An agent system for collaborative version control in engineering

Barry Florida-James
Newcastle University, Newcastle, UK
Nick Rossiter
Newcastle University, Newcastle, UK
Kuo-Ming Chao
Newcastle University, Newcastle, UK

Keywords
Agents, Engineering, Database management

Abstract
In this paper we present a system of distributed co-operating agents whose goal is to manage change and organise version sets in an engineering environment. The agents are designed for full lifecycle support and inter-operation across heterogeneous networks. The agent communication is based on common object request broker architecture (CORBA) but an extra messaging layer is developed which utilises a language built in Vienna development method-specification language (VDM-SL). Problems encountered in the use of engineering database management systems are investigated and solutions are proposed in the context of agents. A version model is presented in two ways; informally based on our assumptions on a general design process and formally in VDM-SL. An industrial case study is presented and preliminary results shown.

This research was supported by the UK Engineering and Physical Sciences Research Council (Grant No. GR/ J40270). The authors wish to thank John Fitzgerald for his advice on VDM modelling and Keith Hutchinson for his help in preparing the manuscript.

Integrated Manufacturing Systems
11/4 [2000] 258–266

[ISSN 0957-6061]

1. Introduction

The adoption of concurrent engineering principles in the production of large made-to-order (MTO) products has demanded significant progress in the supporting information systems towards the goal of an integrated engineering environment. The paperless design house is becoming increasingly realistic but total concurrent engineering activity, supported by computer aided design (CAD) and other computer based engineering systems, has not yet been realised. Whilst it is common for design engineers to have access to a central repository, it is unusual for this system to be optimised for performance and complete product lifecycle support. We share the view of Cutcosky *et al.* (1993), that a centralised product model is the logical framework for an integrated engineering process, however this quickly becomes a bottleneck when data physically resides on only one system. Added to this problem are the business issues of having to replace existing software and hardware and therefore re-train designers in the use of new systems and tools.

We believe that research from artificial intelligence, especially in the areas of knowledge sharing and re-use and agent architectures, should be applied to engineering systems. An environment should be produced where information is shared at formally defined levels instead of ad hoc data transactions. In this way designers engaged in traditional engineering disciplines should be aided in understanding product data from their collaborative partners, thus leading to a reduced product development cycle. In this paper we present a system for complete product configuration and version control which allows designers a consistent view throughout the complete lifecycle and also a retention of their own product representation and design tools.

The approach used is an agent-based (Luck and d'Inverno, 1995; Shen and Barthès, 1995) framework. This framework incorporates the standard for the exchange of product model data, STEP (Fowler, 1995) which is becoming mature enough to be used in some industries and also distributed object technology CORBA. CORBA is the underlying communication mechanism in our implementation and has the advantage that the architecture is abstracted to a level of communicating agents so that an agreed vocabulary is all that is required for integration. In the remainder of the paper, we present the version model and the proposed agent architecture, demonstrate the system with a case study taken from the petro-chemical industry and we develop a formal specification of the system using VDM-SL (ISO95, 1996).

2. Related work

Over the past ten years much research effort has been focused on the development of computer based systems aimed at the exchange and maintenance of data on large engineering products. Concurrently, research in artificial intelligence has been addressing issues of co-operation and knowledge sharing in various domains. In this section we examine these two strands of research and review recent progress.

A number of system prototypes consider heterogeneous distributed environments. KADBASE (Howard and Rehak, 1989) was one of the earlier systems to address the issue of semantic and syntactic translation. Later work at Queensland University (Yang and Papazoglou, 1995) classifies and organizes correspondences between heterogeneous object-oriented schema. This information resides in a knowledge base attached to each

local database. The knowledge base allows remote objects to be treated as local data types and also determines which part of a query is local and which is remote. At Stanford (Wiener *et al.*, 1996) a data warehouse system has been produced based upon CORBA objects and asynchronous messaging. Here a meta datastore is used, to resolve relationships across datastores. However, none of the above systems consider an evolution of project descriptions over time.

Katz (1990) describes a set of various criteria that a version model for engineering databases should meet. He describes interesting combinations of version properties as distinguishable version states but does not propose a suitable architecture for development of these models. Krishnamurthy and Law (1997) address many of these issues in a CAD environment but do not propose a model that would be generally applicable beyond this environment.

Morenc and Rangan (1992) state that, in concurrent engineering environments, activities typically involve a high degree of data and function interdependencies. Designers must be able to design independently, therefore resulting in assumptions having to be made about other design models whilst ensuring that overall consistency is maintained. We directly address this issue of consistency within the demands of these requirements.

