

Analysis of Inheritance Mechanisms in Agent-Oriented Programming

Lobel Crnogorac

Dept. of Computer Science
The University of Melbourne
Vic. 3052, Australia
E-mail: lobel@cs.mu.oz.au

Anand S. Rao

Australian AI Institute
Level 6, 171 La Trobe Street
Vic. 3000, Australia
E-mail: anand@aaii.oz.au

Kotagiri Ramamohanarao

Dept. of Computer Science
The University of Melbourne
Vic. 3052, Australia
E-mail: rao@cs.mu.oz.au

Abstract

Research into the design methodologies for agent-oriented programming (AOP) is still in its infancy, with most of the work focused towards applying the well-known object-oriented concepts. Integration of inheritance into AOP seems a natural step in that direction. This paper addresses the notion of inheritance in AOP. We identify and formally define two problems which significantly reduce the usefulness of inheritance. We present a formal framework to investigate the notion of inheritance and the above problems. We define two alternative inheritance mechanisms for AOP and formally show the trade-offs involved. We hope that the results obtained from the analysis will help AOP language designers in their search for better inheritance mechanisms and a successful integration of inheritance within AOP.

1 Introduction

Multi-agent systems are concurrent systems based on the notion of autonomous, reactive, internally-motivated agents interacting in a constantly changing environment. Research in this field has been primarily concerned with conceptual and architectural issues of these systems and how they can be applied to real-world problems. More recently, there has been an increasing trend towards capturing the essential features of these systems and embodying them into language constructs, popularly called Agent-Oriented Programming (AOP) [Shoham, 1993; Wooldridge and Jennings, 1994], as well as developing a methodology for analysing and designing such systems, the emergence of Agent-Oriented Methodology (AOM) [Brazier *et al.*, 1995; Kinny *et al.*, 1996].

Naturally, work in AOP and AOM has been influenced by existing paradigms - especially the Object-Oriented Programming (OOP) paradigm. Although, AOP is often viewed as a special case of OOP [Shoham, 1993], very little attention has been given to one of the major aspects

of OOP, namely, *inheritance*. The basic concept of inheritance allows the user to define a new class by extending an existing class with additional variables and methods. By sharing code among classes in this way, the total amount of code in a system can be reduced drastically.

Agents, unlike classical objects, need to synchronise their autonomous and reactive behaviour (multiple threads), they need to interact with the environment and other agents (the agent space is inherently concurrent) as well as commonly being required to satisfy real-time constraints. As a result, agents are usually significantly more complex than typical objects, both in their internal structure and in the behaviours they exhibit. Code re-use becomes highly important in such a context since it is much easier to design systems by extending already proven programs. Thus, the very complexity of agents makes the notion of inheritance a very useful and powerful tool for AOP. This paper addresses the notion of inheritance - its meaning, its potential benefits, and the problems that arise in introducing it within AOP.

An *inheritance mechanism* specifies exactly how existing agent descriptions may be inherited to produce new descriptions. One of the important decisions to be made by the language designer with respect to inheritance is deciding on its granularity, *i.e.*, what is the smallest unit of re-use within agents. This choice influences the usefulness of the concept, its expressiveness and its flexibility in aiding the development of re-usable agents.

The potential benefits of introducing inheritance in AOP are two-fold. First, it can reduce the complexity of the programming effort by enabling code sharing. In particular, an agent system may be designed by modelling the abstract behaviours of its constituent agents (*e.g.*, their roles). Agents exhibiting related abstract behaviours are expected to share a large part of their implementations. Second, it can facilitate the modular verification of multi-agent systems. Assume that we have proven certain behavioural properties of an agent family. It would be very useful, if rather than starting from first principles, one could prove properties of a derived agent

family from provable properties of the original family.

However, the realisation of these potential benefits depends crucially on how one defines the inheritance mechanism. In particular, two types of problems could occur: the *lack of behaviour preservation* and the *presence of inheritance anomalies*. The lack of behaviour preservation leads to the need to prove properties about each agent implementation in an inheritance hierarchy from first principles (*i.e.*, the second benefit above is not realised). Inheritance anomalies are related to the expressive power of an inheritance mechanism. An inheritance anomaly occurs when the inheritance mechanism is not powerful enough to provide code sharing between agents implementing related abstract behaviours (*i.e.*, the first benefit above is not realised). These two problems can greatly reduce the usefulness of inheritance. Solving them requires the design of more complex inheritance mechanisms. We show that this usually leads to a trade-off, *i.e.*, designing more complex mechanisms may reduce the occurrence of inheritance anomalies while increasing the problems with behaviour preservation.

