Reuse and Abstraction in Verification: Agents Acting in Dynamic Environments

Catholijn M. Jonker[*] Jan Treur[*] Wieke de Vries[‡]

[*] Department of Artificial Intelligence, Vrije Universiteit Amsterdam,
De Boelelaan1081a, 1081 HV Amsterdam, The Netherlands
{jonker, treur}@cs.vu.nl
[‡] Institute of Information and Computing Sciences, Universiteit Utrecht,
PO Box 80.089, 3508 TB Utrecht, The Netherlands
wieke@cs.uu.nl

Abstract. To make verification a manageable part of the system development process, comprehensibility and reusability of properties and proofs is essential. The work reported in this paper contributes formally founded methods that support proof structuring and reuse. Often occurring patterns in agent behaviour can be exploited to establish a library containing properties and proofs. This is illustrated here by verifying the class of single agents acting in dynamic environments. First, a notion of abstraction for properties and proofs is introduced that provides means to structure and clarify verification. Also, the paper contributes to establishing the library by proposing a reusable system of generic co-ordination properties for applications of agents acting in dynamic environments.

1 Introduction

Verification is an important part of agent-oriented software engineering, because it is the only way to guarantee that demands made on aspects of the system behaviour are satisfied. The high degree of complexity of agent system behaviour is as much the reason as the problem here: by simply checking the code of the agent system or by testing, proper behaviour can never be sufficiently established. Proper functioning is often crucial, because agent systems are increasingly employed in circumstances where mistakes have important consequences, for example in electronic commerce. But verification of agent systems is generally not an easy task. As agents may operate in a world that is constantly changing, and agent systems can consist of a number of interacting but independent agents, expressing behavioural requirements may lead to complex formulae. Therefore verification of agent systems is hardly ever done in practice.

So, means are needed to make verification of agent systems manageable. Developers of agent systems should be enabled to verify the system they are building, assisted by tools, even if they are not specialists in formal theory. Properties and proofs have to be intuitively clear to the verifier and even, at least to some degree, to

the stakeholder(s) of the system, as verification results are part of the design rationale of the system. Also, time complexity of the verification process has to be controlled.

This paper discusses some principles that contribute to the support of verification of agent systems. These principles can be used for all agent systems, but here, they are applied in the context of single agents that performs actions in dynamic environments.

In [6] a compositional verification method was introduced. Verifying in a compositional manner supports reuse of verification results and limits the complexity of the process, by making proofs more local. In [1] it was shown how this method can be applied to prove properties of a system of negotiating agents.

However, this does not solve all problems. To manage the complexity of the proofs, and to make their structure more transparent, additional structuring means and reuse facilities are necessary, extending the method of compositional verification. This paper contributes two manners to support proof structuring and reuse.

On the one hand a *notion of abstraction* is introduced that facilitates structuring of properties and proofs. To this end, the language to describe properties of agent systems is extended with new, more abstract, constructs. Parts of formulas can be given an intuitively enlightening name. This leads to a more informal look and feel for properties and proofs, without losing any formal rigour. The abstract notions form a higher-level language to describe system behaviour. The terminology of this language abstracts away from details of the system design, and is closer to the way human verifiers conceptualise agent system behaviour. There are a number of benefits:

- Properties and proofs are more readable and easier to understand.
- Coming up with properties and proofs becomes easier, as the words chosen for the abstracted formulas guide and focus the cognitive verification process of the verification engineer, providing clean-cut agent concepts.
- Verification becomes explainable, as part of the design rationale documentation of a system.

On the other hand, common characteristics of agent systems can be exploited to support reuse. With the paradigm of agents, a range of agent concepts is associated. For example, most agents receive observations and communicated information from their environment and perform actions to manipulate their environment. For this to yield desired results, proper co-ordination with the environment is essential. Properties regarding this apply to many agent systems and thus are highly reusable. Support of reuse requires that a library of predefined templates of properties and proofs is available. By identifying generic elements in the structure of proofs and properties, *reusable systems of properties and proofs* can be constructed. To illustrate this, this paper proposes a system of co-ordination property properties for applications of agents acting in dynamic environments. The properties and proofs of this system are an example of the contents of the verification library. Some advantages of reuse are:

- Verification becomes faster. Often, the verification engineer only has to look up suitable properties and proofs from the verification library and customise these by instantiation.

- Verification becomes easier. The contents of the library are usually phrased using abstraction, so properties and proofs are more intuitively clear, making them more easy to use.