Dattola (1996) presents a co-operative system of agents for hypertext version control but this system does not have the strict requirements associated with an engineering environment. The software agents that Genesereth and Ketchpel (1994) present were the starting point for our system but we chose to narrow the domain of the agents, in order to make the implementation more realisable.

Our view of what defines an agent is derived from Shoham (1993), and also from the Foundation for Intelligent Physical Agent (FIPA) standard (FIPA, 1998). We believe that an agent should have responsibilities and obligations that determine its behaviour and that the overall behaviour of the system is determined by a common goal.

In summary, we have not found a suitable management system for version control in a multidisciplinary design environment that is independent of the underlying design tools. Co-operating agents appear to offer a solution to this problem but have not yet been applied to this domain. In this paper we present such a version management system.

3. Version management

The scheme for version management is agent based. The goal of our agent architecture is global consistency of data and the ability to reflect change in all models of a disparate design process. An agent communication language (ACL) is implemented on top of CORBA but does not use the CORBA event services as this is too limited (see KQML and CORBA at http://www-ksl.stanford.edu/email-archives/kqml.messages/327.html). In its present form the ACL that we apply is specified in Vienna development method (VDM) and has a very limited vocabulary. However, we demonstrate that it is sophisticated enough to convey all the necessary semantics required for a configuration management system.

The version mechanism is session-based, that is updates are made at the end of a user session. The mechanism is intended to be used in full lifecycle support and also provides a history of the design process. In order to achieve this an agent architecture has been developed.

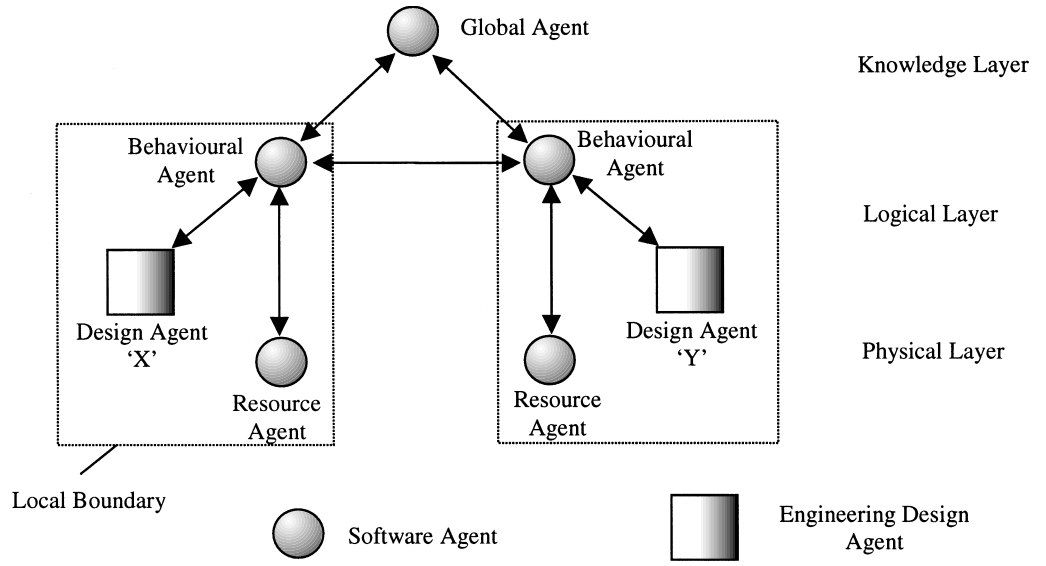
The architecture shown in Figure 1, consists of three layers which may be considered as the physical layer, the logical layer and the knowledge layer. Each layer is depicted by a separate type of agent namely resource, behavioural and global. As is shown each agent in the engineering process is represented by a behavioural agent, the resource agent manages the physical resources of the design agent and the global agent fulfils a specific role in the maintenance of the global or enterprise product model (Florida-James *et al.*, 1997).

To understand the role of each type of agent in more detail we will first present an informal description of our version model, based on our assumptions, on a general design process. The role of each agent in fulfilling that model is then described. In the following section formal specifications are presented in VDM-specification language (SL) describing the logic controlling the agent behaviour. An example of change management, from a case study, is also given.

