ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Gerard O'Regan and Sharon Flynn (Eds)

1st Irish Workshop on Formal Methods

Proceedings of the 1st Irish Workshop on Formal Methods, Dublin 3-4 July 1997

Paper:

A Generic Model for State-based Agent Systems

Arthur P. Hughes and Claus Pahl

Published in collaboration with the
British Computer Society

A Generic Model for State-based Agent Systems

Arthur P. Hughes

Department of Computer Science, University of Dublin, Trinity College
Dublin 2, Ireland

Claus Pahl

Department of Information Technology, Technical University of Denmark
2800 Lyngby, Denmark

Abstract

This paper presents a framework for modelling state-based agent systems. Agents are state-based entities in a possibly distributed environment. We will in particular address problems that arise when aspects of locality and shared objects have to be dealt with. The *Irish School of Constructive Mathematics* $CM^\clubsuit$ provides an algebraic framework based on group theory and recursion theory. Monoids are used to model the foundations of state-based systems: states and state change. It offers extensive methodological means to develop and verify specifications of distributed, state-based systems.

This paper is an investigation into structural and consistency aspects of dynamically evolving systems under consideration of locality and sharing. A model is presented which preserves the integrity of such systems by providing operators with appropriate algebraic properties.

1 Introduction

The notion of state is predominant in a large number of systems from programming languages, data repositories (such as file systems or data bases) to agent- or process based architectures. Having an internal state that can be modified is a common property of objects in those systems. Active entities that live in an arbitrary (modifiable) environment — we will call them **agents** in a **world** — may also have an effect on the state of other entities. This access might depend on statically predefined structures and also the dynamic shape of the current world. Thus, each agent might have its own perspective or view on the world in a particular state. Agents deal with instantiations of certain **notions** (i.e. **resources**) which are stored as part of their state. These notions might be in a simple case just identifiers denoting some kind of entity.

Defining such a system formally, we can profit from the advantages offered by formal methods: unambiguity, preciseness, abstractness. The Irish School of Constructive Mathematics ($CM^\clubsuit$) is a mathematical framework for formal specification (see its influence on the design of the *Irish School of VDM* ($VDM^\clubsuit$) [6, 10]) and is mainly based on the works of Mícheál Mac an Aircinnigh [1, 2, 3].

The technical contribution of our paper is a generic model for agent systems enhanced by results that allow us to deal with problems that occur if there are conflicting non-local notions for an agent (e.g. which non-local identifier has to be accessed). Our model allows the discussion of general properties of agent systems. The model will be applied for agent-based planning in distributed artificial intelligence and programming language semantics. Programming language semantic will appear to be a very suitable application to illustrate our ideas.

We will show that our generic model (which could as well be specified in *VDM* [13] or *Z* [23] in a similar way) benefits in particular from the results of Constructive Mathematics when reasoning about the model has to be done. Especially recent results from monoid theory will be applied. Integrity aspects such as controlled changes on non-local objects can be guaranteed.

Content. Section 2 introduces principles of Constructive Mathematics. The specification for simple state-based systems is dealt with in section 3. In the following section 4 the full model is presented and more sophisticated ap-

plications in our approach are investigated. The Appendix summarises some more advanced results of Constructive Mathematics related to the presented work.

2 Irish Constructive Mathematics

In $CM^\clubsuit$ we always start with building a model in the problem domain. $CM^\clubsuit$ concentrates on exhibiting structure and relationships between structures. Therefore, the algebraic structures group and monoid are the foundation of our approach. In fact, monoids are a very common structure in Computer Science. They characterise data types (sets) with an associative binary operation and an identity element under the operation (inverse elements as required for groups are not so common yet).

The **basic structures** (power set, sequence, map) are the results of applying functors $\mathcal{P}(-)$, $(-)^*$ and $- \rightarrow -$ (resp.) to domains. '—' is an argument placeholder.

1. The powerset functor $\mathcal{P} : A \mapsto \mathcal{P}A$ constructs $\mathcal{P}A$ for a domain A .
2. For domain A the star functor $*$: $A \mapsto A^*$ constructs the domain of sequences.
3. If A and B denote domains, then the domain of maps (partial functions) is constructed by $\rightarrow$: $A \times B \mapsto A \rightarrow B$.

$X \rightarrow Y$ denotes the space of all partial and total mappings from X to Y . By Y^X we denote the space of all total mappings $X \rightarrow Y$. The **removal** of a map $\mu \in X \rightarrow Y$ with respect to a set $S \in \mathcal{P}X$ is denoted by $S \triangleleft \mu$. This expression can be expressed in the *curried* form $\triangleleft[S]\mu$ or $\triangleleft_S \mu$. The significance of this notation will become clear if used in combination with other operators, e.g., $\mu' = (\mathcal{I} \rightarrow \triangleleft_S)\mu$, where $\mathcal{I}$ is the identity function and $\mu \in X \rightarrow \mathcal{P}Y$. $\triangleleft[S]\mu$ denotes the **restriction** of a mapping $\mu \in X \rightarrow Y$ by a set $S \in \mathcal{P}X$ (complementary to removal). $\mu_1 \sqcup \mu_2$ denotes the **extension** (augmentation) of the mapping $\mu_1 \in X \rightarrow Y$ by the mapping $\mu_2 \in X \rightarrow Y$ when the domains of the mappings are disjoint. dom and rng denote domain and range of a mapping. $\bigcup/\{S_1, \dots, S_n\}$ abbreviates $S_1 \cup S_2 \cup \dots \cup S_n$. χ is the characteristic function, i.e. tests membership.

