Modular Multi-Agent Design

Michael Fisher, Louise Dennis and Anthony Hepple

Department of Computer Science, University of Liverpool, Liverpool L69 3BX, UK
EMAIL: {MFisher,L.A.Dennis,A.J.Hepple}@liverpool.ac.uk

Abstract

In this paper our aim is to bring together formal specification, automated verification, dynamic agent organisations, and visual modelling in order to provide a simple, but semantically coherent, framework for designing and developing multi-agent systems.

1 Introduction

We are here concerned with providing a formal design method for multi-agent systems that is, if possible, simple, graphical and flexible. But why do we need another design method? There already exist many (general) *formal design approaches* [35, 28, 29, 42, 33, 31], *agent design methodologies* [45, 46, 3, 24, 37] , *design approaches for software architectures* [43, 1] and *graphical design methods* [27]. However, none of these are entirely satisfactory for the design of reliable multi-agent systems [10, 11, 32, 9].:

- many agent design methods do not explicitly model the structural relationship between agents and those that do tend to represent a static, initial view of the system, that does not support adaptive multi-agent organisations [11];
- they rarely allow modular or compositional view of individual agents, in a way that guarantees adherence to specification statements;
- while the history of formal methods has provided us with many useful, formally based, design methods none of these take into account all the fundamental aspects of agents such as goals, beliefs, actions, teams, norms, roles, etc;
- similarly, software architectures although being more flexible, do not incorporate core concepts that we see as being central to autonomous agents and multi-agent systems;
- and, finally, general methodologies go not have the graphical design notations to support agent/multi-agent concepts and the agent-oriented methodologies that do, often have no direct formal semantics.

Further, in [9], Cabri *et. al* suggest some criteria that a useful multi-agent design methodology should fulfil. These criteria call for an emphasis on the domain (and hence agent) structure, normative interactions (to facilitate adaptation to change), communication and verification support. So, we here aim to address these difficulties by providing a design approach that solves many of these problems and aims to satisfy the criteria mentioned.

We will begin, in Section 2, by considering a simple view of agents, each having an underlying formal specification. We will briefly consider what logics we can use for such specifications. In Section 3, we look at the uses of such individual agent specifications. These include deductive proof, execution, algorithmic checking, refinement, etc., all of which we aim to do *automatically* where possible.

In Section 4, we turn to multi-agent systems and, in particular, their organisational aspects. In our model, a multi-agent system is itself an agent and so we consider the way in which an agent can contain other agents, and the way in which the multi-agent system behaviour is defined by its constituent agents. In particular, the uses of agent specifications given in Section 3 naturally carry over to multi-agent systems. This viewpoint on multi-agent systems provides a naturally modular approach to the design of such systems.

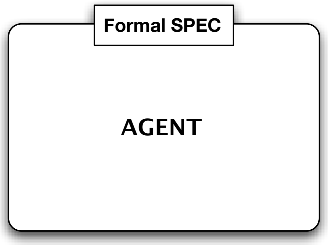
We next move on to consider what happens when agents occurring *within* an agent effectively get behaviour from that agent, as well as providing behaviour for the overall system. So, in Section 5, we modify our notation to incorporate this important aspect, and provide semantics for for the interaction between an agent and its contents. In Section 6 we consider what happens when agents move in/out of contexts and, in particular, how this dynamically affects the formal specification of the multi-agent system.

Finally, in Section 7 we consider concluding remarks, outstanding problems and future directions.

Overall, this work brings together our work on a number of aspects of agents, including: formal specification (time, belief, goals, probability, etc); agent verification via deductive methods; agent verification via model checking; direct execution of formal agent specifications; and abstractions for programming agent organizations. We here bring some of these together to *try to* provide a simple, but semantically coherent, mechanism for designing and developing multi-agent systems. Ideally, this should be: flexible; dynamic; tractable; modular; intuitive; and visualizable!