The key contributions of the paper are at two levels. At the technical level it presents a formal framework of inheritance, behaviour preservation, and inheritance anomaly. Using this framework, we prove that certain classes of inheritance mechanisms are behaviour preserving and anomaly-free. Examples of inheritance mechanisms are provided using the agent model of an implemented agent system – dMARS (distributed Multi-Agent Reasoning System). At the conceptual level, we believe the paper brings the technical rigour of analysis of classical programming languages to the field of agent-oriented programming. We hope that our analysis can help other researchers to make similar investigations of formal aspects of agent-oriented concepts before introducing them as language constructs in AOP languages.

2 Informal analysis

This section presents the dMARS programming environment and gives examples of typical problems that can arise when inheritance is introduced.

2.1 The agent setting

dMARS is an agent-oriented distributed real-time system based on the popular PRS model of agents [Georgeff and Lansky, 1986]. It provides a representational framework and reasoning mechanism for implementing agents.

An *agent family* X is a tuple $\langle B_X, G_X, P_X \rangle$. B_X is a set of belief atoms where each belief atom $b(\mathbf{v})$ (b is a predicate symbol, $\mathbf{v} = (v_1, \dots, v_n)$, v_i is a term) provides information about the state of the environment as perceived by the agent. G_X is a set of goals, where each goal describes the desired state of the system. If g is a predicate symbol and $\mathbf{v}$ is as defined before, the

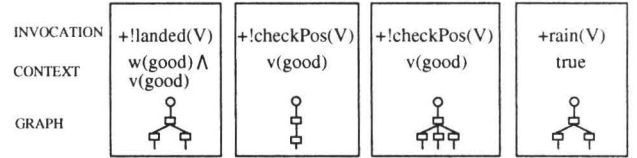


Figure 1: Plans of *SimplePlane*

goal $!g(\mathbf{v})$ states that the agent wants to achieve a state where $g(\mathbf{v})$ is a true belief. P_X is a set of plans, where each plan specifies how to accomplish a goal or react to certain situations. An agent family is a set of *agent instances* (or simply *agents*) that have the same set of beliefs, achieve the same set of goals, and use the same set of plans to achieve all their goals. Agent instances are dynamic, run-time entities, while agent families are static descriptions corresponding to *classes* in OOP. At any point in time an agent contains a set of ground belief atoms. Agent reacts to events occurring in its environment by invoking its plans. Plans can also be invoked by events generated internally by the agent. Each plan consists of an invocation condition, context condition and a plan graph, *e.g.*, the plan p is denoted by $\langle p_{inv}, p_{con}, p_{pg} \rangle$. The invocation condition identifies the event that must occur for this plan to be invoked. The events considered are the addition (+)/deletion (−) of beliefs/goals. Hence, $+b(\mathbf{v})$, $-b(\mathbf{v})$, $!g(\mathbf{v})$, $!g(\mathbf{v})$ are events. The context condition is a logical formula composed from belief atoms and the connectives $\wedge$ and $\vee$. The context condition must be satisfied by the set of ground belief atoms of the agent for the plan to be applicable. Plan graphs are acyclic graphs with their edges labelled with the sub-goals or primitive actions that have to be satisfied or performed in order for the main goal to be satisfied. We make a distinction between *public* goals and *private(internal)* goals. In particular, $G_X \subseteq \{!g(\mathbf{v}) : \exists p \in P_X \text{ such that } p_{inv} = +!g(\mathbf{v})\}$, is the set of all public goals that can invoke plans of an agent family X . A specific instance of an agent family acquires an initial set of ground beliefs, goals to be achieved, and the plans of the agent family. When the agent interacts with the environment, it adopts and commits to plans forming intentions. Details are provided elsewhere [Rao, 1996].

Example 1 Figure 1 shows several plans of an agent family *SimplePlane* which describes agents simulating simple types of planes. The event $+!landed(\text{base})$ triggers the first plan which is designed to land the plane in good weather ($w(\text{good})$) and good visibility ($v(\text{good})$). The execution of this plan involves executing the sub-goal $!checkPos(V)$ which then triggers the respective plans. The choice between the two plans for $!checkPos(V)$ is based on a notion of priorities. The higher priority plan is tried first, and if it fails then the second plan is tried.