A **monoid** $(M, \oplus, u)$ is defined by

- a set M ,
- a binary, associative operator $\oplus$, i.e. for $n, m, p \in M$:

$$(m \oplus n) \oplus p = m \oplus (n \oplus p)$$

- a (unique) identity element $u \in M$ with

$$m \oplus u = m = u \oplus m$$

Natural numbers form a monoid under addition with 0 as identity. Words over some alphabet form a monoid under concatenation (this is usually referred to as the *free monoid*). Priming of domains denotes the removal of the identity element.

A **monoid homomorphism** h from the monoid $(M, \oplus, u)$ to some other monoid $(P, \otimes, v)$ is a map $h : M \rightarrow P$ such that

$$h(m_1 \oplus m_2) = h(m_1) \otimes h(m_2)$$

$$h(u) = v$$

If the sets from the domain and range monoids of h coincide then h is called an **endomorphism**.

3 State-based Systems

In this section we will investigate principles of state-based systems to give a flavour of *specification and reasoning of state-based systems* in our mathematical framework. The core elements of $CM^\clubsuit$ have been presented in section 2. A summary of the mathematical machinery we use for our generic model can be found in the Appendix. We will refer to particular sections if necessary.

A notion of state is the basic mechanism on which the agents of our worlds rely. Therefore, the principles of specifying state and state change and reasoning about it will be presented here in an introductory section.

State-basedness is one of the paradigms in computing. In *denotational semantics* of programming languages [24, 9] a domain of memories is assumed. This is similar to the view of computers as machines with memory. We can define the meaning of a program (or the computation on a computer) by a partial function on memories.

3.1 Simple State-based Systems

A simple system can be defined by an abstract machine consisting of a state and operations on that state. The simplest notion of state is that of mapping some notions to associated elements, e.g. identifiers to values, expressed by the space $X \rightarrow Y$. This space will be later on used to represent local states of agents. The simplest state changing operation is overriding the value assigned to a particular identifier. We will use a special monoid, the *monoid of maps under override*, as the basic structure to model state and state change.

$$(X \rightarrow Y, \dagger, \theta)$$

$\dagger$ denotes the **override** operator which overwrites one mapping by another mapping. For mappings $\mu_1, \mu_2 \in X \rightarrow Y$ override is defined by:

$$\mu_1 \dagger \mu_2 = \triangleleft[\text{dom } \mu_2] \mu_1 \sqcup \mu_2$$

θ denotes the null map (the identity of the monoid). Override is naturally used to specify a substitution of bindings in a state, e.g. the semantics of the assignment statement $x := e$ in programming languages with a given environment μ :

$$\llbracket x := e \rrbracket_\mu \triangleq \mu \dagger [x \mapsto \llbracket e \rrbracket_\mu]$$

3.2 Locations

It is common for memory management for program implementation or data bases to introduce locations, e.g. to deal with allocation and de-allocation of notions. Stores are indexed by locations, i.e. are mappings from locations to some elements. Notions are mapped to stores.

The *outer laws for the monoid of maps* (see Appendix A) can be used for reasoning, if the assignment is defined on more complex environments, e.g. mapping variables to stores which are mappings from locations LOC to values $(X \rightarrow (LOC \rightarrow Y))$ or stack-based structures. Removal and restriction are outer laws for the monoid of maps, i.e. override *distributes* over $\triangleleft$ and $\triangleleft$.

An *indexed monoid* $(X \rightarrow M', \oplus, \theta)$ can be constructed from a base monoid $(M, *, u)$ and an indexing set X (see [3]). Elements of the new monoid are simply maps from X to M' in $(X \rightarrow M')$ where $M' = \triangleleft[u]M$, the identity element is the null map θ and the associative binary operator $\oplus$ is an indexed version of the base operator $*$. If $*$ is set union for example, then we usually construct $\oplus$ to be a map-building operator defined in terms of map extend and map override.

The monoid of *indexed mappings* $(X \rightarrow (Y \rightarrow Z)', \oplus, \theta)$, e.g. $VAR \rightarrow (LOC \rightarrow VAL)$ as an application for programming language semantics with locations, is the monoid of maps $(Y \rightarrow Z, \dagger, \theta)$ indexed with respect to X , i.e., we have another algebraic structure usable to prove properties. Later on we will use indexed mappings to introduce agents with local states.

Figure 1: A world of agents

4 Modelling Dynamically Evolving Systems with Locality

In this section, a **generic model** for **multi-agent architectures** is presented. Characteristics of the agent systems to be modelled are

- a dynamic shape of a collection of agents
- state-based agents
- locality aspects with respect to agents and their access possibilities
- side-effects or sharing with respect to an accessibility relation

These architectures are also sometimes called *open distributed agent systems*. Openness refers to the ability of agents to join or to leave the system.

We will indicate that the presented model covers a broad range of possible applications. We are able to deal with problems that arise if aspects of locality (distributed and local substates) and access to non-local objects occur. Security aspects are the main issue in our investigation of agent systems. In particular, we concentrate on integrity (other forms of security are e.g. confidentiality or availability) [7, 21].

Properties of our model will be established and will be based on the framework of Constructive Mathematics. Some of the results are obtained by transforming parts of the model into other spaces in which useful algebraic properties hold. Details about how monoids are constructed from others (obtaining isomorphic structures) and other technical results are presented in the appendix.