2 Formal Specification

We are interested in the formal specification of agents. A formal specification *exactly* describes the agent's behaviour, and our graphical representation of this is:



The formal specification is typically a logical formula describing the dynamic, informational and motivational behaviour of the agent. In the BDI approach [40, 39], and in other works on formal agent theory [44, 14, 20], a *wide* variety of logics have been developed and have been used to represent different aspects of agent behaviour. These include modal logics of belief (B), goals (G), wishes (W), desires (D), intentions (I), actions ($[\alpha]$), abilities (A), knowledge (K), etc, all with an underlying temporal ($\Diamond$) and/or probabilistic ($P^{0.3}$) basis. So, given some combination of these logics we can specify the behaviour of the agent formally, e.g:

$$(B_i \Diamond \varphi \land A_i \bigcirc \varphi) \;\Rightarrow\; P^{0.7} \Diamond \varphi$$

which has intended meaning

> *if agent i believes that eventually φ will be satisfied, and agent i is able to satisfy φ in its next state, then there is a 70% probability that agent i will satisfy φ eventually.*

There is much work on the use of different logics, and specifically on different combinations of logics, though we will not discuss the different logical aspects just yet. However, we just note that the formal specification describes the *exact* behaviour of the agent, with respect to the aspects specified. For example, a specification combining temporal and goal descriptions exactly describes the agent's goals, its temporal development, and the temporal aspects of its goals. It does not directly say anything about the agent's beliefs, its probabilistic aspects, or anything else not related to the specification.

So, to begin with, we assume that a specification SPEC_A for an agent A is provided in an appropriate logic, $\mathcal{L}$. Thus, SPEC_A $\in \mathrm{WFF}(\mathcal{L})$. Importantly, we must also be sure that the specification is not inherently inconsistent. Thus we also require that SPEC_A is satisfiable, i.e.:

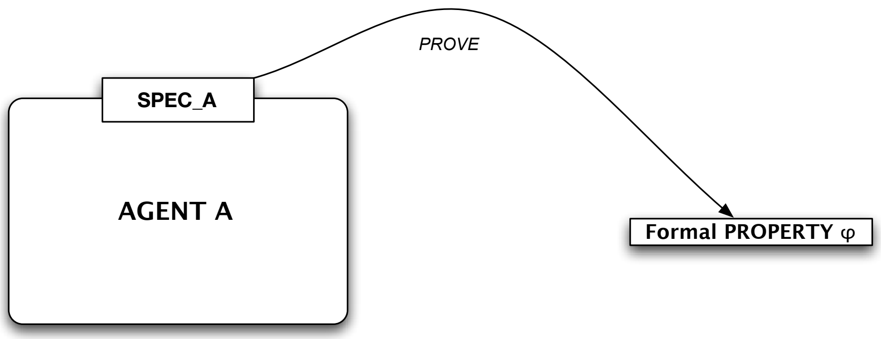
$$\exists \mathcal{M}.\ \mathcal{M} \models_{\mathcal{L}} \mathsf{SPEC\_A}\,.$$

We now consider what we might *do* with such agent specifications.

3 Uses of Agent Specifications

3.1 Proof

A typical use for formal specifications is as an unambiguous description of the agent's behaviour. As such, we can assess what properties hold of the agent behaviour by attempting proof.

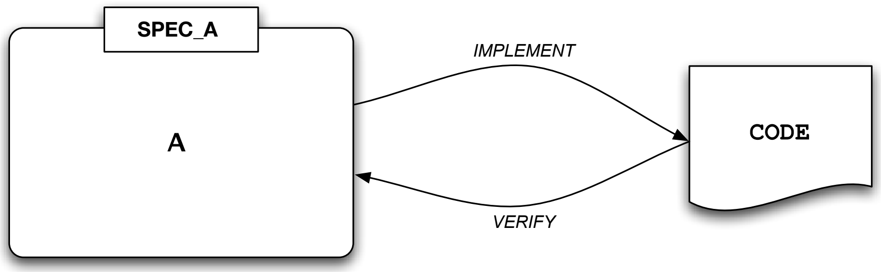


For example, if we can establish that '$\mathsf{SPEC\_A} \Rightarrow \varphi$', where φ is some logical property, then we know that any behaviour of the system satisfies φ. As this is typically assessed deductively then we will need to establish

$$\vdash_{\mathcal{L}} \mathsf{SPEC\_A} \Rightarrow \varphi\,.$$

3.2 Implementation and Verification

There is a strong relationship between the formal specification of the agent and the code that implements this specification.