The plan for $+rain(V)$ reacts to a change in the environment. It is triggered when the new belief $rain(V)$ is added to the agent's set of beliefs. ■

We now illustrate the problems that may arise when a simple inheritance mechanism is added to an agent-oriented language such as dMARS. Inheritance is introduced by following well-known OOP concepts. Hence, an agent (object) inherits all the plans (methods) of its parent, re-defines some plans and possibly adds new plans.

2.2 Behaviour preservation - an example

Agents, interacting with the environment generate behaviours. Characterising the interesting aspects of a behaviour is highly application specific and non-trivial. In some applications, behaviour can be characterised by considering *what* goals were achieved, while in some others one may want to capture *how* the goals were achieved, or *when* they were achieved (*e.g.*, real-time systems). In this section, we look at behaviour preservation with respect to the first characterisation, *i.e.*, what goals are achieved. Suppose we have implemented an agent family X , and proven that it achieves some desired behaviour. A new agent family Y is defined by inheriting X and adding some new plans. Does an arbitrary instance of Y at least still achieve all the goals that an arbitrary instance of X could achieve (if placed in the same environment)? In other words, can we guarantee some properties of Y without needing to prove them explicitly?

Example 2 Suppose instances of *SimplePlane* always land successfully in good weather and visibility. Consider a new agent family *AdvancedPlane* which inherits all the plans of *SimplePlane* and adds a new plan for the goal $!landed(V)$ enabling its instances to land in all weather and visibility conditions. We have inherited and extended *SimplePlane* with a new plan. Now we need to verify that an instance of *AdvancedPlane* can actually land in all conditions. However, we cannot even guarantee that it can land in good weather and visibility. If the new plan has higher priority than the inherited plan for $!landed(V)$ then the new plan is triggered first. The new plan could contain an error, causing *SimplePlane* to crash. Hence, a verification of the desirable behaviour for *AdvancedPlane* must be done from first principles. ■

Note that there are always particular examples of inheritance which do in fact preserve behaviour. In Example 2, if the inherited plan has higher priority than the new plan then the behaviour of *SimplePlane* is preserved by *AdvancedPlane* (it is immediately known that both can land in good weather and visibility). More complex examples can be constructed that do not involve plan priority and yet fail to guarantee behaviour preservation. Such examples are based on the operational semantics of plan execution. Ideally, the inheritance mechanism should guarantee behaviour preservation.

2.3 Inheritance anomaly - an example

Suppose we have defined an agent family X which implements our desired behaviour. Now, we want to build a specialised agent family Y that can achieve all the goals that X can achieve, and also some additional goals. We should be able to implement Y by inheriting all the plans from X , while any behaviours not exhibited by X are implemented by writing new plans. If it is not possible for X and Y to share the code in this way, then we have an occurrence of inheritance anomaly.

Example 3 Suppose we define a new agent family *NewPlane* by inheriting *SimplePlane* and adding a new plan for $!checkPos(V)$. The new plan has the context condition $v(poor)$, enabling the agent to check its position in poor visibility. Thus, an instance of *NewPlane* should be able to land in good weather, under any visibility. The problem is that the context condition of the inherited plan for $!landed(V)$ still restricts the agent to land only in good visibility. This plan must be re-written with a new context condition, $w(good)$. Hence, even though there is some code sharing between *NewPlane* and *SimplePlane* (the plan for $+rain(V)$ and the existing two plans for $!checkPos(V)$), the amount of code sharing has been unnecessarily reduced. It is possible to construct examples where none of the plans can be shared. ■

Re-defining plans of the parent family is error-prone, time consuming, and it violates the notions of modularity and encapsulation. Inheritance was designed to solve exactly this problem by code sharing. Thus, inheritance anomalies greatly decrease the usefulness of inheritance.

3 Formal analysis

In this section, we firstly define an inheritance mechanism to be the set of transitions from one syntactic specification to another. We provide two examples of such mechanisms with varying expressive power. Next, abstract behaviours of agent families are formalised and two alternative ways of defining behavioural equivalence (*i.e.*, when do two agent families exhibit the same behaviour) are considered. The language semantics determines how syntactic specifications are mapped into behaviours. The notions of behaviour preservation and inheritance anomaly are then formalised by analysing the behaviours of derived and original syntactic specifications. Behaviour preservation is similar to the notion of *soundness*: only the desirable behaviours should be derivable by the given inheritance mechanism. Inheritance anomaly is similar to the notion of *completeness*: all desirable behaviours should be derivable by the inheritance mechanism. Finally, we prove which of our inheritance mechanisms are behaviour preserving and anomaly-free.