3.1 Version model

The version model that we present here has a number of key concepts which separate it from other schemes (Bohrani *et al.*, 1992; Cellary and Jomier, 1990; Chou and Kim, 1986; Dittrich and Lorrie, 1988). First, we place no unrealistic requirements on the underlying design tools. An ability to export data is the only assumption. Second, in a more natural representation, we describe the tools as communicating agents with

Figure 1
Three tier agent framework



asynchronous messaging capabilities rather than as remote objects or functions which may be invoked or called. Finally we allow design agents to collaborate without relaxing any autonomy constraints on the individual design tools.

We will proceed with a description of our model by defining the main terms used in it.

An *entity* is an item which is considered as a design object in any participating model at some stage in the product lifecycle.

A *configuration* is a unique set of entities which when logically related describe the complete product model at a given point in time.

A *version* is a specific instance of a given entity, which may be derived from any previous version through a series of change operations.

Change operations are defined as *add*, *delete* and *modify*.

Version levels are described as *private*, *declared* and *recorded*.

3.2 Entity version management

Local models are managed by the local database and we assume no control or access to this structure. This allows designers to continue working with tools they are familiar with and also to introduce the system to legacy applications. We use the commonly described wrapper method (Roth and Schwarz, 1997) to access these systems but our wrapper is contained within the resource agents. The resource agent understands STEP and hence performs the translating of design entities from the local repository to the global repository.

The proposed scheme is a forward deltas scheme (Rochkind, 1975) where deltas are stored as a list of entities on which primitive operations have been performed. It is assumed that the three primitive operations we *describe*, *modify*, *create* and *delete*, can be used to represent all design actions across all domains.

The labelling scheme applied by the resource agent is external to any local scheme the design tool or database may have. It is, of course, possible to utilise any available local scheme provided that in the global context we get a unique version set identifier. The scheme is adapted from that of Keller *et al.* (1995) but at present we do not implement their optimisation processes. Entities are initially *create*d within a version set with a unique identifier corresponding to the EXPRESS model entity name. The entity is physically stored within the resource agent but rules concerning the creation and deletion of entities and their relationships at the global level are controlled by the global agent. An advantage of our scheme of resource agents is that it is reactive in the sense that changes made in another model are automatically reflected in all local models by the application of a new label. This label is actually designated by the resource agent and is required to be unique in the local context and to tell us which agent caused the change. Rules that govern the resource agent behaviour are given in Section 4.2 along with examples. These rules are further formalised in the VDM-SL specification.

As stated earlier we do not distinguish between complex and simple entities in our versioning system. Entities are related in

terms of a hierarchy by the global agent. STEP, however, defines a rigid hierarchical structure in its application protocols. The system does not choose to ignore this and indeed this structure is available within the local models where it is very useful. We use a less rigid definition of an entity for global version control. As stated this allows entities to exist at different levels of detail in different models or domains. The reason this is useful is that models based on mismatched domains can be related and merged using hierarchies containing generalisations and specialisations of existing terms (Florida-James *et al.* 1997). The difficulty is that a consistent representation of the knowledge of these relations has to be stored somewhere; this is one role of the global agent.

For example, consider a structural design representation consisting of a deck entity which contains four subsystems – high pressure, medium pressure, low pressure and power supplier. In the process design representation the low pressure subsystem is further decomposed into two pumps a compressor and an inspection gear launcher. In this example we can easily visualise the domain mismatch. The process representation is much more detailed than the load bearing concerns of the structural agents. It is also easy to see how the problem may be addressed by a hierarchical representation of the relationships between entities. Hence in the global agent there exists a list of entities and a set of relation types, namely, *is a part* of or *is equivalent to*.

3.3 Configuration management

The global and behavioural agents combine to give an overall configuration management system for the complete product lifecycle. This system has been developed to support earlier work on change propagation in an integrated design environment (Guenov, 1996). In this system the behavioural agents may be described as the logical layer and the global agent as the knowledge layer. The behavioural agents define the rules for co-operation and change management whereas the global agent has knowledge about design entities and their relationships in the global context of the total product.

The first aspect of the configuration management scheme is the labelling of design models. Versions are caused by change and hence the process of change is represented within our labelling scheme. When a requirement for a change is issued a conversation on this change is started between the behavioural agents and the currently declared versions of each model are updated with a new label as shown in Figure (2a). If this change is agreed the label on each version then becomes the timestamp and the other labels are removed. The change issue and delta storage are handled by the resource agent. In Figure (2b) a change is issued but this conflicts with constraints in another model so this agent produces a set of alternatives which compromise both constraints and the label is now composed by adding the agent's alternative. At this point the behavioural agents are in a state of conflict resolution and all subsequent alternatives are labelled in the same way. When the conflicts are resolved the label reverts to the timestamp as in the previous scenario.