If some executable code is produced that is intended to implement the agent, then the formal specification *should* provide a semantics (at least a partial one) for that code. In general we require

$$\tau(\mathtt{CODE}) \models_{\mathcal{L}} \mathsf{SPEC\_A}$$

where 'τ' translates CODE to a corresponding semantic model for $\mathcal{L}$.

There are a number of ways in which code can be developed from such a formal specification, including *refinement* (which we will consider later), *synthesis*, and *exploratory* (or extreme/agile) programming.

Program synthesis is the automatic generation of executable programs from high-level specifications [34]. Specifically, *deductive* program synthesis is the derivation of a program from a logical specification, by following deductively correct transformation rules. Much work has been undertaken over the last 20 years on the synthesis of distributed systems from logical specifications, using both deductive and algorithmic methods [13, 30]. Although synthesis procedures have not yet been developed for full agent specifications (incorporating all the temporal, doxastic, probabilistic, goal-related, etc, details we might require), synthesis techniques for cooperative processes are now being developed [41].

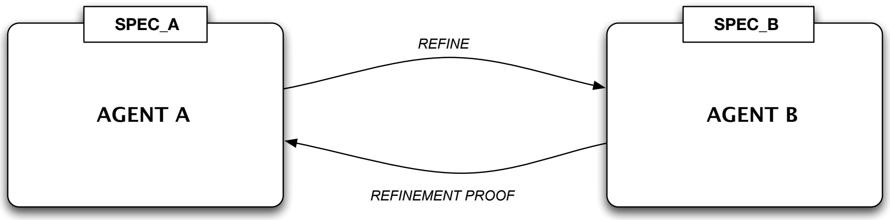
Exploratory programming (or extreme/agile techniques in general) takes a more *ad hoc* route towards implementation. However, with many agile development techniques we can expect to identify parts of the specification with the design as it evolves. For example, we would envisage that, at some point most (if not *all*) of the *safety* formulae in the specification would be satisfied by the (partial) implementation, while few of the *liveness* formulae in the specification would be satisfied before the later stages of development[1].

While the above techniques, notably refinement and synthesis, guarantee that the executable code produced will match the formal specification of the agent, it is also vital to be able to verify that arbitrary code (for example, produced by a third party) matches the specification. Thus, if we are provided with executable code, for example in **`Java`**, or in an agent-oriented language such as **`AgentSpeak`** [38, 8], and also with a logical property to compare the code against, then we can utilise *formal verification* techniques in order to assess whether or not the code satisfies the required property. The behavioural requirements we have of complex agents can be specified using formulae from an appropriate formal logic. As above, the formal logic used can incorporate a wide range of concepts matching the view of the system being described, for example *time* for dynamic/evolving systems, *probability* for systems incorporating uncertainty, *goals* for autonomous systems, etc. This gives great flexibility in the descriptive language that can be used. Given such a specification, we can check this against models/views of the system under consideration in a number of ways. The most popular is that of *model checking* [12]. In such approach, the specification is checked against all possible executions of the system; if there is a finite number of such executions, then this check can often be carried out automatically. Indeed, the verification, via model checking, of both hardware systems (such as chip designs) and software systems (such as Microsoft device drivers) has been very successful in industry as well as academia [2, 4].

In recent years, increasingly sophisticated techniques for the verification of multi-agent systems have been developed [6, 7, 5, 16]. The agents involved are rational and are represented in one of a number of a high-level agent languages describing their beliefs, intentions, etc [15]. In particular, verification has been applied to BDI languages, such as **`GOAL`**, **`AgentSpeak`** and **`Jason`** [38, 8].

3.3 Refinement

The traditional mechanism for utilizing formal specifications in the development of correct systems is to use *refinement*. Here, given a formal agent specification, we devise a second specification typically describing similar behaviours to the original but with a more deterministic and detailed approach.