4.1 A General Model of Dynamically Varying Structures

4.1.1 Models and their Elements

Consider the following situation; we have a collection of agents, associated with each agent is a collection of notions or resources and each notion has some associated elements. The same notion may be used by different agents. An element

$$\kappa \in \mathcal{K} = W \rightarrow (X \rightarrow Y)$$

models this situation, where W denotes the **agents**, X denotes a set of possible **notions** (or **resources**) and Y denotes **associated elements** to the notions (which might be any kind of information).

Let us look at the elements (see figure 1) in more detail:

- γ is a relation describing a world¹ of agents with interrelationships (expressing the use of resources). σ describes how agents $w \in W$ see their personal world (σ is a substructure of γ). γ is of type $W \rightarrow \mathcal{PW}$, e.g. a tree or a graph.
- A modifying operation *Restructure* can be applied to worlds which changes the shape of the world, but does not alter particular (local) notions.
- There is an operation *Change* : $\mathcal{K} \rightarrow \mathcal{K}$. A change updates internal information. These changes are initiated by agents. Changes for w might have side-effects on other agents (which ones is expressed by γ).
- κ denotes a set of agents with their associated structure (the latter is assumed to be $X \rightarrow Y$), κ is of type $\mathcal{K} = W \rightarrow (X \rightarrow Y)$.

γ is the basic structure and describes the structure of the living world or the living system. w is a living entity with a predefined structure (which might not necessarily live in a certain state of the world). The execution of *Change* has to guarantee the **integrity** of the whole world with respect its structure γ . *Restructure* is much simpler, therefore we will concentrate on *Change* in the following.

We can think of a collection of agents W as a collection of independent entities (agents, processes, procedures) which have their own view or perspective on their environment (or universe of discourse). They share elements in the environment with others. They might not only think differently about their environment, they also might affect it (and due to their different opinions, there might be conflicts). We assume that local and global entities accessible for agents are structured in the same way. We have assumed a space $X \rightarrow Y$ associated with W by $W \rightarrow (X \rightarrow Y)$. X is the domain on which conflicts might arise. By Y , arbitrary information can be associated to the conflicting notion.

4.1.2 Identification

A particular notion might appear several times in one world (even for the view of a given agent w). Thus, the most appropriate has to be **identified**. The problem occurs since *naming* is *relative* (and not absolute).

$\mathcal{K}$ is the space of W -indexed maps $X \rightarrow Y$, i.e. agents with their associated structures. $\mathcal{K}$ represents the space of the overall view. Personal views or perspectives of each agent are represented in the space $\mathcal{K}_p$. κ^p is partitioned with respect to notions. Let $\mathcal{K}_p \subseteq \mathcal{K}$ denote the collection of agents with notions and their associated elements in which a notion is associated with one and only one agent. A particular notion appears only once in an element $\kappa^p \in \mathcal{K}_p$, this is called a **partition** in contrast to **distributions** in $\mathcal{K}$. The p in κ^p indicated that κ^p is partitioned. Personal views do not contain irrelevant information. Some of the information of the overall view are faded out. Therefore, $\mathcal{K}_p$ is a subspace of $\mathcal{K}$. An element $\kappa \in \mathcal{K}$ might look like:

$$\kappa = [\begin{array}{l} w_1 \mapsto [\begin{array}{l} x_1 \mapsto y_1, \\ x_2 \mapsto y_2 \end{array} \\ w_2 \mapsto [\begin{array}{l} x_1 \mapsto y_3 \end{array} \end{array}]$$

An element $\kappa^p \in \mathcal{K}_p$ represents the private view and might look like

$$\kappa^p = [\begin{array}{l} w_1 \mapsto [\begin{array}{l} x_2 \mapsto y_2 \end{array} \\ w_2 \mapsto [\begin{array}{l} x_1 \mapsto y_3 \end{array} \end{array}]$$

expressing that the information about $x_1 \in w_1$ is not relevant for w_2 .

4.1.3 Worlds

Each agent is possibly related to other agents. A relation

$$\gamma \in W \rightarrow \mathcal{PW}$$

---

¹The notion of *worlds* is used to indicate that we have *possible worlds semantics* in mind (see [14] or [24]). Possible worlds are a collection of models that are indistinguishable to an observer. The idea of possible worlds goes back to Leibniz.

captures this inter-relationship between agents. The characteristics of γ should reflect the way in which the agents behave, i.e. access their resources. σ can be derived from γ . Using (an expanded) σ , we will implement *Change*. Now, what is the meaning of changing the interpretation of a notion in the above situation? Given the agent $w \in \text{dom } \kappa$ we would like to define the change in the form

$$\llbracket \text{Change}(x \mapsto y) \rrbracket_{w\gamma\kappa} = \dots$$

This leads to the question: which notion x of which agent should be updated with y , if the notion is a non-local one to w ? The notion update should be the nearest notion to the agent w with respect to the relation γ . This shall be captured by the idea of a closure.

4.1.4 Closures

Let us assume that γ defines a tree, i.e. for each element in the range there is a unique predecessor. γ shall be called a **closure** or a predecessor relation.

$$\begin{aligned} \text{Clsr}_{w\gamma} : T(W) &\rightarrow T(W) \\ \text{Clsr}_{w\gamma} : \sigma &\mapsto \neg\chi_\gamma(w) \rightarrow \sigma \\ \chi_\gamma(w) &\rightarrow \text{Clsr}_{\gamma(w)\gamma}(\text{add}(w, \sigma)) \end{aligned}$$

add is overloaded (e.g. concatenation $\wedge$ for sequences, union $\cup$ for sets). Starting with an arbitrary world as a *sequence*, a sequence of agents results. The sequence begins with the agent w and is continued by those agents related by γ . To this end we use $\text{Clsr}_{w\gamma} : W^* \rightarrow W^*$ for sequences with $w \wedge \sigma$ instead of $\text{add}(w, \sigma)$. *Clsr* produces a list of agents based on w and the application of $\gamma(w)$ (recursively). This recursive function returns a sequence of agents up to the given agent w when evaluated for the null sequence Λ (a leaf in the tree γ):

$$\text{Clsr}_{w\gamma}(\Lambda) = w_1 w_2 \dots w.$$

The above algorithms do not terminate for graphs, if there are cycles. The general problem is therefore: given a directed graph γ and a node w in γ , produce an inverted graph with respect to w .