In order to cope with the situation where the above resolution fails at this point in time, that is where parallel designs exist, we simply clone (Dattola, 1996) the global agent and give it another label based on the same scheme. For example in our case study, from the offshore industry, the development of two designs progressed simultaneously for a period whilst a decision on whether a concrete or metal jacket would be produced. Eventually however one design became inactive. Our experience shows that, whilst cloning the global agent may be expensive in system terms, it allows the model to continue consistently without inhibiting the design process. These clones very rarely stay active for long periods.

4. Agent descriptions

4.1 Messaging design

The messaging system supports communication in both broadcast and directed modes. The implementation chooses how to send messages depending on the circumstances in which the agents communicate. Messages may be sent to all, some or just one specific agent depending on its content. The system is supported by CORBA remote object calls and so the diversity of what can be sent is almost unlimited, in fact it would be possible for agents themselves to be transferred. A centralised post office, which is controlled by the global agent, determines the assignment of agents to subjects and conversations within the system.

4.2 Resource agent

The following state-based model shows the function of the resource agent, in its role as the version control. The resource agent also handles physical storage optimisation and conversion from local repositories to the STEP data standard.

Figure 3 showing the state changes in the resource agent is adapted from Krishnamurthy and Law (1997), but significantly we allow external messages to influence the internal states. This is illustrated by the change request arrow. This message causes an alternative to be declared. This alternative now has only one route to being declared, that is by being activated and incorporated into the private model. This process is analogous with an approval process and this is unique to our model.

4.3 Behavioural agent

The behavioural agent is the representative of each design discipline in the agent structure. It contains rules about co-operation and negotiation with other behavioural agents and uses these to operate version control over its own resources. The behaviour is controlled by a number of state variables which represent change activity and design activity.

Design activity is designated by two states:

1 *active* – it is safe for design to continue as normal; or
2 *frozen* – legitimate design activity is currently postponed due to some global inconsistency or constraint violation.

Change management is represented by a unique change identifier and two sets of variables: *activator*, a variable representing the agent which originated the change and *respondor* which represents the agents acting in response to a change. Agents must act as either an activator or responder in a given change conversation but may participate in any number of changes. Hence an agent may be acting as am activator and a respondor simultaneously. An agent's design activity can only be either *active* or *frozen*. An activator may be in the following states *pending* (evaluation/wish), *pending* (requirement), *resolving* or *recording*. A respondor may be *evaluating* or *conflicting*.

These states represent the following conditions:

- *pending* – awaiting one or more responses to a change request;
- *resolving* – awaiting the result of a management decision process;
- *recording* – updating the agent details after a change has been accepted;
- *evaluating* – assessing locally the effects of a change; and
- *conflicting* – having a change violate a local design constraint.

The rules for controlling these states are described in the change management examples and formally represented in VDM-SL.

Changes are described in three levels *required*, *evaluation* or *wish*. These levels have different effects on the state variables and different message passing priorities.

4.4 Global agent

The global agent has the responsibility for maintaining consistency across the information sources. It has the following

Figure 2
Labelling scheme

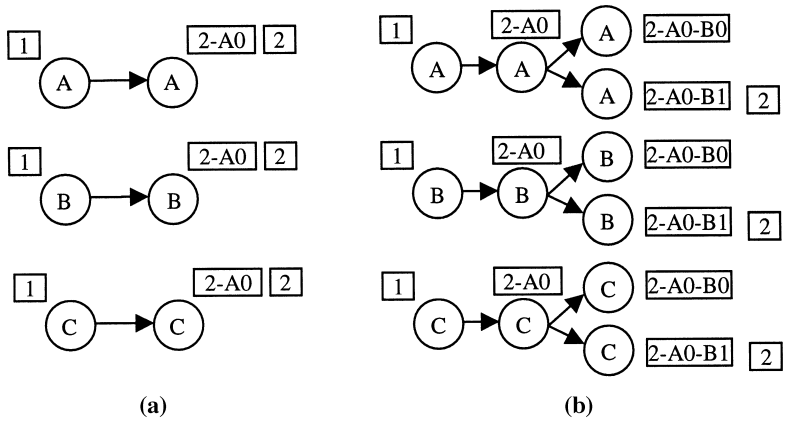
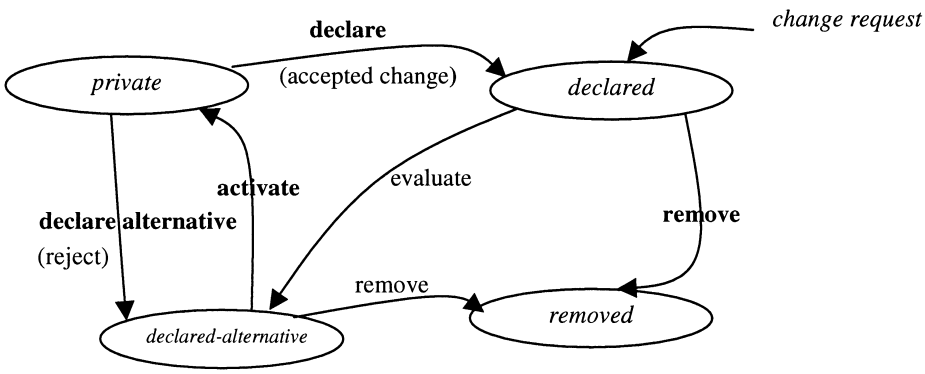