The basic idea here is that any agent that conforms to the new specification ('SPEC_B') should also conform to the original specification ('SPEC_A'). In traditional formal methods there are many approaches to refinement, but the key aspect is that we must establish (either implicitly or explicitly through a *refinement proof*) that any acceptable behaviour of SPEC_B is also an acceptable behaviour of SPEC_A. Specifically, we must establish

$$\vdash_{\mathcal{L}} \text{SPEC\_B} \Rightarrow \text{SPEC\_A}\,.$$

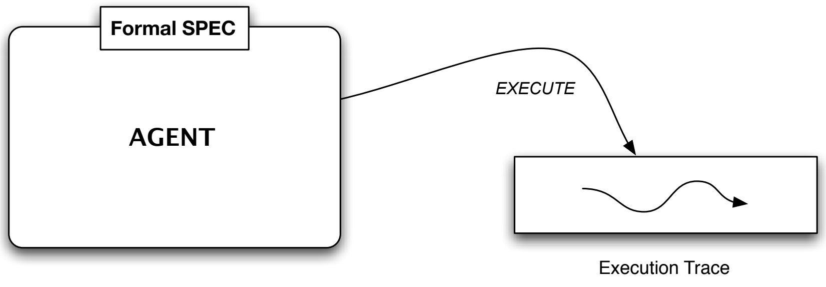
The obvious advantage of this is that if a property holds of SPEC_A, e.g. 'SPEC_A $\Rightarrow \varphi$' then the above implies that 'SPEC_B $\Rightarrow \varphi$'.

Again, since the specifications are potentially given using complex combinations of logics, then such proofs can be non-trivial.

[1]Safety formulae typically characterize "bad things that should never happen" (in temporal logic, $\square\neg bad$, where '$\square$' is the "always in the future" operator) while liveness formulae typically characterize "good things that should eventually happen" (in temporal logic, $\lozenge good$, where '$\lozenge$' is the "sometime in the future" operator) [36].

3.4 Direct Execution

There is one further way that an agent's formal specification can be used in implementation. Rather than, as in synthesis, attempting to build a program that follows the specification in all possible scenarios, we can treat the specification as a program itself and *directly execute* the logical specification. One can think of the distinction between synthesis and direct execution as being similar to that between compilation and interpretation within programming language implementations.



Logically, the execution mechanism can be characterised by a function '`execute`' such that

$$(\exists \mathcal{M}.\ \mathcal{M} = \texttt{execute}(\mathsf{SPEC})) \ \Rightarrow\ \mathcal{M} \models_{\mathcal{L}} \mathsf{SPEC}$$

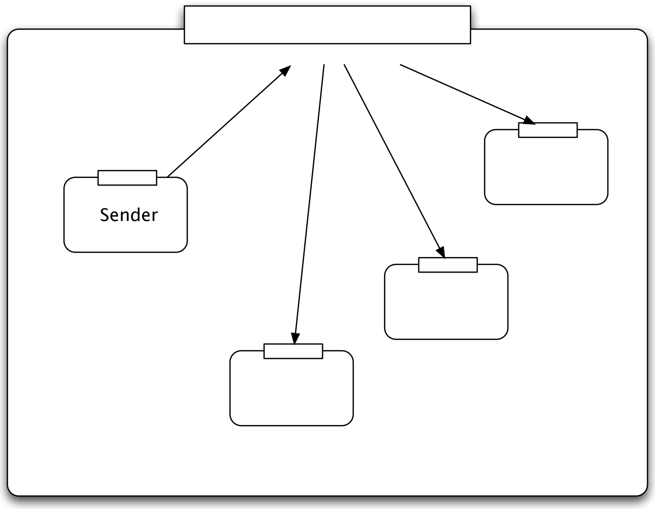
Direct execution of logical agent specification has been explored over a number of years, leading to the stage where logical specifications of goals, beliefs and temporal aspects can be executed [21]. The advantage of direct execution is that it is *much* quicker than synthesis and can often be applied where automatic synthesis is not even possible; the disadvantage is that, while synthesized code is guaranteed to work in any situation, executable specifications may reach unsatisfiable states (although much work has been done on developing heuristics to try to avoid this).

4 Multi-Agent Systems

Now that we have single agents, can specify them, manipulate them, verify them and implement them, we consider multiple such agents interacting together.