In the following section, the generic system consisting of an agent acting in a dynamic environment is sketched. For this application, a system of co-ordination properties is given in Section 4. But first, Section 3 presents the two languages to describe system behaviour, the detailed language and the abstract language, and the connection between them. In Section 5, the abstraction mechanism is applied; abstract predicates are introduced for parts of properties, yielding an abstract language. In Section 6, proofs are briefly discussed. Finally, Section 7 concludes.

2 The Domain of Agents Acting in a Dynamic Environment

In this section the characteristics of the application class of an agent in interaction with a dynamic environment are briefly discussed. A reusable system of properties for this class will be presented later on, describing correct co-ordination of the agent with its environment.

Agents that can perceive and act in a dynamic environment are quite common. An example is an agent for process control (e.g. in a chemical factory). For this class of single agent systems, an important property is *successfulness of actions*. This means that all actions the agent initiates in its environment yield their expected effects. Because this property is to be proven for a class of systems, it is needed to abstract from domain-dependent details of systems and give a generic architecture that defines the class.

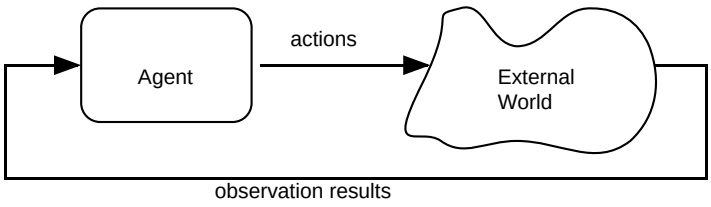

Fig. 1. Agent and external world in interaction

The specification of a generic architecture for a single agent in a dynamic world depicted in Figure 1 consists of two components in interaction: an agent (Ag) and the external world (EW). Only a few aspects of the functioning of the system are specified by the architecture. The agent generates actions that are transferred from its output interface to the external world, and the external world generates observation results that are transferred to the input interface of the agent. Based on the observation results the agent is to decide which actions are to be performed.

The system employs a formal language internally. This language is an order-sorted predicate logic. The *input interface* of the agent is defined by the formal ontology observation results containing a binary relation observation_result. Formulae that can be

expressed using the information type observation results are, for example, observation_result(at_position(self, p0), pos), or observation_result(at_position(self, p1), neg).

The *output interface* of the agent is defined by the formal ontology actions to be performed based on the sort ACTION and the unary relation to_be_performed. For example the statement to_be_performed(goto(p)) can be expressed in this ontology. For the external world the input and output interfaces are the opposite of the agent's interfaces.

Realistic characteristics of the agent systems in the class defined above are:

- perceptions take time
- the generation of actions takes time
- execution of actions in the world takes time
- unexpected events can occur in the environment

Proving successfulness of actions under these circumstances is intricate because an action can only succeed when its execution is not disturbed too much. If two executions of actions overlap or evens happen during executions, actions could fail. Also, while an agent is observing and reasoning, the situation in the world might change. A system of co-ordination properties will be proposed that takes all these influences into account.

In the literature, varying attitudes towards these disturbances can be found. In one part of the literature (e.g., standard situation calculus, as described in [9; 11]), these disturbances are excluded in a global manner, e.g., action generation and execution have no duration at all and no events occur at all. The problem with these global assumptions is that they violate the characteristics of most of the application domains. Some literature takes into account duration of action execution (e.g., [13]). Literature that also takes into account the reasoning and decision processes in action generation is very rare. Another lack in the literature is that most authors don't try to verify implemented systems; they only state theories regarding actions, without relating them to practical system engineering.

3 Temporal Models and Temporal Languages

For phrasing properties, a language is needed. Behaviour is described by properties of the execution traces of the system. In this section, the language used for this is introduced. Also, this section introduces the language abstraction formalism.

3.1 Basic Concepts

By adding a formalisation of time to the language internally used in the generic system, a formal language is obtained to formulate behavioural properties. This language is still semantical in nature; properties of traces are described in a direct manner. A formal logic could be added, but this is not essential for our purposes.

The state language SL(D) of a system component D is the (order-sorted) predicate logic language based on the interface ontologies of D. The formulae of this language

are called *state formulae*. An *information state* M of a component D is an assignment of truth-values {true, false, unknown} to the set of ground atoms in SL(D). The set of all possible information states of D is denoted by IS(D).

The time frames are assumed *linear with initial time point* 0. Time frames must be discrete; using dense time frames is also possible, as long as some constraints are obeyed. A *trace* $\mathfrak{M}$ of a component D over a time frame T is a sequence of information states $(M^t)_{t \in T}$ in IS(D). Given a trace $\mathfrak{M}$ of component D, the information state of the input interface of component C at time point t is denoted by state($\mathfrak{M}$, t, input(C)), where C is either D or a component within D. Analogously, state($\mathfrak{M}$, t, output(C)) denotes the information state of the output interface of component C at time point t.