Figure 3
State changes at resource agent



commitments or obligations (Shoham, 1993):

- knowing the location of resources;
- knowing the current configuration of the product;
- controlling creation and deletion of system objects; and
- knowing relationships between objects at the global level.

Global objects and global relations can only be created and destroyed by the global agent. These global IDs are maintained across transactions and access to global objects and relations is only via the global agent. As well as knowing the current configuration the global agent maintains a history of the product evolution.

5. Case study – change management

This change scenario is taken from a case study of an offshore platform. Here we can see how the system of agents cope in a concurrent engineering environment. There are four agents in the system: process design, layout design, electrical systems and cost. The process engineer is required to change the export pressure of a pump. This increases the pump size and causes the generator size to be increased. As a consequence these can no longer fit in the available deck area so a constraint is violated within the layout design. Figure 4 shows the model changes and the version numbering.

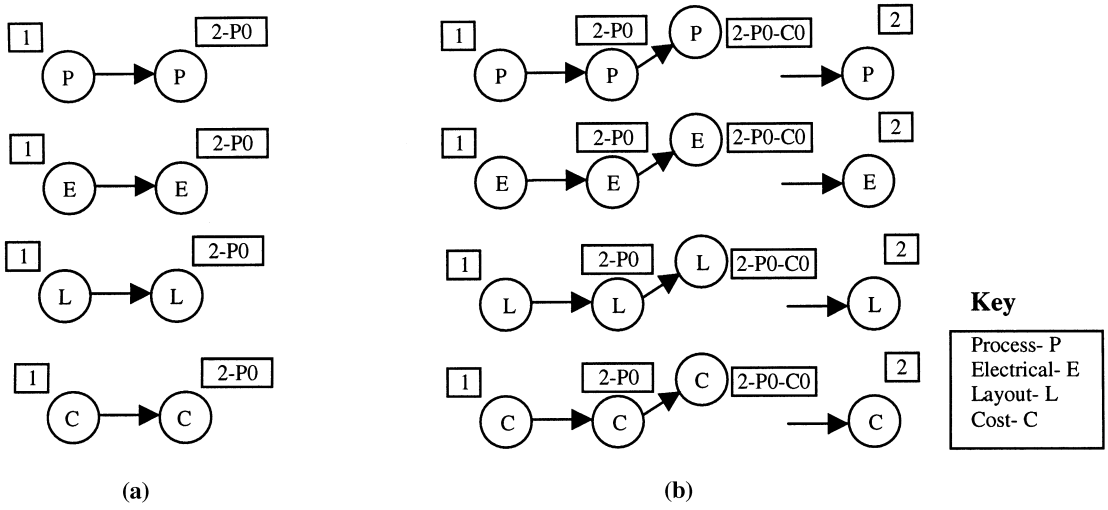
The layout agent has replied to the change request with a conflict. The activator invokes a stage of resolution and design is frozen by all agents. At this point the cost engineer produces an alternative generator which is more expensive but smaller, as shown in Figure (4.b). In this example the agents evaluate the new equipment fit and respond that this arrangement is within the specified constraints and the change is accepted. If this were not true then a management process would be invoked and a new global agent created. Figure 4 shows the various changes in the state variables at the agents involved.

Table I shows stage 1 with all agents active and activator and respondor states normal (N). After the requirement for the pump change is issued by the process designs behavioural agent, all design activity is frozen (F). The process activator state is set to pending-requirement (PR) whilst the other agents respondor states are evaluating (E). The layout respondor is then set to conflict (C) due to the constraint violation on the deck size. The activator replies to this by applying resolution (RE) and all respondors are now set to conflict. Concurrently, the cost engineer activator is set to pending-requirement (PR) as the alternative generator is selected and the process activator is now set to evaluating (E). As the final step, all design activity is now resumed and the respondor and activator states are set to normal (N) with the cost engineer recording (RC) the decision.