4.1 The Computational Model

We take a very simple, and logically coherent, model of multi-agent computation, based on [23, 22, 17]. Agents are independent and interact with each other purely by message-passing. Indeed, we essentially use *broadcast* message passing. Thus, agents broadcast messages within their environment. In this formalism, such an environment is itself described by an agent and we refer to this as a *context*. Broadcast is in fact implemented by sending a message to the context (this 'environmental' agent) which in turn forwards it to the other agents in that context:



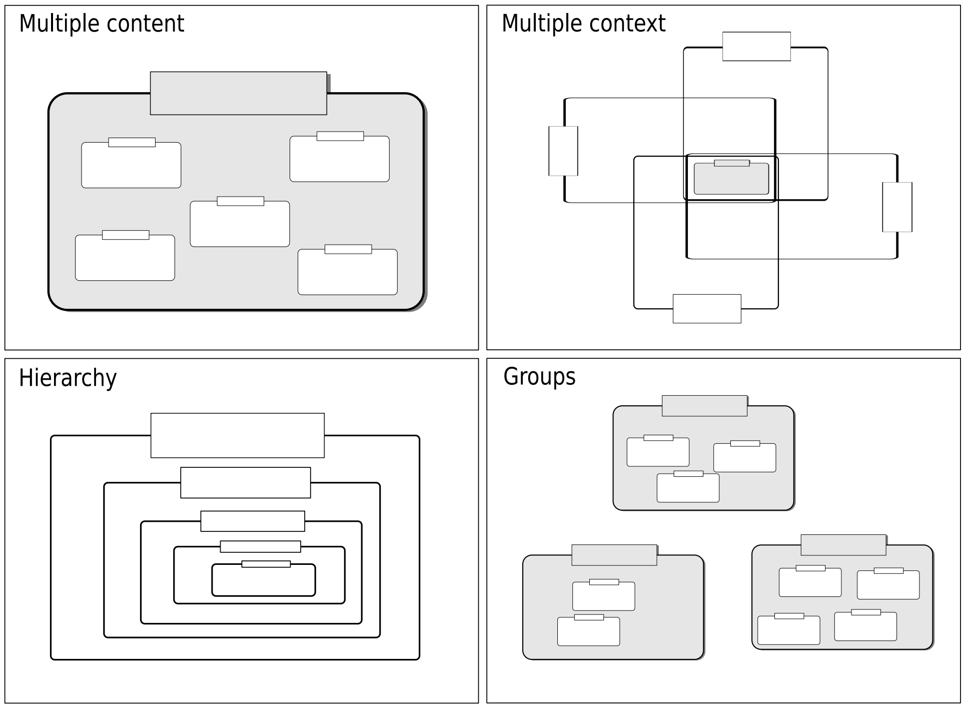
From the above we can see two other important aspects of our multi-agent model. Firstly, an agent can contain other agents; secondly, contexts themselves are also treated as agents. In fact there is no distinction between agents and *any* agent can contain others. Correspondingly, one agent can be within *several* other agents. Thus, our agent description is no longer just a logical specification of the agent's behaviour, SPEC, but also comprises descriptions of the agent's *content* and *context* [23]:

$$\begin{array}{lll} Agent & ::= & \mathsf{SPEC} \\ & & Content : \mathcal{P}(Agent) \\ & & Context : \mathcal{P}(Agent) \end{array}$$

where '$\mathcal{P}$' is the *powerset* operator.