These information states can be related to formulae via the satisfaction relation $\models$. If φ is a state formula expressed in the input ontology for component C, then

state($\mathfrak{M}$, t, input(C)) $\models$ φ

denotes that φ is true in this state at time point $t \in T$

These statements can be compared to *holds*-statements in situation calculus [9]. A difference, however, apart from notational differences, is that we refer to a trace and time point, and that we explicitly focus on part of the system. Based on these statements, which only use predicate symbol $\models$, behavioural properties can be formulated in a formal manner in a sorted predicate logic with sorts T for time points, Traces(C) for traces of component C and F for state formulae. The usual logical connectives such as $\neg$, $\wedge$, $\Rightarrow$, $\forall$, $\exists$ are employed to construct formulae, as well as < and = (to compare moments in time). The language defined in this manner is denoted by TL(D) (Temporal Language of D). An example of a formula of TL(S), where S refers to the whole system, is:

$\forall \mathfrak{M} \in$ Traces(S):
$\forall$ t1 : state($\mathfrak{M}$, t1, output(Ag)) $\models$ to_be_performed(A) $\Rightarrow$
$\exists$ t2 > t1 : state($\mathfrak{M}$, t2, output(Ag)) $\models$ to_be_performed(B)

This expresses that every decision of Ag to do action A is always followed by a later decision to do B.

The languages TL(D) are built around constructs that enable the verifier to express properties in a detailed manner, staying in direct relation to the semantics of the design specification of the system. For example, the state formulae are directly related to information states of system components. But the detailed nature of the language also has disadvantages; properties tend to get long and complex. The formalism of abstraction, described in Section 3.2, alleviates this considerably.

3.2 The Language Abstraction Formalism

Experience in nontrivial verification examples has taught us that the temporal expressions needed in proofs can become quite complex and unreadable. Also, details of the formalisation blur the generic agent concepts in properties. As a remedy, new language elements are added as abbreviations of complex temporal formulae. These new language elements are defined within a language AL(D) (meaning Abstract Language of component D). As a simple example, for the property that there is action

execution starting in the world at t a new predicate `ActionExStarting` can be introduced. Then the property can be expressed in the abstracted language:

`ActionExStarting`(A, t, EW, $\mathcal{M}$)

which is interpreted as:

state($\mathcal{M}$, t, input(EW)) $\models$ to_be_performed(A)

Semantics of these new language elements is defined as the semantics of the detailed formulae they abstract from. In logic the notion of *interpretation mapping* has been introduced to describe the interpretation of one logical language in another logical language, for example geometry in algebra (cf. Chapter 5 in [5]). The languages AL(D) and TL(D) can be related to each other by a fixed interpretation mapping from the formulae in AL(D) onto formulae in TL(D).

The language AL(D) abstracts from details of the system design and enables the verifier to concentrate on higher level (agent) concepts. Proofs can be expressed either at the detailed or at the abstract level, and the results can be translated to the other level. Because formulae in the abstract level logic can be kept much simpler than the detailed level logic, the proof relations expressed on that level are much more transparent.

4 Properties for Proving Successfulness of Actions

In Section 4.1, an informal introduction to the system of co-ordination properties is given. The system itself appears in Section 4.2.

4.1 Approaching the Problem of Co-ordination of Actions

For the application class described in Section 2, the aim is to prove that under the specified assumptions all actions executed in the agent system are successful, that is, yield all of their effects. To arrive at a reusable and intuitively pleasing proof, it was necessary and illuminating to separate the different aspects into a number of properties. These will constitute the *system of co-ordination properties*. In this section, some important aspects are described informally.

An action succeeds when its execution renders the appropriate effects. This effect has to happen during the execution of the action to be recognisable as an effect of that particular action. We assume there is some means to detect the end of an action execution. Just like the start of an execution of A is indicated by a to_be_performed(A)-atom, the end is indicated by an ended(A)-atom.

Action executions can fail because of three reasons. The first reason is overlapping of action executions. So, to guarantee success, overlapping should not happen. Property COORD0 formalises this. This property is proved from other properties of the agent and the world using induction, but the proof is left out. Figure 2 illustrates COORD0.