4.1.5 Agent Structures

The notions and their associated elements with a structure of agents must be found by expanding agents with their internal structure:

$$\begin{aligned} \text{Expd}_\kappa : T(W) &\rightarrow T(\mathcal{K}) \\ \text{Expd}_\kappa : \sigma &\mapsto \triangleleft[\kappa]^T \sigma \end{aligned}$$

κ represents the whole world. Restrict $\triangleleft$ normally expects a set, but is here adapted to other types T (i.e. $\triangleleft$ is overloaded). For a *sequence* of agents $\text{Expd}_\kappa : W^* \rightarrow \mathcal{K}^*$ is defined by $\text{Expd}_\kappa : \sigma \mapsto \triangleleft[\kappa]^* \sigma$. The expansion is here achieved by applying Expd_κ with respect to the restriction $\triangleleft[\kappa]$ (where $\triangleleft[\kappa]w = \triangleleft[\{w\}]\kappa$) across the sequence:

$$(\text{Expd}_\kappa \circ \text{Clsr}_{w\gamma})(\Lambda) = \langle [w_1 \mapsto \kappa(w_1)], [w_2 \mapsto \kappa(w_2)], \dots, [w \mapsto \kappa(w)] \rangle$$

The above expression denotes the expanded closure (personal view) for an agent, but still including irrelevant information.

4.1.6 Most Closely Related

A collection of agents shall be reduced in such a way that the notions nearest the agent w dominate those notions further way from the agent w . To achieve this, a binary operation $\ddagger$ on $\mathcal{K}_p$ is defined. This operation will remove all irrelevant information from the current κ . *Shadowing of information* means that the most closely defined notion is the relevant one, other definitions of the same notion are *shadowed* by the closest one. Since they are irrelevant, they shall be removed and we obtain a much simpler, *partitioned* structure with respect to notions as opposed to only *distributed* structures before.

Here, we benefit from the results of $CM^\clubsuit$. The *monoid of inverse maps*

$$(\text{Inv}(X \rightarrow Y), \dagger, \theta)$$

(see Appendix C) can be used to model distributed systems and their integrity. This monoid is constructed from the monoid of maps under override and is isomorphic to the latter (see Lemma C.3 in the Appendix). Let D be an arbitrary space which is to be partitioned. The partitioning can be defined by the inverse μ^{-1} of a map $\mu : D \rightarrow E$, i.e. $\mu^{-1}(y)$ for $y \in E$ gives a partition. Note that there is a close relationship between *partitioning* and *indexed monoids*.

For $\kappa_1^p, \kappa_2^p \in \mathcal{K}_p$ then define,

$$\kappa_1^p \dagger \kappa_2^p = (\mathcal{I} \rightarrow \delta(\kappa_2^p))' \kappa_1^p \oplus \kappa_2^p$$

(where the prime denotes removal of entries of the form $w \mapsto \theta$ – see Appendix C) with

$$\delta(\kappa_2^p) = (\cup / \circ \text{rng} \circ (\mathcal{I} \rightarrow \text{dom}))(\kappa_2^p)$$

The expression $\delta(\kappa_2^p)$ produces a set of all notions in κ^p . The operation $\dagger$ guarantees a partition with respect to the notions. $\dagger$ does not depend on the particular form of σ or γ . Important in the definition of $\dagger$ is, that it is based on the *indexed override* $\oplus$.

$$(\mathcal{K} = W \rightarrow (X \rightarrow Y)', \oplus, \theta)$$

is the *monoid of indexed maps under indexed override*, which is, as the monoid of inverse maps, constructed from the monoid of maps under override.

$$\kappa \oplus [w \mapsto \mu] = \begin{cases} \kappa \sqcup [w \mapsto \mu] & \text{if } \chi[w]\kappa \\ \kappa \dagger [w \mapsto \kappa(w) \dagger \mu] & \text{otherwise} \end{cases}$$

By reducing the above sequence of agents with respect to $\dagger$

$$(\dagger / \circ \text{Expd}_\kappa \circ \text{Clsr}_{w\gamma})(\Lambda) = [w_1 \mapsto \kappa(w_1)] \dagger [w_2 \mapsto \kappa(w_2)] \dagger \dots \dagger [w \mapsto \kappa(w)],$$

each notion will be associated with one and only one agent. The expression selects the personal view of agent w in the system κ with respect to γ .

4.1.7 Projecting

The above three operations are combined into one projection function,

$$\begin{aligned} \text{Proj}_{w\gamma} : \mathcal{K} &\rightarrow \mathcal{K}_p \\ \text{Proj}_{w\gamma} : \kappa &\mapsto (\dagger / \circ \text{Expd}_\kappa \circ \text{Clsr}_{w\gamma})(\Lambda) \end{aligned}$$

The function selects the notions in each world which are visible to the agent w . This structure corresponds to the personal view σ , but is already expanded by the associated structures in order to prepare the definition of *Change*.