3.1 Inheritance mechanism

First, we provide a general definition of an inheritance mechanism and then specialise it to AOP. An inheritance mechanism can be viewed as a transition relation between syntactic specifications. More formally we have:

Definition 1 An *inheritance mechanism* is a tuple $(Spec, \Delta, \rightarrow, \dashrightarrow)$ where $\rightarrow \subseteq Spec \times \Delta \times Spec$. An element of $\rightarrow$, (a, δ, b) is called a *transition* where $a, b \in Spec$ and $\delta \in \Delta$. $Spec$ is the set of all syntactic specifications. The set Δ is the set of syntactic entities which specify the differences between a and b . We write $a \xrightarrow{\delta} b$ for $(a, \delta, b) \in \rightarrow$. Furthermore, $a \xrightarrow{\delta \in \Delta^*} b$ is used to denote the transitive closure of $\rightarrow$, i.e., a sequence of transitions. Remark that the overloading of the notion $a \xrightarrow{\delta} b$ is quite harmless. The relation $\dashrightarrow \subseteq \rightarrow$ is the set of *incremental* transitions which maximise code sharing. ■

In the context of AOP, a syntactic specification is simply the definition of an agent family. Intuitively, the transitions specify how inheritance can be used to move from a syntactic specification a (e.g., *SimplePlane* in Example 2) to a new specification b (a full definition of *AdvancedPlane*) by specifying the differences (the new plan for *!landed(V)*) in δ . The general form of Definition 1 is specialised to AOP by instantiating $Spec$ to the set of all possible agent family definitions, i.e., the set including $X, Y, \dots$ etc. Introducing inheritance to an agent language also means defining a particular instance of relations $\rightarrow$ and $\dashrightarrow$, i.e., giving a set of transitions. Given our definition of an agent family the following possibilities arise: We can derive an agent family specification Y from an agent family X , by (a) adding or deleting belief atoms; (b) adding or deleting goals; and (c) adding, deleting, or modifying plans. While additions and deletions of belief atoms, goals, and plans in an agent family specification are straightforward, there are many ways in which a plan can be modified. A plan can be modified by modifying the invocation, context, or plan graph. We do not consider Case (a) in this paper. Now we consider a useful type of inheritance obtained by following the basic idea of inheritance as found in OOP (as in Section 2):

Definition 2 $X \xrightarrow{\delta}_{mod} Y$ for some δ iff $|P_X| \leq |P_Y|$ and $B_X = B_Y$. It follows that $G_X \subseteq G_Y$. ■

Agent family Y is obtained from X by having the same set of beliefs, but adding more goals, adding new plans for these goals, and modifying some or all of the existing plans for the existing goals. Definition 2 states that if an agent family Y has more plans than an agent family X then Y can be obtained by inheritance from X . The point is that re-defining all the inherited plans may leave the specifications of X and Y totally unrelated (no code is shared) apart from their relative number of plans. The specifications of X and Y only share code if at least

some inherited plans are not re-written in Y . Of course, the desirable situation arises when there is a maximal amount of code sharing in the system, i.e., when some new plans are added and none of the inherited plans is re-defined. This leads to the notion of *incremental* transitions. Incremental transitions only extend the existing family specifications without re-defining them in any way. We consider two useful inheritance mechanisms. The first mechanism (Definitions 2 and 3) allows incremental addition of plans for new and existing goals. The second mechanism (Definitions 2 and 4) allows incremental addition of plans only for new goals.

Definition 3 $X \dashrightarrow_{plan} Y$ iff $X \xrightarrow{\delta}_{mod} Y$ and $P_X \subseteq P_Y$. ■

The relation $\dashrightarrow_{plan}$ allows family specifications to be extended by the addition of new plans. The granularity of $\dashrightarrow_{plan}$ is a single plan since any new plan can be added. Another possible choice is the granularity of sets of plans, i.e., the set of plans for an existing goal cannot be incrementally extended. The relation $\dashrightarrow_{set}$ is the subset of all incremental transitions of $\dashrightarrow_{mod}$ at the granularity of sets of plans.