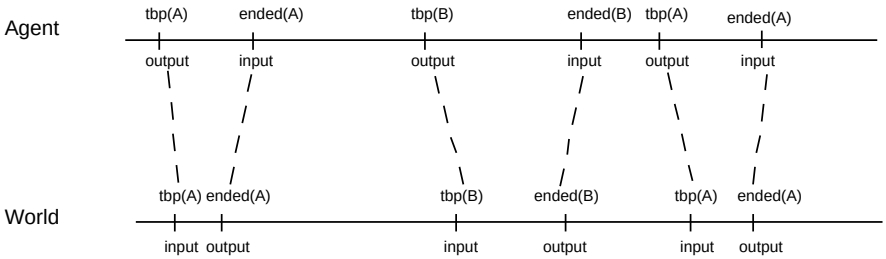

Fig. 2. No overlapping of executions

Note that COORD0 also enables identification of action execution.

But action executions can also fail due to events. In our view, events are changes to the world state that aren't controlled by the agent. These can be due to the dynamics of the world itself (natural events). But changes due to the aftermath of a failed action also are events. So, each change to the world state either is the effect of a successful action or it is an event. In some other approaches, events designate all causes of state changes, including agent-initiated actions. This might be confusing, but is just a matter of definitions.

Property COORD3 is a property of the world, stating that no events happen during action execution. This is quite a strong demand, as not every event might interfere with the action. The demand can be relieved by only forbidding interfering events, which is a minor extension.

A third reason for action failure is that the world can change prior to an action execution starting in the world. Between the moment the world situation arises that gives the agent reason to decide to do an action and the start of the execution of this action, events or other action effects could occur, disrupting the applicability of the action. Property COORD5 states that all actions are still applicable the moment their execution starts in the world.

In the following, it is shown that under the assumptions associated with the class of systems considered here, absence of these three causes for failure is sufficient to prove successfulness of actions.

4.2 The System of Co-ordination Properties

One of the main objectives of this paper is to establish a set of properties that enable the verifier to prove that all actions executed will succeed. The system of co-ordination properties presented here provides a clear separation of all aspects involved. The properties are phrased in the abstract languages AL, although the interpretation of this language in the detailed language is presented later, to show the intuitive power of the abstract languages.

The system is structured in the following way: COORD1 is the target property, formalising action successfulness, COORD0 is the foundational property, parts of which are frequently used as condition in other co-ordination properties, COORD2, -3, -4 and -5 are used to prove COORD1.

COORD0 is the foundation of the system of co-ordination properties. It enables the verifier to identify action executions, by formalising Figure 2. The property states that action executions don't overlap, not in the world and neither in the agent. The agent- and the world-part will be part of the conditions of many properties to come, to enable identification of action executions. This is the abstract formula:

COORD0:
$\forall\, \mathcal{M} \in$ Traces(S)
NoOverlappingInWorld($\mathcal{M}$) $\wedge$
NoOverlappingInAgent($\mathcal{M}$)

COORD1 formalises action successfulness, stating that all action executions of the system are applicable and yield all expected effects. Informally:

> "When an action execution in the world begins at t1 and ends at t2,
> then
> the action is applicable at t1 in the world
> and
> all expected effects of the action in that world situation will be realised during the execution."

This is the abstract formalisation:

COORD1:
$\forall\, \mathcal{M} \in$ Traces(S) $\forall$ A $\in$ ACTION $\forall$ t1 $\forall$ t2 > t1 :
(ActionEx(A, t1, t2, EW, $\mathcal{M}$) $\Rightarrow$
Appl(A, t1, $\mathcal{M}$) $\wedge$
ExpEffectsHappen(A, t1, t2, EW, $\mathcal{M}$))

It is essential that all action executions are applicable at the moment they start in the world. When actions are applicable at the moment the execution starts, the expected effects of the action are the desired effects. When an action is not applicable, there might be no effects at all, or unwanted ones.

COORD2 is an auxiliary property stating that all action executions started at times that the action is applicable will be successful. Informally:

> "When an action execution in the world begins at t1 and ends at t2,
> and
> when the action is applicable at t1 in the world
> then
> all expected effects of the action in that world situation will be realised during the execution."

And this is its abstract formalisation:

COORD2:
$\forall\, \mathcal{M} \in$ Traces(S) $\forall$ A $\in$ ACTION $\forall$ t1 $\forall$ t2 > t1 :
ActionEx(A, t1, t2, EW, $\mathcal{M}$) $\wedge$
Appl(A, t1, $\mathcal{M}$) $\Rightarrow$
ExpEffectsHappen(A, t1, t2, EW, $\mathcal{M}$)

COORD3 is a demand on the world that states that there are no events happening during action executions. Informally:

> "If there is an action execution in the world
> and
> action executions do not overlap
> then
> no events happen during the execution."