The difficult part is now completed, what remains is the definition of three auxiliary functions implementing some operations on maps:

Splitting. Given a $\kappa^p \in \mathcal{K}_p = W \rightarrow (X \rightarrow Y)$ the following function breaks κ^p into two parts, the first part is the agents W and their associated notions X , the second part is the notions and their associated elements $(X \rightarrow Y)$:

$$\begin{aligned} \text{Split} : \mathcal{K}_p &\rightarrow \text{Split}(\mathcal{K}_p) \\ \text{Split} : \kappa^p &\mapsto ((\mathcal{I} \rightarrow \text{dom})\kappa^p, (\cup / \circ \text{rng})(\kappa^p)) \end{aligned}$$

$$\text{where } \text{Split}(\mathcal{K}_p) \subseteq (W \rightarrow \mathcal{P}'X) \times (X \rightarrow Y).$$

Merging. The inverse function of splitting is merging Mrg (note that $\mathcal{K}_p$ is a partition)

$$\begin{aligned} Mrg : Splt(\mathcal{K}_p) &\rightarrow \mathcal{K}_p \\ Mrg : (\beta, \mu) &\mapsto (\mathcal{I} \rightarrow \triangleleft[\mu])\beta \end{aligned}$$

which merges two elements.

Substitute. The last function to define overrides an argument ν by a constant map μ :

$$\begin{aligned} Sbst_\mu : (X \rightarrow Y) &\rightarrow (X \rightarrow Y) \\ Sbst_\mu : \nu &\mapsto \nu \dagger \mu \end{aligned}$$

All three functions (and $\dagger$ and $Proj$ as well) do not depend on the particular form of σ or γ and are therefore generic.

4.1.8 Change

We can now give the meaning of an update *Change* of a world κ under γ by a maplet $x \mapsto y$ initiated by an agent w , using the functions defined above:

$$\begin{aligned} \llbracket Change(x \mapsto y) \rrbracket_{w\gamma} \kappa = \\ \kappa \oplus (Mrg \circ (\mathcal{I} \times Sbst_{[x \mapsto y]}) \circ Splt \circ Proj_{w\gamma})(\kappa) \end{aligned}$$

It is clear for the above definition that we must first obtain the private views of the agent w ($Proj_{w\gamma}$). This produces an element of $\mathcal{K}_p$, which is split into two parts by $Splt$, one of the parts is updated $Sbst_{[x \mapsto y]}$, then we form an element of $\mathcal{K}_p$ by applying Mrg and use this to update (override) κ .

Our general principle is reflected in this specification: the whole system is taken, the personal view of the respective agent w is selected. This view is splitted into the map for agents and into a map for notions. Since we have a guaranteed partition we can easily update the interpretation for the notion x . Then the two parts are merged again and the whole system is updated (overridden) by the new personal view.

The core principle of dealing with the problems that might occur for systems with local and global aspects, is to reduce the whole system to the view of the modifying agent, perform changes on that view and then update the whole system.

4.2 Application for Multi-Agent Systems for Planning

Distributed artificial intelligence (DAI) is concerned with the development and analysis of cooperating (intelligent) processes, called agents [11].

Multi-agent architectures consist of a set of autonomous agents which cooperate to achieve a common goal [22, 25]. Each agent has to construct his own plan which in turn is part of a global plan. Agents are expected to help in building the global plan. Since cooperation is needed for planning, actions, plans and knowledge are to be distributed. *Cooperation* is the central aspect in DAI (distributed problem solving). It can be divided into the issues of *communication*, i.e. transfer of information, and *synchronisation*, i.e. temporal ordering of actions.

A large class of architectures are termed *blackboard*-architectures. A blackboard is a shared memory which is used to write on and read from. A blackboard can contain problem knowledge, actions to be performed and plans. Agents are not explicitly synchronised in this model, they are synchronised by access to the blackboard (see e.g. [5]).

This idea (see figure 2) fits easily into our framework.

- A tree γ represents a multi-agent system describing access to blackboard (a cone construction with the blackboard as central element — all elements are pointing to the blackboard).
- σ is the personal view of an agent (sees the blackboard)
- w is either an agent or the blackboard.

Figure 2: Distributed artificial intelligence

- *Notions* are plans and their *associated elements* are plan details or problem data with their associated information.
- *Restructure* might be adding or removing agents, implemented by simple operations on a tree.
- *Change* is update of plans.

These ideas can also be found in languages in which coordination models are implemented, see [4]. Each agent might have his own local data repository and has access to the global data repository, the blackboard. Agents are autonomous, i.e. not influenced by others in their decisions. They do not communicate directly with each other. The relation γ is a tree which has the blackboard as the root and the agents as leafs. Note, that we do not aim at modelling planning strategies or learning aspects in multi-agent systems, our model describes the basic structural framework on which these aspects can be modelled. Work in DAI mainly concentrates on for example the process of planning or learning among agents (even if some of these approaches use a similar notation to ours, see [25] or [5]). The importance of a clearly structured, consistency guaranteeing architecture is often neglected.

4.3 Application for Programming Language Semantics

The treatment of Algol-like languages requires a complex model, if nesting of procedures (e.g. with static scoping) and local variables are involved [24, 17, 18]. Procedures shall be regarded as agents. We will present a $CM^\clubsuit$ -model for a state-based language with:

- local variables
- a type system including procedural types
- block structure

Being based on $CM^\clubsuit$ and its operators, it will allow us to prove properties of the model.