6. VDM-SL model

The VDM-SL model gives a formal representation of our versioning mechanism. VDM-SL is a model oriented language and we develop it by defining the data types and then developing the functionality. Using the IFAD VDM-SL tool-box (The VDM-SL Tool Group, 1994) we were able to make our model

Figure 4
Application to models of labelling scheme



executable and validate it. The model is defined in three parts representing the resource agent, the global agent and a model representing the concurrent message passing and the co-operating behavioural agents. The models are too large to be presented completely here but we will show the essential stages in model development with suitable extracts in VDM-SL. The third part is the largest and most complex and so we will examine it first.

6.1 Implementation of version control

The function described below ApplyRes is invoked when an agent receives a message from an activator telling it that conflict resolution has been applied.

```
ApplyRes: AgentID * ChangeID * AgentID *
VersionControl -> VersionControl
ApplyRes (agent, change, thisAgent, vc) ==
let conflict: RespondorStates = <Conflict>,
frozen:DesignActivity = <Frozen>,
   a = vc.AgentInfo (thisAgent)
in
mk_VersionControl(vc.timestamp, vc.labels,
vc.AgentInfo++
{thisAgent |-> mk_DesignAgent (a.version,
frozen, a.activators, a.respondors ++ {change
|-> conflict},
a.alternatives, a.replies, a.vState)}, vc.
ChangeDetails, vc.MessageBox)
pre vc.Agentinfo (thisAgent).respondors
(change) = <Evaluating>;
```

In this case the action is fairly straight forward, all responder states are set to conflict and design activity is frozen. No other changes to the version control model is made and no messages issued to other agents. The pre condition asserts the allowable design states before the application of the new states.

The following excerpt is from the function CreateVersion which is called as a response to an IssueChange message.

```
if vc.ChangeDetails(change) = <Conflict>
then
if type = 1 then
mk_VersionControl (vc.timestamp, vc. labels,
vc.AgentInfo ++ UpdateAgent (thisAgent,
vers, frozen, norm, eval, change, vc), vc.
ChangeDetails,
Send (type, recip,mk_ReplyConflict
(thisAgent, change), vc.MessageBox))
```

In this function the changes to the state variables are made by an auxiliary function UpdateAgent. Concurrently a ReplyConflict message is sent to the activator on this change. As can be seen the priority denoted by type is the same as the priority of the original message.

The following functions Record, Resolve and MoveOn show an agreed change being recorded and the agent states being updated.

```
Record: AgentID *VersionLabel *ChangeID
*VersionControl -> VersionControl
Record (agent, version, change, vc) ==
let da = vc.AgentInfo (agent) in
mk_VersionControl (vc.timestamp, vc.labels,
vc.AgentInfo ++ {agent |-> mk_DesignAgent
(da.version, da.activityState, da.activators ++
{change |-> <Recording>} ,da.respondors,
da.alternatives, da.replies, <recorded>)} ,
vc.ChangeDetails, vc.MessageBox)
pre vc.AgentInfo (agent) . vState =
<declared>;
```

The pre condition states that an agent must be in the state declared before it can be recorded.

```
Resolve: AgentID * ChangeID *
VersionControl -> VersionControl
Resolve (agent, change ,vc) ==
let version = mk_VersionLabel
(vc.timestamp, ChooseAlternative (vc.labels
(change)))
in
Record (agent, version, change, vc)
post MoveOn (vc);

MoveOn: VersionControl -> bool
MoveOn(vc) ==
forall agent in set rng vc.AgentInfo &
agent.vState = <recorded>;
```

The post condition on MoveOn is that all the agent states must now be recorded.