And this is the formalisation in the abstract language:

COORD3:

$$\forall\, \mathcal{M} \in \text{Traces(EW)}\ \ \forall\, A \in \text{ACTION}\ \ \forall\, t1\ \ \forall\, t2 > t1 :$$
$$\texttt{ActionEx}(A, t1, t2, EW, \mathcal{M}) \wedge \texttt{NoOverlappingInWorld}(\mathcal{M}) \wedge \texttt{NoEventsDuring}([t1, t2], EW, \mathcal{M}) \Rightarrow$$

COORD4 is a demand on the world that says that an action execution in the world will be successful when the action is applicable and there are no disturbances caused by overlapping executions or events. These are all conditions for action success, as long as the world satisfies COORD4. Informally:

> "If an action execution in the world begins at t1 and ends at t2
> and
> action executions do not overlap
> and
> no events happen during the execution
> and
> the action is applicable at t1
> then
> all effects of the action will be realised during the execution."

This is the formalisation:

COORD4:

$$\forall\, \mathcal{M} \in \text{Traces(EW)}\ \ \forall\, A \in \text{ACTION}\ \ \forall\, t1\ \ \forall\, t2 > t1 :$$
$$\texttt{ActionEx}(A, t1, t2, EW, \mathcal{M}) \wedge \texttt{NoOverlappingInWorld}(\mathcal{M}) \wedge \texttt{NoEventsDuring}([t1, t2], EW, \mathcal{M}) \wedge \texttt{Appl}(A, t1, \mathcal{M}) \Rightarrow \texttt{ExpEffectsHappen}(A, t1, t2, EW, \mathcal{M})$$

COORD5 simply states that an action is applicable at the moment its execution starts. This is a necessary condition for success of this action. This is its formalisation:

COORD5:

$$\forall\, \mathcal{M} \in \text{Traces(S)}\ \ \forall\, A \in \text{ACTION}\ \ \forall\, t1\ \ \forall\, t2 > t1 :$$
$$\texttt{ActionEx}(A, t1, t2, EW, \mathcal{M}) \Rightarrow \texttt{Appl}(A, t1, \mathcal{M})$$

Because the abstraction formalism is exploited, these properties are relatively easy to read and understand, even without knowing the formal meaning of abstract terms, which is provided in the next section. Technical details are hidden beneath intuitively

clear notions. The clarity and brevity of the formulas make the verification process more manageable, as the abstract concepts yield a natural view of the system's behaviour and prevent getting lost in symbolic clutter while constructing proofs.

The system of co-ordination properties is applicable for many systems with a single agent that performs actions in a changing world. By simple instantiation of the system specific details, such as the set of actions, the conditions of applicability and the effects of these actions, the system can be customised.

5 Abstract Formulations

In this section, a number of predicates of the abstract language are defined. The abstract language enables the verifier to express temporal properties of system behaviour using a vocabulary of clean-cut concepts. To be able to distinguish elements of the abstract language, a `different font` is used to denote them.

But first, some auxiliary abbreviations are introduced. All relate to *changes* in the system state. The ⊕-notation, pronounced as *just*, is used to denote a change to a certain information state. The symbol ≡ means "is defined as".

$$\oplus\text{state}(\mathfrak{M}, t1, \textit{interface}) \models \varphi \quad \equiv \quad \text{state}(\mathfrak{M}, t1, \textit{interface}) \models \varphi \quad \wedge$$
$$\exists\, t2 < t1\ \forall\, t:\ (t2 \le t < t1 \ \Rightarrow\ \text{state}(\mathfrak{M}, t1, \textit{interface}) \not\models \varphi)$$

The definition of $\oplus\text{state}(\mathfrak{M}, t1, \textit{interface}) \not\models \varphi$ is analogous. Closely related is the $\otimes t1, t2\oplus$-notation, defined as follows:

$$\otimes t1, t2\oplus\text{state}(\mathfrak{M}, t2, \textit{interface}) \models \varphi \quad \equiv \quad \oplus\text{state}(\mathfrak{M}, t2, \textit{interface}) \models \varphi \ \wedge$$
$$\forall\, t:\ (t1 < t < t2 \ \Rightarrow\ \neg\, \oplus\text{state}(\mathfrak{M}, t, \textit{interface}) \models \varphi)$$

Again, an analogous definition can be given for the variant with $\not\models$ instead of $\models$. This notation can be used to say that the information state has just changed in some way at t2, for the first time since t1.

All further definitions concern elements of the languages AL. Some notions are formally defined; others are only informally sketched.