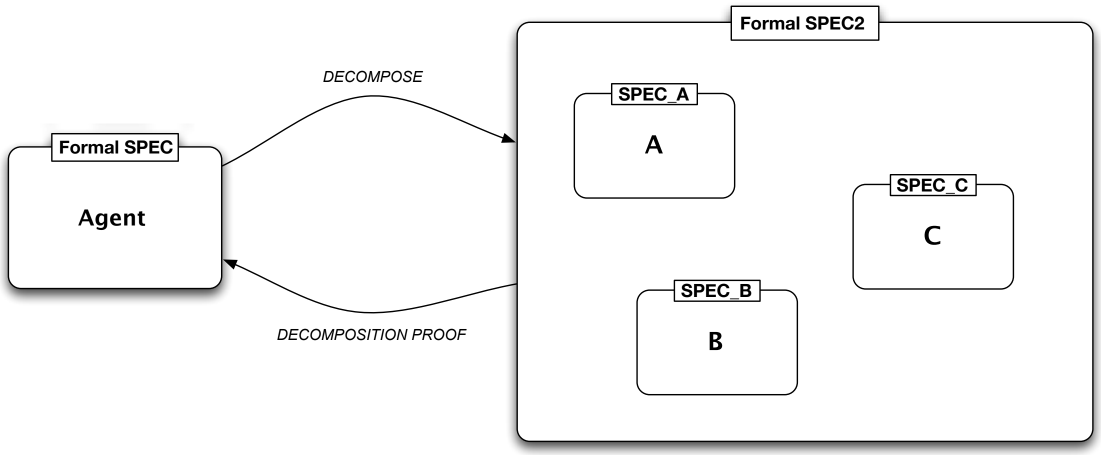
When an agent broadcasts a message, the message by default goes to all agents within the same context. If an agent is within several contexts, then the agent can select which contexts to broadcast throughout. As can be seen from the diagram above, the containing agent only affects the behaviours of agents within it through message-passing.

As we can see, the key notions are not only that each agent has a formal specification, but that each agent also has a (possibly empty) set of other agents within it, and a (possibly empty) set of agents it is within. We will see that these abstractions of *behaviour*, *content* and *context* are vital to our modelling of complex multi-agent organisations. In particular, the notion of context is very useful for representing many different multi-agent abstractions, such as locations, roles, teams, etc. With this simple content/context approach we are able to represent a variety of agent architectures, some simple examples being characterized as follows.



4.2 Decomposition

Once we move to a multi-agent scenario, we can introduce another use for agent specifications. As opposed to the form of refinement described above, where one agent is refined into another agent, we can now refine one agent into a multi-agent system. We call this *decomposition*:



Just as we required a refinement proof to establish the correctness of refinement, so we also need a *decomposition proof* to establish the correctness of decomposition. However, the proof corresponding to decomposition is typically different to that required for ‘normal’ refinement. Recall that we still wish to prove

$$\text{SPEC2} \Rightarrow \text{SPEC}$$

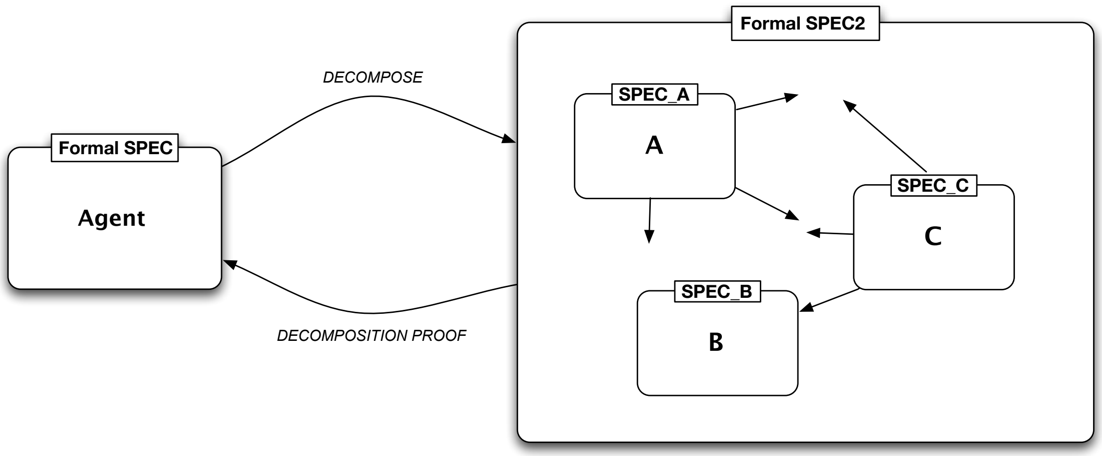
So, ideally, we would like to prove something like

$$(\text{SPEC\_A} \wedge \text{SPEC\_B} \wedge \text{SPEC\_C}) \Rightarrow \text{SPEC}$$