Definition 4 $X \dashrightarrow_{set} Y$ iff $X \dashrightarrow_{plan} Y$ and $\forall p \in P_Y, p \notin P_X \implies p_{inv} \cap \{q_{inv} : q \in P_X\} = \emptyset$ where $p_{inv} = \pm!g(v)$. ■

Under this definition only the plans written to achieve new goals can be added. For instance, a new plan with the invocation condition $+!landed(V)$, $+!checkPos(V)$ or $+rain(V)$ cannot be added in Example 1.

3.2 Behavioural equivalence

Next, we consider the notion of behaviour of an agent family. As discussed earlier, one can capture the notion of behaviour in many ways depending on the purpose of a particular application domain. We need some way of comparing the behaviours of derived agent families with the behaviours of the original agent families. In this paper, we consider two alternatives. We say that two agent families are behaviourally equivalent if: (a) they have the same public goals; or (b) their instances can achieve the same goals, when placed in the same environment. We consider each of these alternatives more formally.

Definition 5 Let X and Y be agent families. $X \equiv_1 Y$ iff $G_X = G_Y$ ($X \leq_1 Y$ iff $G_X \subseteq G_Y$). ■

Definition 5 is a simple definition of behavioural equivalence which states that agent families X and Y have equivalent behaviour ($X \equiv_1 Y$) if they have the same public goals. Hence, their instances have the potential to achieve the same goals. However, instances of X and Y may achieve the same goals under very different conditions (since their plans may be different). This definition corresponds to the OOP notion of type-equivalence

based on interfaces. It can be shown that $\equiv_1$ is an equivalence relation, it is reflexive, symmetric and transitive.

The second definition of behavioural equivalence is concerned with more details about exactly under which conditions are particular goals achieved by an agent. It uses the notion of *traces*. Assume there is an external observer who records the sequence of all the events occurring in the environment of an agent. At every point in time the observer has noted down a finite trace of a possible evolution of this concurrent agent/environment system. The (infinite) set of all such finite sequences of events is called the set of *traces* of the system and is denoted by $traces(x||E)$ where x is an agent instance, E is the environment (non-deterministic) and $||$ is the concurrency operator. Placing an agent into an environment determines the agent's initial beliefs, its initial goals and its subsequent behaviour. Different instances of an agent family yield the same set of traces if placed in the same environment, but may exhibit different sets of traces if placed in different environments. We define $traces(X)$ as the union of all $traces(x||E)$ for all possible E and x .

Example 4 Two possible traces of the system in Example 1 are: $tr_1 = \langle +!landed(base), +pos(5), +sun(strong), +pos(3), +wind(weak), +pos(1), +landed(base) \rangle$ and $tr_2 = \langle +!landed(base), +rain(strong), +pos(5), +wind(strong), +crashed() \rangle$. ■

Definition 6 The trace $s \preceq t$ if s is a prefix of t ($\langle e_1 \rangle \preceq \langle e_1, e_2 \rangle$). Concatenation of two traces is denoted by $\hat{\ } (\langle e_1 \rangle \hat{\ } \langle e_2, e_3 \rangle = \langle e_1, e_2, e_3 \rangle)$. We use the standard *interleaving* semantics of concurrent events. ■

Definition 7 Let x, y be arbitrary instances of X, Y . Let $goal_traces_{!g(v)}(x||E) = \{s : s \hat{\ } (+g(v)) \in traces(x||E)\}$ where $!g(v) \in G_X$ is the goal to be achieved. Then, $X \leq_2 Y$ if for any environment E , whenever $s \in goal_traces_{!g(v)}(x||E)$ either:

- a) $\exists t \in goal_traces_{!g(v)}(y||E)$ such that $t \preceq s$; or
- b) $\exists t \in traces(y||E) : s \preceq t$, and for each such t
 $\exists u \in goal_traces_{!g(v)}(y||E) : s \preceq u \preceq t \vee t \preceq u$.