Concerning action executions

The notion of an *action execution* is central to the system of co-ordination properties, so formalisation is desired. Both for the world and the agent, an action execution is defined to happen between t1 and t2 when a tbp-atom appears at t1 and the first matching ended-atom appears at t2. These definitions only yield the right intuitions when property COORD0 holds. If not, then it is not reasonable to take on the first matching ended-atom as belonging to the tbp-atom, as it could be the end of an earlier executed instance of the same action.

Action executions are defined for the world as well as for the agent. First, new predicates are introduced and explained in informal terms. Next, formal interpretations in terms of the detailed language are given.

Let A ∈ ACTION and t1, t2 > t1 be moments in time. Then, the abstract formula

ActionEx(A, t1, t2, EW, $\mathcal{M}$) denotes that there is an execution of A in the world starting at t1 and ending at t2.

Interpretation in terms of the detailed language:

ActionEx(A, t1, t2, EW, $\mathcal{M}$) ≡ ⊕state($\mathcal{M}$, t1, input(EW)) ⊨ to_be_performed(A) ∧
⊗t1,t2⊕state($\mathcal{M}$, t2, output(EW)) ⊨ ended(A)

The predicate ActionEx(A, t1, t2, Ag, $\mathcal{M}$) is the corresponding agent notion.

Concerning applicability

Actions can only be successfully executed in certain world states. There must be nothing obstructing the execution of the action. For each action A, the existence of a formula appl(A) is assumed, describing exactly the world situations in which the action can be fruitfully executed. It is not excluded that the effects of the action are already present in these world situations. Now, applicability can be defined straightforwardly:

Let A ∈ ACTION and t1 be a moment in time. Then, the abstract formula

Appl(A, t1, $\mathcal{M}$) denotes that action A is applicable in the world at t1.

Interpretation in terms of the detailed language:

Appl(A, t1, $\mathcal{M}$) ≡ state($\mathcal{M}$, t1, output(EW)) ⊨ appl(A)

Concerning expected effects

When an execution of an action A starts at t1 in the world, the effects expected depend on the factual world situation at t1. So, the following notion takes into account the output information state of EW, at the time the execution starts.

Let A ∈ ACTION, l ∈ groundliterals(world info) and t be a moment in time. Then, the abstract formula

ExpEffect(l, A,t1, EW, $\mathcal{M}$) denotes that l is expected to become true as a result of executing A in the world at t1.

Concerning effects of actions and events

A literal is defined to be an *effect of an action* when the literal is an expected outcome that becomes true during execution of the action. Note that this doesn't mean that the literal becomes true as a result of the action, though this will be usually the case. But when during an action execution an event happens, which causes changes that are also expected effects of the action being executed, these changes will be seen as effects of the action. This choice is made because there is no means by which an external observer can distinguish changes caused by actions from changes caused by events. A literal is defined to be an *effect of an event* when it is not an effect of any action.

Let A ∈ ACTION, l ∈ groundliterals(world info) and t be a moment in time. Then, the abstract formula

`ActionEff`(A, l , t , EW, $\mathcal{M}$) denotes that at t, l becomes true as a result of executing A.

Interpretation in terms of the detailed language:

`ActionEff`(A, l , t , EW, $\mathcal{M}$) ≡ ∃ t1 < t ∃ t2 ≥ t :
⊕state($\mathcal{M}$, t , output(EW)) ⊨ l ∧
`ActionEx`(A, t1, t2, EW, $\mathcal{M}$) ∧
`ExpEffects`(l, A,t1, EW, $\mathcal{M}$)

Let l ∈ groundliterals(world info) and t be a moment in time. Then, the abstract formula

`EventEff`(l , t , EW, $\mathcal{M}$) denotes that at t, l becomes true as a result of some event.

Interpretation:

`EventEff`(l , t , EW, $\mathcal{M}$) ≡ ⊕state($\mathcal{M}$, t , output(EW)) ⊨ l ∧
¬ ∃ A ∈ ACTION: `ActionEff`(A, l , t , EW, $\mathcal{M}$)

The next abstract formula is used to state that during an interval in time there are no effects of events.

Let *int* be an interval in time. Then, the abstract formula

`NoEventsDuring`(*int*, EW, $\mathcal{M}$) denotes that there are no events taking place in the world during *int*.

Definition within the abstract language:

`NoEventsDuring`(*int*, EW, $\mathcal{M}$) ≡ ∀ l ∈ groundliterals(world info) ∀ t ∈ *int* :
¬ `EventEff`(l , t , EW, $\mathcal{M}$)

Concerning successful actions

The following formula of the abstract language states that an execution of A is successful, meaning that all expected effects are achieved during the execution:

Let A ∈ ACTION and t1, t2 > t1 be moments in time. Then, the abstract formula

`ExpEffectsHappen` (A, t1, t2, EW, $\mathcal{M}$) denotes that all expected effects of doing A between t1 and t2 are achieved.