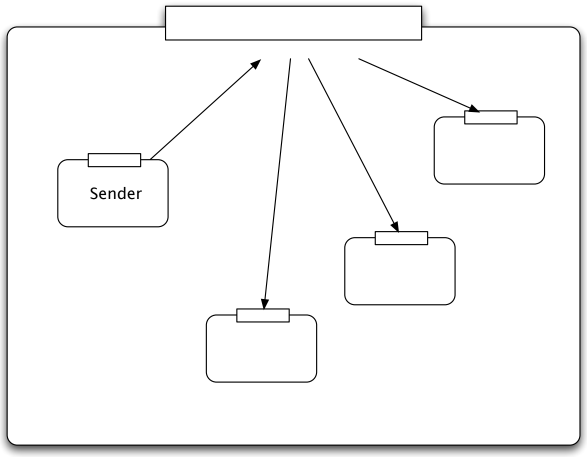
However, the enclosing agent may well impose some additional behaviour on those agents within it. A simple example of this is communication:

$$(\text{SPEC\_A} \wedge \text{SPEC\_B} \wedge \text{SPEC\_C} \wedge \Box(\text{send} \Rightarrow \Diamond\text{receive})) \Rightarrow \text{SPEC}$$

Here, "$\Box(\mathsf{send} \Rightarrow \Diamond\mathsf{receive})$" is an example of an environmental constraint/formula provided by the context agent. Essentially this guarantees that any message sent (i.e., broadcast) from an agent within this context will eventually be received by every other agent in the context[2].



While, in the above diagram, communication appears to be directly between the agents within the context, we see from the decomposition formula that communication actually occurs via the formal specification of the context; recall:



The specification of communication, as given within the context's specification, defines the communication within that context. As we will see below, such communications formulae are just simple examples of additional formulae imposed by the context [19].

Aside. In the above we have assumed a simple form of concurrency where agents execute concurrently, but synchronously. Though it is clearly more complex, we can describe the *asynchronous* composition of agents, following the use of more sophisticated, dense, temporal logics as in [18].

4.2.1 Modular Design

If we are using decomposition as part of a modular design process, possibly following a refinement methodology, then we may already have a proof that

$$\mathsf{SPEC2} \Rightarrow \mathsf{SPEC}$$

[2]Note that we have simplified this formula slightly for illustrative purposes.

and instead of proving

$$(\mathsf{SPEC\_A} \wedge \mathsf{SPEC\_B} \wedge \mathsf{SPEC\_C}) \Rightarrow \mathsf{SPEC}$$

we will prove that

$$(\mathsf{SPEC\_A} \wedge \mathsf{SPEC\_B} \wedge \mathsf{SPEC\_C}) \Rightarrow \mathsf{SPEC2}$$

Once we incorporate constraints from the context, we get

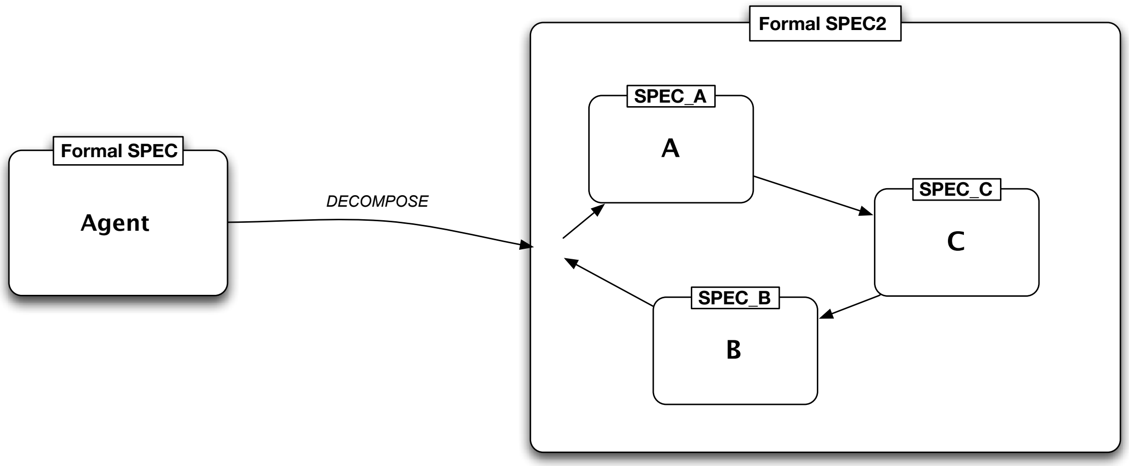
$$(\mathsf{SPEC\_A} \wedge \mathsf{SPEC\_B} \wedge \mathsf{SPEC\_C} \wedge \Upsilon) \Rightarrow \mathsf{SPEC2}$$