The elements of our generic model can be instantiated as follows (see figure 3):

- A tree γ represents access links for activated procedures (according to the static nesting of procedure definitions).
- The sequence σ represents access links for a particular procedure activation.
- Agents w are activations of procedures.
- Notions are variables (identifiers) and their associated elements are values.
- *Restructure* is starting a new activation or finishing an activation (inserting or deleting on a tree).

Figure 3: Programming language semantics

- *Change* is state override.

We will make our main model from section 4.1 now more concrete

$$\mathbf{N} \rightarrow (I \rightarrow V)$$

with natural numbers representing agents (procedures are identified by numbers) and identifiers I and their values V .

We now introduce an operator related to the binary operator on the monoid of inverse maps $(\text{Inv}(X \rightarrow Y), \ddagger, \theta)$ (see Appendix C) which will be useful in addressing the semantics of local variables in a Pascal-like language with (possibly nested) procedures as parameters. For $\gamma_i, \gamma_j \in \mathbf{N} \rightarrow (I \rightarrow V)$ define

$$\gamma_i \ddagger \gamma_j = (\mathcal{I} \rightarrow \triangleleft[\delta(\gamma_j)])' \gamma_i \oplus \gamma_j$$

where $\delta(\gamma) = \cup / \circ \text{rng}(\mathcal{I} \rightarrow \text{dom})\gamma$ and where the priming denotes the removal of entries of the form $n \mapsto \theta$ (empty local environments). $\ddagger$ updates a global environment γ_i by γ_j . $n \in \mathbf{N}$ points to the local environment of an activation at nesting depth n . This operator and a subset of the set $\mathbf{N} \rightarrow (I \rightarrow V)$ form a *monoid*.

Let $\kappa \in \mathbf{N} \rightarrow (I \rightarrow V)$ be the activation stack of a program with the current procedure as the top element. Let $\gamma \in \mathbf{N} \rightarrow \mathcal{P}\mathbf{N}$ be the tree representing links from activations of procedures to activations of procedures in which the former were statically defined. What is the environment of the current procedure and what effect does the execution of the current procedure have on the activation stack?

From κ, γ and the current procedure a sequence of activations $\langle \omega_i \rangle$ can be obtained, where $\omega_i \in \mathbf{N} \rightarrow (I \rightarrow V)$ for each i . The section of the activation stack relevant to the current procedure can be found by applying a reduction to the sequence $\ddagger / \langle \omega_i \rangle$.

The environment $\mu \in I \rightarrow V$, in which the current procedure should be evaluated, is $\cup / \circ \text{rng} \circ \ddagger / \langle \omega_i \rangle$ (see $\pi_2 \circ \text{Splt}$). The position of the environment μ in the activation stack, denoted by $\beta \in \mathbf{N} \rightarrow \mathcal{P}'X$, is $\beta = (\mathcal{I} \rightarrow \text{dom}) \ddagger / \langle \omega_i \rangle$ (see $\pi_1 \circ \text{Splt}$). The execution of the current procedure transforms the environment μ into the environment μ' . The activation stack after the completion of the procedure will have the form $\kappa \oplus (\mathcal{I} \rightarrow \triangleleft_{\mu'})\beta$.

Even though we have been talking about language semantics, a language itself has not appeared explicitly so far. Let us adopt now an imperative language with properties as described in the introduction of this subsection. An assignment can be directly defined by *Change*. Assuming that procedures are program values, they are storable in the environment μ . They can be assigned to global variables and passed as parameters to other procedures. This allows procedures to be called outside the defining context. In a language with static scoping and side-effects, a global object accessed by a procedure might not exist. Our approach handles this problem in the following way. The operation *Clsr* computes the closure of a procedure with respect to nesting γ . If an accessed object is not defined in one of the procedures in the closure, the program can be aborted.

5 Conclusions

We have presented a generic model which should allow us to specify different applications sharing some common properties:

- a world of agents which might change its structure dynamically
- agents with a local state
- agents might effect other agents or share resources with other agents (these effects depend on some relation between agents and/or the dynamic configuration of the world)

As indicated, a number of different problems in specification and programming fit into our framework. Applications in distributed planning in artificial intelligence and aspects of locality and nesting in programming language semantics have been addressed.

The model was motivated by an attempt to model distributed agent systems, e.g. blackboard architectures for distributed planning. Looking at a variety of agent systems (such as programs, if procedures are regarded as agents which share data and which might be passed around), we observe that we have to deal with systems that not only change their state, but also alter their shape (or their structure). Agents might depend on each other based on static or dynamic relationships and either of these relationships can affect updates on the whole system. These dynamically evolving structures can also be found in process architectures formulated in the π -calculus [15, 16] where communication channels between processes might be passed around and used to create new connections dynamically. Since these aspects of locality and of static and dynamic dependencies recur in various areas, we have tried to capture and generalise them in our abstract model, which, we believe, provides a novel approach of viewing and reasoning about agent systems.

Our model is state-based. An introduction into specifying and reasoning about state-based systems has been given. Having state-based entities in a distributed environment, the problems of locality and visibility which might depend on static and dynamic interrelationships occur between those entities. The model consisting of domains and operation could be specified in a similar way in other modeltheoretic specification approaches as well. However using the monoid of maps under override, and constructing the monoid of indexed maps under indexed override or the monoid of inverse maps on the first, — which are specific results of our framework $CM^\clubsuit$ — we obtain a clearer structuring using their algebraic properties, which in turn allowed us to gain some new insights into structure and properties of agent systems.

The presented model should be applied to other distributed architectures such as CORBA [19] or the *World-Wide Web*. Possibly, generalisations have to be carried out, e.g. we might need to model static and dynamic interrelations between agents by two different relations.