Definition within the abstract language:

`ExpEffectsHappen` (A, t1, t2, EW, $\mathcal{M}$) ≡ `ActionEx`(A, t1, t2, EW, $\mathcal{M}$) ∧
∀ l ∈ world info ∃ t3 ∈ ⟨t1, t2] :
`ExpEffect`(l, A,t1, EW, $\mathcal{M}$) ⇒
`ActionEff`(A, E , t3 , EW, $\mathcal{M}$)

Concerning overlapping executions

The notions `NoOverlappingInWorld`($\mathcal{M}$) and `NoOverlappingInWorld`($\mathcal{M}$) denote that action executions don't overlap, neither in the world nor in the agent. No formal definitions are given.

6 Proofs

In this section, a complete proof tree of COORD1 is given. In order to prove COORD1, it is possible to stay entirely within the abstract language; no abstractions need to be expanded into the detailed language. This makes the proof very easy.

To prove COORD1, all that is needed is performing simple modus ponens on a subset of the system of co-ordination properties. Figure 3 shows the proof tree:

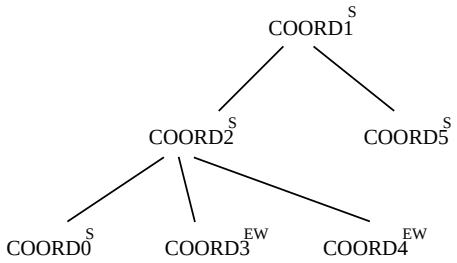

Fig. 3. The proof tree

The proofs of COORD0 and COORD5 are left out. These are more complex, but the really difficult parts of the proof can be done once and for all.

7 Discussion

One of the challenges to improve development methods for agent systems is to provide appropriate support for verification of agent systems being built in practice. The current state is that from the theoretical side formal techniques are proposed, such as temporal logics, but that developers in practice do not consider them useful. Three main reasons for this gap are that

- behavioural properties relevant for agent systems in practice usually have such a high complexity that both fully automated verification and verification by hand are difficult,
- the formal techniques offered have no well-defined relation to design or software specifications of the real systems used in practice, and
- the formal techniques offered require a much higher level of formal skills than the average developer in practice possesses.

This paper addresses these issues in the following manner. Two languages are proposed: a detailed language, with a direct relation to the system design specification, and an abstract language in which properties can be formulated in a more conceptual manner. Both languages have been defined formally; moreover, well-defined relationships exist between the two languages, and between the detailed language and the system design specification. Proof structures can be made visible within the abstract language; by this abstraction, complexity of the property and proof

structures are reduced considerably. More detailed parts of proofs and properties can be hidden in the detailed language, and show up in the abstract language only in the form of, more abstractly formulated, reusable lemmas.

Two roles are distinguished within the verification engineering process: the verification support developer, who defines libraries of reusable properties in the abstract language, and their related properties in the detailed language, and the verification engineer, who uses these libraries to actually perform verification for a system being developed in practice.

The approach has been illustrated by addressing the case of co-ordination of actions. Under realistic assumptions, such as action generation and action execution with duration, it is a complex problem to guarantee the successfulness of action executions. A set of reusable co-ordination properties has been defined both in the detailed language and in the abstract language. Indeed, the abstract formulations are much more accessible and explainable than their detailed counterparts. It has been shown that the abstractly formulated relationships between these properties can be expressed within the abstract languages. The co-ordination properties found have become part of a library of reusable properties that is being developed.

When we try to relate our work to the other contributions to the AOSE workshop, our approach of formally verifying requirements of agent systems seems to be unique. Many papers describe methodologies for designing and analysing multi-agent systems. These methodologies enable system developers to build a well-structured agent system, but they don't offer support to verify whether the behaviour of the resulting system obeys the requirements of the prospective user. For example, in [14], a methodology is proposed that supports the developer "through an entire software development lifecycle from problem description through implementation". From the requirements of the user, which may be informal or formal, goals are distilled which guide the development process. These goals are the essence of the set of requirements, but they don't seem to be formal in nature. So, it's not possible to proof whether the resulting multi-agent system reaches its goals.