6.2 Resource agent

The resource agent model demonstrates the internal state changes and the external messages that influence it. The complexity of queues and priorities is not included in this state based model.

```
Messages = NewVersion | Retrieve |
EvaluateChange | Resolve;

state ResourceAgent of
version: VersionLabel
objects : set of ObjectID
changes: map ChangeID to Deltas
modelStates: map VersionLabel to
VersionState
expressmapping : map UniqueObject to
Express
```

Table I
State changes at design agents

Agent state	1	2	3	4	5	6
Proc-activity	A	F	F	F	F	A
Activator	N	PR	PR	RE	RE	N
Responder	N	N	N	N	E	N
Electrical-activity	A	F	F	F	F	A
Activator	N	N	N	N	N	N
Responder	N	E	N	C	E	N
Layout-activity	A	F	F	F	F	A
Activator	N	N	N	N	N	N
Responder	N	E	C	C	E	N
Cost-activity	A	F	F	F	F	A
Activator	N	N	N	PR	PR	RC
Responder	N	E	N	C	N	N

```
inv mk_ResourceAgent (version, objects,
changes, states, express ==
ObjectsUnique (objects)
and ChangesUnique (changes)
and Consitent (changes, states)
init ra == ra = mk_ResourceAgent(<a0>,
{}{\->}, {|->}, {|->})
end;
```

The resource agent consists of a version, a list of objects, a mapping between change identifiers and the changes, a mapping to represent the version states as mentioned earlier and a mapping representing the conversion from local object to EXPRESS model. The invariant functions ensure that objects and changes have unique identifiers and that the domain of modelStates is a subset of version.

```
StoreDeltas(chng : ChangeID, delta:Deltas)
ext wr objects
wr changes
post objects = objects~ union
ProcessDelta(deltas)
and changes = changes~ munion {chng |->
deltas};
```

The function StoreDeltas processes changes and stores them. The post condition states that the set of objectIDs after the change is equivalent to those before with the application of the changes. Also, the value of changes now includes the old value plus the new set of changes.

```
RetrieveObject(o:ObjectID, version:
VersionLabel) obj : Express
ext rd expressMapping
post obj =
expressMapping(mk_ObjectMarker(o,
version));
```

RetrieveObject returns an EXPRESS model of the given object selected from the version set by the identifier version.

```
Activate(version : VersionLabel)
ext wr modelStates
pre modelStates (VersionLabel) =
<alternative>
post modelStates(VersionLabel) = <private>;
```

Activate demonstrates an internal state change caused by an alternative being activated. The pre and post conditions control the state change.

6.3 Global agent

The global agent model is the most straight forward as we represent simply the agents ability to manage the consistency of global objects and relationships between them.

```
state GlobalAgent of
configuration : VersionLabel
objects : map ObjectID to Object
relationships : map RelationID to Relation
agents : set of Agent
inv mk_GlobalAgent (configuration, objects,
relationships, agents) ==
ObjectsUnique(objects) and
RelationShipsUnique(relationships) and
ObjectRelationsConsistent(objects,
relationships)
init ga == ga = mk_GlobalAgent (<a0>,
{|->} , {|->} , {} )
```

The global agent stores the global configuration, the list of global objects and the relationships between these objects. It also stores a directory of the participating agents in the variable agents. The invariants make sure that IDs are unique within the global agent and that the cross referencing between objects and relationships is consistent.

```
CreateObject(oid:ObjectID) == (objects :=
objects munion {oid |-> object} )
ext wr objects
pre ObjectID not in set dom objects;
DeleteObject(oid:ObjectID) == (objects :=
{oid} <-: objects)
ext wr objects
pre oid in set dom objects;
```

DeleteObject and CreateObject show the operation of the global agent, similar functions also exist for the relationships in the global agent. The pre conditions ensure that consistency is maintained by not allowing duplication within or removal of object identifiers not in the domain.

7. Conclusion

A novel system of version control has been presented which uses agents to fulfil its responsibilities. These agents address problems in three distinct layers: physical, logical and knowledge. Hence, we have three distinct types of agent: resource, behavioural and global.

In order to represent the agents in a declarative manner and also to verify the version control we chose to first specify our agent system in VDM-SL. The advantages of doing this were:

1 An executable model was available before system implementation on which test cases could be run.
2 The rules governing version control were explicitly stated and therefore could be;

:

- verified using formal techniques; and
- revised prior to a costly implementation.

It is our belief that the formal model has been successful for the reasons stated and also because version control is in essence a rule based exercise rather than an algorithmic one. Therefore it is much more intuitive to devise a set of rules and compare these with

design practice than to verify a complex algorithm.

Finally, the ability to describe product data in a meaningful manner throughout the complete lifecycle is critical in the successful engineering of large made-to-order products. This is the role of a version management system. In today's complex design environments, traditional storage mechanisms are inadequate. The use of agents, enabling one to incorporate predicates, knowledge and data models in one software component, addresses this issue. The architecture presented gives the ability not only to store data but also to manage global change and record design decisions meaningfully.