Also, $X \equiv_2 Y$ iff $X \leq_2 Y$ and $Y \leq_2 X$. Again, it can be checked that $\equiv_2$ is reflexive, symmetric and transitive. ■

This definition of behavioural equivalence requires that all instances of two agent families can achieve the same goals if placed in the same environment (the goals may be achieved by adopting different plans, and thus generating different internal events). An observer records events occurring in the environment until the achievement of the goal (or failure). The set $goal_traces_{!g(v)}$ contains all traces which lead to the achievement of goal $!g(v)$ ¹. The trace tr_1 from Exam-

ple 4 (without the last event $+landed(base)$) is an element of $goal_traces_{!landed(base)}$ of an instance of *SimplePlane*, while tr_2 is not. An agent family Y preserves and extends the behaviour of an agent family X ($X \leq_2 Y$) if whenever an instance x of X achieves the goal then an instance y of Y (assuming y is in the same environment as x) either: (a) has already achieved the goal; or (b) independently of what happens in the environment after this point, y eventually achieves the goal.

Example 5 Recalling the examples of Section 2 we have: $SimplePlane \leq_2 NewPlane$ (*NewPlane* can land in poor visibility), and $NewPlane \leq_2 AdvancedPlane$ (*AdvancedPlane* can also land in bad weather). ■

3.3 Behaviour preservation

Having defined the inheritance mechanism and behavioural equivalence we examine their relationship by revisiting the notions of behaviour preservation and inheritance anomaly. Behaviour preservation expressed in a general form addresses the soundness issue. It ensures that an incremental transition from one syntactic specification to another leads to a related behaviour (the original behaviour is preserved and extended, as given by $\leq$).

Definition 8 Let $a, b \in Spec, \delta \in \Delta$ and let $\leq$ be an ordering on behaviours. Inheritance mechanism $(Spec, \Delta, \rightarrow, \dashrightarrow)$ is *behaviour preserving* under $\leq$ if $\forall a, b, a \dashrightarrow^\delta b \implies a \leq b$. ■

Definition 8 is specialised to AOP by considering specific inheritance mechanisms and specific behavioural equivalences. Behaviour preservation also depends on the exact semantics of the chosen language. Theorem 1 shows that the inheritance mechanism which allows new plans to be added for new and existing goals is behaviour preserving only under our simpler definition $\leq_1$ (for counter-example under $\leq_2$ see Example 2).

Theorem 1 ($Spec_{AOP}, \Delta_{AOP}, \rightarrow_{mod}, \dashrightarrow_{plan}$) is *behaviour preserving* under $\leq_1$.

Proof: $X \dashrightarrow_{plan}^\delta Y \implies G_X \subseteq G_Y \implies X \leq_1 Y$. ■

The inheritance mechanism where plans can be added only for new goals is behaviour preserving under both characterisations of behavioural equivalence.

Theorem 2 The mechanism $(Spec_{AOP}, \Delta_{AOP}, \rightarrow_{mod}, \dashrightarrow_{set})$ is *behaviour preserving* under $\leq_1$ and under $\leq_2$.

Proof: Let $X \dashrightarrow_{set}^\delta Y$ and $s \in goal_traces_{!g(v)}(x||E)$. The trace of s contains $+!g(v)$ which triggered some plan $q \in P_X$ to execute. Since $P_Y = P_X \cup \delta$, and since $\{p_{inv} : p \in P_X\} \cap \{p_{inv} : p \in \delta\} = \emptyset$, the same plan $q \in P_Y$ was executed in y . It follows by similar arguments that $s \in goal_traces_{!g(v)}(y||E)$. Hence, $goal_traces_{!g(v)}(x||E) \subseteq goal_traces_{!g(v)}(y||E)$. Also, $X \leq_2 Y$ by letting $t = s$ in Case (a) of Definition 7. The proof for $\leq_1$ is similar to Theorem 1. ■

¹We assume that an agent is never asked to achieve a goal which is not in the set of its public goals.

The results of Theorems 1 and 2 seem to suggest that the inheritance mechanism employing the incremental additions at the granularity of sets of plans is the better solution. The next section will make a similar comparison with respect to the problem of inheritance anomalies.

3.4 Inheritance anomaly

Inheritance anomaly, in its general form, addresses the completeness issue. An inheritance mechanism is anomaly-free if all related behaviours can be derived by incremental transitions, thus maximally sharing code.

Definition 9 Let $a, b \in Spec$ and $\leq$ be an ordering relation on behaviours. Inheritance mechanism $(Spec, \Delta, \rightarrow, \dashrightarrow)$ is *anomaly-free* under $\leq$ if $\forall a, b. a \leq b \implies \exists \delta \in \Delta^* \text{ and } c \in Spec \text{ such that } a \dashrightarrow^\delta c \text{ and } b \equiv c$, where $\equiv$ is the equivalence relation induced by $\leq$. ■

Definition 9 requires that any related behaviour of a must be reachable from a by incremental transitions. The desired behaviour of b is reached by inheriting and extending a to obtain c . If this requirement cannot be satisfied for some a and b then we have an occurrence of the anomaly. Note that in general c and b are different implementations of the same behaviour. Similarly to behaviour preservation, the anomaly is highly language dependent, and also depends on the definition of $\equiv$.