Acknowledgements

The authors would like to thank Mícheál Mac an Airchinnigh, Hugh Gibbons and Greg Winters for their helpful comments and suggestions.

References

- [1] M. Mac an Airchinnigh. Conceptual Models and Computing. PhD thesis, Trinity College, Univ. of Dublin, 1990
- [2] M. Mac an Airchinnigh. Tutorial Lecture Notes on the Irish School of the VDM. In S. Prehn and W.J. Toetenel, editors, VDM'91 – Formal Software Development Methods, pages 141–237. Springer-Verlag, October 1991. Lecture Notes in Computer Science No. 552
- [3] Micheal Mac an Airchinnigh. Formal Methods and Testing. Technical report, Department of Computer Science, Trinity College, Dublin, Ireland, 1993
- [4] P. Ciancarini. Coordination Models and Languages as Software Integrators. *ACM Computing Surveys*, 28(2):300–302, 1996

- [5] E.-E. Doberkat, W. Hasselbring, and C. Pahl. Investigating Strategies for Cooperative Planning of Independent Agents through Prototype Evaluation. In P. Ciancarini, editor, Proc. First International Conference on Coordination Models and Languages, Cesena, Italy. Springer-Verlag, 1996
- [6] Alexis Donnelly, Dara Gallagher, and Arthur Hughes. The indexed monoid - a story of three proofs. Technical report, Trinity College, Dublin, 1995
- [7] J.E. Ettinger. Introduction: Key Issues in Information Security. In J.E. Ettinger, editor, Information Security - An Integrated Approach. Chapman & Hall, 1993
- [8] R. Goldblatt. Topoi: The Categorical Analysis of Logic, volume 98 of Studies in Logic and the Foundations of Mathematics. North-Holland, 1984
- [9] C.A. Gunter. Semantics of Programming Languages: Structures and Techniques. MIT Press, 1992
- [10] A.P. Hughes and A.A. Donnelly. An Algebraic Proof in VDM♣. In J.P. Bowen and M.G. Hinchey, editors, Proceedings ZUM'95: The Z Formal Specification Notation. Springer-Verlag, Lecture Notes in Computer Science No. 967, 1995
- [11] M.N. Huhns. Distributed Artificial Intelligence. Pitman/Morgan Kaufmann, 1987
- [12] N. Jacobson. Basic Algebra I. W.H. Freeman, 1985. (2nd edition)
- [13] C.B. Jones. Systematic Software Development with VDM. Prentice Hall, 1990
- [14] Dexter Kozen and Jerzy Tiuryn. Logics of programs. In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, Vol. B, pages 789–840. Elsevier Science Publishers, 1990
- [15] R. Milner, J. Parrow, and D. Walker. A calculus of Mobile Processes, part I. Information and Computation, 100(1):1–40, 1992
- [16] R. Milner, J. Parrow, and D. Walker. A calculus of Mobile Processes, part II. Information and Computation, 100(1):41–77, 1992
- [17] P.W. O'Hearn and R.D. Tennent. Semantics of local variables. In Applications of Categories in Computer Science, volume 177 of Lecture Notes Series. Cambridge University Press, 1992
- [18] F.J. Oles. Type Algebras, Functor Categories, and Block Structure. In M. Nivat and J.C. Reynolds, editors, Algebraic Methods in Semantics. Cambridge University Press, 1985
- [19] Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1995
- [20] G. Papy. Groups. Macmillan & Company Ltd., 1964. Translated from the French (Dunod, 1961) by M. Warner
- [21] D. Russell and G.T. Gangemi Sr. Computer Security Basics. O'Reilly & Associates, Inc., 1992
- [22] M.P. Singh. Multiagent Systems: a Theoretical Framework for Intentions, Know-how, and Communications. Springer-Verlag, Lecture Notes in Artificial Intelligence No. 799, 1994
- [23] J.M. Spivey. The Z Notation (2nd Edition). Prentice Hall, 1992
- [24] R.D. Tennent. Semantics of Programming Languages. Prentice Hall, 1991
- [25] F. von Martial. Coordinating Plans of Autonomous Agents. Springer-Verlag, Lecture Notes in Artificial Intelligence No. 610, 1992

Appendix — Advanced Elements of CM♣

The appendix contains results of Constructive Mathematics which are relevant for modelling agent systems.

A Outer Laws for the Monoid of Maps

Consider the space

$$\triangleleft[\mathcal{P}X] \subset (X \rightarrow Y)^{(X \rightarrow Y)}$$

which is a subset of the space of total mappings from $X \rightarrow Y$ to itself. Each element of $\triangleleft[\mathcal{P}X]$ is an endomorphism on the monoid of maps, i.e. for mappings $\mu_1, \mu_2 \in X \rightarrow Y$ and for a set $S \in \mathcal{P}X$ then,

$$\begin{aligned} \triangleleft[S](\mu_1 \dagger \mu_2) &= \triangleleft[S]\mu_1 \dagger \triangleleft[S]\mu_2 \\ \triangleleft[S](\theta) &= \theta \end{aligned}$$

Thus, removal with respect to a set is an **outer law** for the monoid of maps. A monoid with operators is formed. Georges Papy [20] used the terminology for groups. Also each element of the space $\triangleleft[\mathcal{P}X]$ is an endomorphism on the monoid of maps. Thus restriction with respect to a set is also an outer law for the monoid of maps.