References

Borhani, M., Barthès, J.P.A., Anota, P. and Galliard, F. (1992), "A synthesis of the versioning problems in object-oriented engineering systems", *Proceedings of the Third International Conference on Data and Knowledge System for Manufacturing and Engineering*, Lyon.

Cellary, W. and Jomier, G. (1990), "Consistency of versions in object-oriented database", *Proceedings of the 16th International Conference on VLDB*, Brisbane.

Chou, H.-T. and Kim, W. (1986), "A unifying framework for version control in a CAD environment", *Proceedings of the 12th International Conference on VLDB*, Kyoto.

Cutkosky, M.R., Englemore, R.S., Fikes, R.E., Genesereth, M.R., Gruber, T.R., Mark, W.S., Tenebaum, J.M. and Weber, J.C. (1993), "PACT: an experiment in integrating concurrent engineering cystems", *Computer*, Vol. 20, pp. 28-37.

Dattola, A. (1996), "Collaborative version control in an agent-based hypertext environment", *Information Systems Journal*, Vol. 20 No. 4, pp. 337-59.

Dittrich, K.R. and Lorie, R.A. (1988), "Version support for engineering database systems", *IEEE Transactions on Software Engineering*, Vol. 14 No. 4, pp. 429-37.

FIPA (Foundation for Intelligent Physical Agents), (1998), *Draft Specifications*, 1998.

Florida-James, B., Hills, W. and Rossiter, N. (1997), "Semantic equivalence in engineering design databases", *Proceedings, 4th International Workshop on Knowledge Representation Meets Databases*, Athens.

Fowler, J. (1995), "STEP for data management, exchange and sharing", *Technology Appraisals*.

Genesereth, M.R. and Ketchpel, S.P. (1994), "Software agents", *Communications of the ACM*, Vol. 37, pp. 48-53.

Guenov, M. (1996), "Modeling design change propagation in an integrated design environment", *Computer Modeling and Simulation in Engineering*, Vol. 1, pp. 353-67.

Howard, H.C. and Rehak, D.R. (1989), "KADBASE: interfacing expert systems with databases", *IEEE Expert*, Vol. 4 No. 3, pp. 65-76.

ISO95 International Organisation for Standardisation (1996), "Information technology – programming languages, their environments and system software interfaces – Vienna development method specification language – part 1: base language", *International Standard ISO/IEC 13817-1*.

Katz, R. (1990), "Toward a unified framework for version modelling in engineering databases", *ACM Computing Surveys*, Vol. 22 No. 4, pp. 375-408.

Keller, Ullman, A.M. and Ullman, J.D. (1995), "A version numbering scheme with a useful lexicographical order", *Proceedings of the IEEE Data Engineering Conference*, Taipei, pp. 240-8.

Krishnamurthy and Law (1997), "A data management model for collaborative design in a CAD environment", *Engineering with Computers*, Vol. 13 No. 2, pp. 65-86.

Luck, M. and d'Inverno, M. (1995), "A formal framework for agency and autonomy", *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS 95)*, San Francisco, CA.

Morenc, R. and Rangan, R. (1992), "Information management to support concurrent engineering environments", *Proceedings of the 1992 ASME International Computers in Engineering Conference*, San Francisco, CA, pp. 135-48.

Rochkind, M. (1975), "The source code control system", *IEEE Transactions on Software Engineering*, Vol. SE- 1 No. 4, pp. 364-70.

Roth, M.T. and Schwarz, P. (1997), "Don't scrap it, wrap it! A wrapper architecture for legacy data sources", *Proceedings of the Twenty Third International Conference on Very Large Databases*, Athens.

Shen, W. and Barthès, J.P. (1995) "DIDE: a multi-agent environment for engineering design", *Proceedings of the First International Conference on Multi-agent Systems (ICMAS 95)*, San Francisco, CA.

Shoham, Y. (1993), "Agent-oriented programming", *Artificial Intelligence*, Vol. 60, pp. 51-92.

The VDM-SL Tool Group (1994), "The IFAD VDM-SL language", *Technical Report IFAD-VDM-1*, The Institute of Applied Computer Science.

Wiener, J.L., Gupta, H., Labio, W.J., Zhuge, Y., Garcia-Molina, H. and Widom, J. (1996), "A system prototype for warehouse view maintenance", *Materialized Views: Techniques and Applications*, Montreal, pp. 26-33.

Yang, J. and Papazoglou, M. (1995), "A configurable approach for object sharing among multidatabase systems", *4th International Conference on Information and Knowledge Management*, Baltimore, MA, pp. 129-36.