Theorem 3 The inheritance mechanism $\rightarrow_{mod}$ is anomaly-free under $\leq_1$. This holds for both $\dashrightarrow_{plan}$ and $\dashrightarrow_{set}$. The proof follows from Definitions 3 and 4. ■

Both inheritance mechanisms suffer from inheritance anomalies under $\leq_2$, as shown in Example 3. However, the choice of the granularity of sets of plans ($\dashrightarrow_{set}$) leads to more anomalies than the finer granularity of $\dashrightarrow_{plan}$.

Theorem 4 Every inheritance anomaly that occurs in $\rightarrow_{mod}$ with $\dashrightarrow_{plan}$ also occurs in $\rightarrow_{mod}$ with $\dashrightarrow_{set}$. The converse does not hold. The proof follows from the fact that $\dashrightarrow_{set} \subseteq \dashrightarrow_{plan}$. ■

This theorem illustrates the trade-off in the choice of inheritance mechanisms. The granularity of sets of plans is behaviour preserving, but it leads to more anomalies.

4 Comparison and conclusion

The notion of inheritance has been studied extensively in OOP and is non-existent in AOP. The well-known inheritance mechanisms in OOP languages do not suffer from the problems described in the paper. Inheritance anomaly occurs, and has been researched, in concurrent object-oriented programming [Matsuoka and Yonezawa, 1993]. However, it has not been formally defined.

In this paper, we have introduced the notion of inheritance into AOP and examined the potential benefits it offers in terms of code sharing and modular verification of agent systems. However, there are potentially

serious problems with inheritance in AOP, namely behaviour preservation and inheritance anomaly. In this paper we have provided a formal framework to examine these problems and the trade-offs involved in solving them. We analysed two inheritance mechanisms and two definitions of behavioural equivalence, and showed which of them were behaviour preserving and anomaly-free. The formal framework we provide is generic and can be applied to other agent-oriented languages, such as AGENT0 [Shoham, 1993], PLACA [Thomas, 1994], or CONGOLOG [Lesperance *et al.*, 1995], by altering the definition of specifications ($Spec$), transitions (Δ), and behavioural equivalence. More importantly, we believe that our approach to the introduction of inheritance into AOP lays the preliminary groundwork for rigorous analysis of language issues, such as inheritance, within the framework of multi-agent system research.

References

- [Brazier *et al.*, 1995] F. Brazier, B. D. Keplicz, N. R. Jennings, and J. Treur. Formal specification of multi-agent systems: a real-world case. In *ICMAS-95*, pages 25–32, California, 1995. AAAI Press/MIT Press.
- [Georgeff and Lansky, 1986] Michael P. Georgeff and Amy L. Lansky. Procedural knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation*, 74:1383–1398, 1986.
- [Kinny *et al.*, 1996] D. Kinny, M. P. Georgeff, and A. S. Rao. A methodology and modelling technique for systems of BDI agents. In *MAAMAW'96*, LNAI 1038, pages 56–71, The Netherlands, 1996. Springer-Verlag.
- [Lesperance *et al.*, 1995] Y. Lesperance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a logical approach to agent programming. In *IJCAI'95 Workshop (ATAL)*, LNAI 1037, pages 331–346, Montréal, Canada, 1995. Springer-Verlag.
- [Matsuoka and Yonezawa, 1993] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in COOP*. MIT Press, 1993.
- [Rao, 1996] Anand S. Rao. AgentSpeak(L): BDI Agents speak out in a logical computable language. In *MAAMAW'96*, LNAI 1038, pages 42–55, 1996.
- [Shoham, 1993] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
- [Thomas, 1994] S. Rebecca Thomas. The PLACA agent programming language. In *ECAI/ATAL-94*, LNAI 890, pages 355–370, The Netherlands, 1994.
- [Wooldridge and Jennings, 1994] M. J. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: A survey. In *ECAI/ATAL-94*, LNAI 890, pages 2–39, The Netherlands, 1994. Springer-Verlag.