land \text{admissible}(D)$.

- $\forall D: \text{decision, optimal}(D) \iff \text{not rejected}(D) \land \text{admissible}(D) \land \text{best}(D) \land \exists G: \text{goal such that} \hspace{1cm} \text{valid-goal}(G) \land \text{objective}(D G)$.

---

<table>
<thead>
<tr>
<th>Optimal decision</th>
<th>Valid-goal assignment</th>
<th>Valid-goal decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admissible decision</td>
<td>Rejected decision</td>
<td>Retracted decision</td>
</tr>
</tbody>
</table>

**Figure 6:** Standard Decision Dependencies

We conclude this section with Figure 6, which illustrates the major dependencies defined in the theory above. In this network, nodes are supported by...
one or more justifications, denoted by arrows pointing to the node supported. The justification itself is supported by positive (solid lines) and negative (dashed lines) links to other nodes. There is reason to believe a node when it has a valid justification. A justification is valid when there is reason to believe each node with a positive link and no reason to believe any node with a negative link. For example, a subgoal is valid as long as its supergoal and parent decision are valid. A decision is valid as long as it is admissible, for whatever reason, and not retracted.

3.3 Behavior

Messages to Redux servers from client problem solvers use a protocol consisting of commands with semantics. We specify here only the non-obvious aspects of such a protocol, previously suggested in Section 2. The commands take typed objects defined in the theory in Section 3 as arguments.

Obviously there are commands for adding and deleting strings directly to and from the database S, and for making and rejecting decisions. The REDUX model imposes some restrictions on these commands. For instance, a decision may not be made to reduce a goal that is already reduced. A rejection reason must be valid, though it may be empty.

The example in Section 2.4 deserves some detail as it otherwise may strike one as magic. When the client problem solver detects a constraint violation among assignments, it may report this to Redux with a CONSTRAINT-VIOLATION-RESOLUTION command, which takes a (conjunctive) list L1 of the offending assignments, a list of any contributing propositions that may occur as strings in S, and a resolution list L2 of assignments that should be made invalid to resolve the constraint violation. Redux will then construct a rejection reason R consisting of a conjunction of the contributing propositions and valid-asg(A) for each element A ∈ L1, A /∈ L2. For every decision D that supports an assignment A ∈ L2, Redux will act as if it had received a rejection command for D with a reason of R.

Additionally, whenever a decision D2 is made for some goal for which another decision D1 was previously made but since rejected with reason R, Redux will conjoin rejected(D1) to any rationale of D2 that the client may supply, including nil. Thus the optimality rationale of the second decision depends upon the validity of the reason for the rejection of the first.

In the example in Section 2.4, upon being told that there is a constraint violation involving some feature of the exotic metal chosen in decision D11 and the hole-size of the connector, and that the latter assignment should be made the culprit in the conflict; REDX-I determines that D11 must be rejected and constructs a rejection reason consisting of the exotic metal feature. When decision D11 is made, the rejection of D11 is automatically a part of its rationale. If D11 ever becomes invalid, ENG-I will be informed by REDX-I that there is an opportunity to reconsider the original choice of motor, which was decision D11.

4 Summary

There are two general principles that may be used to reduce the amount of human work necessary to enable heterogeneous systems to cooperate in distributed design in systems such as PACT and SHADE.

A standard communications language minimizes semantic unification. KQML provides a small set of primitives that allow agents to communicate the terms that matter to them. This facilitates semantic unification because it points to just the connections that must be made, rather than having to unify complete models of heterogeneous agents. Information exchanged for the purpose of design can be used for design revision. The simplest case is that PACT should keep track of requests for information and notify requesters when that information changes.

The Redux server improves the use of these principles. A subset of REDUX primitives adds semantics to the syntactic KQML primitives. This further determines the points of agreement on domain semantics between the systems. REDUX also specifies specific ways that change should be propagated in a distributed design, reducing the necessity for manual change notice specifications.

Another important relation of this paper to the PACT/SHADE work is that the REDUX model is a theory for some class of design, and in that sense is a reusable ontology of the kind proposed in the SHADE project and to be represented in Ontolingua. It is different in that it is not an ontology of design domain objects, but an ontology that can be used to encapsulate such objects.

Finally we add that this is a proposal for a kind of architecture in the same spirit as PACT and SHADE and, indeed, to be built on top of them. The REDUX model has previously been implemented as a full...
problem solver useful for a certain class of problems as described in [7]. The idea presented here is to separate out the part of the model that performs decision maintenance and provide it as a network server to other problem solvers as a way to federate heterogeneous agents.

Implementing this proposal will undoubtedly require extensions and modifications to the basic model, as have already been suggested by the PACT experiment and work done at the University of Kaiserslautern. The hypothesis suggested here to be tested is whether it is possible to develop a structured encapsulation of heterogeneous design computations that provides sufficient added functionality to overcome the cost of structuring. The answer must be determined empirically and we intend to do so in cooperation with the SHADE and PACT projects.

Acknowledgements

Jay Weber was helpful in discussing the PACT example and REDUX applications. Tom Gruber originally suggested somehow positing REDUX as a server for representation by Ontolingua and has contributed to the formalization of the theory. Juergen Paulokat has also supplied valuable assistance. This work was partially funded by the Universität Kaiserslautern.

References