Many approaches are based on UML. The work of Odell et al. [10] extends UML with agent concepts. Though UML is graphical in nature, UML models do represent requirements, as is also explicated by Depke et al. in [2]. A UML model semantics is represented by a formal metamodel. According to the authors of [10], logical specifications like we use them could be expressed using features of the metamodel.
Also, this paper mentions templates as being behaviours common to different problem domains. A template is a behaviour pattern that can be instantiated and customised to fit a specific domain. In our paper, we developed a set of generic properties that describe an often-occurring pattern in agent behaviour, namely performing actions in a dynamic environment. This is similar in spirit.

A very prominent concept in the AOSE workshop was the concept of roles. In [8], Kendall represents patterns of interaction using role models. These role models abstract from application details, just as we do in our system of co-ordination properties. In a recent paper by Ferber et al. [4], the relation between formal requirements on dynamic agent system and the dynamics of abstract organisational concepts like roles and groups is explored.

The languages used in this paper are similar to the approach in situation calculus [9]. A difference is that explicit references are made to temporal traces and time

points. In [12], Reiter addresses proving properties in situation calculus. A difference with our approach is that we incorporate arbitrary durations in the decision process of the agent, and in the interaction with the world. Also, we focus on capturing agent concepts in the abstract language, which makes our approach specifically suitable for verifying agent applications.

References

[1] Brazier, F.M.T., Cornelissen, F., Gustavsson, R., Jonker, C.M., Lindeberg, O., Polak, B., and Treur, J., Compositional Design and Verification of a Multi-Agent System for One-to-Many Negotiation, in: *Proceedings of the Third International Conference on Multi-Agent Systems, ICMAS'98*, IEEE Computer Society Press, 1998, pp. 49-56.

[2] Depke, R., Heckel, R., and Küster, J.M., Requirement Specification and Design of Agent-Based Systems with Graph Transformation, Roles and UML, in: *this volume*.

[3] Engelfriet, J., Jonker, C.M., and Treur, J., Compositional Verification of Multi-Agent Systems in Temporal Multi-Epistemic Logic, in: *Pre-proceedings of the Fifth International Workshop on Agent Theories, Architectures and Languages, ATAL'98* (J.P. Mueller, M.P. Singh, and A.S. Rao, eds.), 1998, pp. 91-106. To appear in: *Intelligent Agents V* (J.P. Mueller, M.P. Singh and A.S. Rao eds.), Lecture Notes in AI, Springer Verlag, in press, 1999.

[4] Ferber, J., Gutknecht, O., Jonker, C.M., Müller, J.P., and Treur, J., Organization Models and Behavioural Requirements Specification for Multi-Agent Systems, in: *Proceedings of the Fourth International Conference on Multi-Agent Systems, ICMAS 2000*, IEEE Computer Society Press, in press. Extended version in: *Proceedings of the ECAI 2000 Workshop on Modelling Artificial Societies and Hybrid Organizations*, in press, 2000.

[5] Hodges, W., *Model theory*, Cambridge University Press, 1993.

[6] Jonker, C.M., and Treur, J., Compositional Verification of Multi-Agent Systems: a Formal Analysis of Pro-activeness and Reactiveness, in: *Proceedings of the International Workshop on Compositionality, COMPOS'97* (W.P. de Roever, H. Langmaack, A. Pnueli eds.), Lecture Notes in Computer Science, vol. 1536, Springer Verlag, 1998, pp. 350-380.

[7] Jonker, C.M., Treur, J., and Vries, W. de, Compositional Verification of Agents in Dynamic Environments: a Case Study, in: *Proceedings of the KR98 Workshop on Verification and Validation of KBS* (F. van Harmelen ed.), 1998.

[8] Kendall, E.A., Agent Software Engineering with Role Modelling, in: *this volume*.

[9] McCarthy, J., and Hayes, P.J., Some Philosophical Problems from the Standpoint of Artificial Intelligence, *Machine Intelligence*, vol. 4, 1969, pp. 463-502.

[10] Odell, J., Van Dyke Parunak, H., and Bauer, B., Representing Agent Interaction Protocols in UML, in: *this volume*.

[11] Reiter, R., The Frame Problem in the Situation Calculus: a Simple Solution (Sometimes) and a Completeness Result for Goal Regression, in: *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy* (V. Lifschitz ed.), Academic Press, 1991, pp. 359-380.

[12] Reiter, R., Proving Properties of States in the Situation Calculus, *Artificial Intelligence*, vol. 64, 1993, pp. 337-351.

[13] Sandewall, E., *Features and Fluents. The Representation of Knowledge about Dynamical Systems, Volume I*, Oxford University Press, 1994.

[14] Wood, M., and DeLoach, S.A., An Overview of the Multiagent Systems Engineering Methodology, in: *this volume*.