The domain mapping, denoted dom , is a homomorphism from the monoid of maps under override $(X \rightarrow Y, \dagger, \theta)$ to the monoid of set under union $(\mathcal{P}X, \cup, \emptyset)$. $\triangleleft[S_1]S_2$ denotes the removal of the set $S_1 \in \mathcal{P}X$ from the set $S_2 \in \mathcal{P}X$. The removal with respect to a set is also an outer law for the monoid of sets. A monoid with operators is formed.

B Indexed Monoid

Theorem B.1 *Let $(M, *, u)$ denote an arbitrary monoid, with unit u , and $(M', *)$ the corresponding semi-group, i.e., with $M' = \triangleleft[u]M$. Then for a space X , the structure $(X \rightarrow M', \odot, \theta)$ is an indexed monoid which inherits its operator properties from $(M, *, u)$, where for μ in $X \rightarrow M'$,*

$$\mu \odot [x \mapsto m] = \begin{cases} \mu \sqcup [x \mapsto m], & \text{if } \neg \chi[x]\mu \\ \mu \dagger [x \mapsto \mu(x) * m], & \text{otherwise} \end{cases}$$

The monoid of bags $(X \rightarrow N', \oplus, \theta)$ is the monoid of natural numbers $(N, +, 0)$ indexed with respect to X . The monoid of relations $(X \rightarrow \mathcal{P}'Y, \odot, \theta)$ is the monoid of sets $(\mathcal{P}Y, \cup, \emptyset)$ indexed with respect to X , which can be used to model graphs. The monoid of *distributed sorted sequences* $(X \rightarrow \Sigma_{\leq}^+, \odot, \theta)$, e.g. hash tables, is the monoid of sorted sequences $(\Sigma_{\leq}^*, \nabla, \Lambda)$ indexed with respect to X .

Properties of indexed monoids were difficult to prove. A new monoid helping to solve our problem shall be introduced.

Lemma B.1 *Let $(M, *, u)$ denote a monoid. Then for a space X , let M^X denote the set of all total mappings from X to M , and let u^X denote the constant mapping from X to the unit u . For $f, g \in M^X$ define,*

$$(f * g)(x) = f(x) * g(x)$$

*for all $x \in X$, then $(M^X, *, u^X)$ is a monoid, called the **X-direct power** of M .*

This construction is taken from N. Jacobson [12]. The indexed monoid operation may be expressed in terms of the X-direct power operation. For mappings $\mu, \nu \in X \rightarrow M'$ we define

$$\mu \odot \nu = ((u^X \dagger \mu) * (u^X \dagger \nu))'$$

where the priming denotes the removal of entries of the form $x \mapsto u$ and u^X denotes the constant mapping from X to u .

Lemma B.2 *For an arbitrary monoid $(M, *, u)$ and a space X , the indexed monoid $(X \rightarrow M', \odot, \theta)$ is isomorphic to the X-direct power monoid $(M^X, *, u^X)$.*

The isomorphism is override on the left by the constant mapping from X to u ,

$$u^X \dagger \_ : (X \rightarrow M', \odot, \theta) \longrightarrow (M^X, *, u^X).$$

The inverse map, which removes entries over the unit u , is an isomorphism as well.

$$(\_)': (M^X, *, u^X) \longrightarrow (X \rightarrow M', \odot, \theta).$$

C The Monoid of Inverse Maps

The tail-recursive function below will invert a mapping forming a bundle (see Topoi by Robert Goldblatt [8]). See [3] for a similar treatment.

$$\begin{aligned} \text{Inv} &: (X \rightarrow Y) \longrightarrow (Y \rightarrow \mathcal{P}'X) \\ \text{Inv}(\mu) &= \text{Inv}[\mu](\theta) \end{aligned}$$

$$\begin{aligned} \text{Inv} &: (X \rightarrow Y) \longrightarrow (Y \rightarrow \mathcal{P}'X) \longrightarrow (Y \rightarrow \mathcal{P}'X) \\ \text{Inv}[\theta]\beta &= \beta \\ \text{Inv}[\mu \sqcup [x \mapsto y]]\beta &= \text{Inv}[\mu](\beta \odot [y \mapsto \{x\}]) \end{aligned}$$

Lemma C.1 *If $\mu \in X \rightarrow Y$ then the collection $\text{rng}(\text{Inv}(\mu))$ is a partition of $\text{dom } \mu$.*

The recursive function is not onto,

$$\text{Inv}(X \rightarrow Y) \subset Y \rightarrow \mathcal{P}'X,$$

as the space $Y \rightarrow \mathcal{P}'X$ contains elements with ranges which are not partitions. The space of maps $X \rightarrow Y$ forms a monoid with the override operator $(X \rightarrow Y, \dagger, \theta)$. An operator is sought such that the space $\text{Inv}(X \rightarrow Y)$ forms a monoid. If $\beta_1, \beta_2 \in \text{Inv}(X \rightarrow Y)$ then define,

$$\beta_1 \ddagger \beta_2 = (\mathcal{I} \rightarrow \lhd[\cup/\text{rng } \beta_2])' \beta_1 \odot \beta_2$$

where the priming denotes the removal entries of the form $y \mapsto \emptyset$.

Lemma C.2 *The space $\text{Inv}(X \rightarrow Y)$ becomes a monoid with the above binary operator, denoted $(\text{Inv}(X \rightarrow Y), \ddagger, \theta)$.*

Lemma C.3 *The monoid of maps $(X \rightarrow Y, \dagger, \theta)$ is isomorphic to the monoid of inverse maps $(\text{Inv}(X \rightarrow Y), \ddagger, \theta)$.*