where Υ is the behaviour/formula provided by the context. In Section 5, we will see how Υ can be represented in a more explicit way.

4.2.2 Transformation-Based Decomposition

In some cases decomposition proofs can be simplified, or even avoided all together. Rather than carrying out arbitrary decompositions (or, refinements in general), we might provide a set of decomposition 'patterns' to the developer. Each such pattern not only gives an abstract view of a way in which an agent can be decomposed into several others, but also comes with a proof template. Once the abstract pattern is instantiated with specific agents, so the template is also instantiated providing either an automatic decomposition proof, or at least a simpler and clearer route to the full decomposition proof.

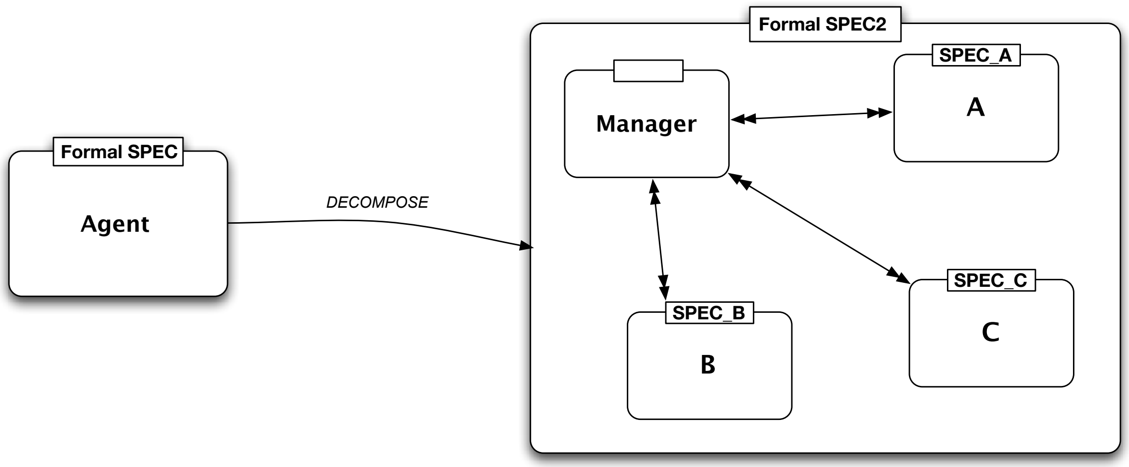
A simple example is



Here, the pattern is that SPEC is of the form "$\psi \Rightarrow \Diamond\varphi$" and the single agent is decomposed into three agents which chain together to eventually generate the required result. Typically:

$$\begin{aligned} \mathsf{SPEC\_A} &= \psi \Rightarrow \Diamond m_1 \\ \mathsf{SPEC\_B} &= m_1 \Rightarrow \Diamond m_2 \\ \mathsf{SPEC\_C} &= m_2 \Rightarrow \Diamond \varphi \end{aligned}$$

Another obvious pattern is to have a specific *manager* agent sub-contracting sub-tasks:



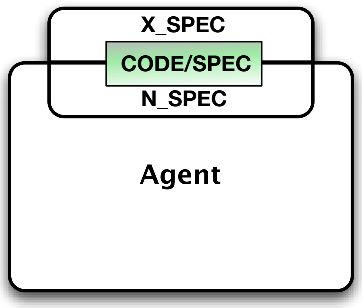
And so on.

5 Behaviours

We now wish to represent different, and more expressive, behaviours often derived from contexts and even multiple overlapping contexts.

5.1 Visual Notation

We now also change our visual representation of an agent to:



This will allow us to distinguish the specification of the agent from (a) the (current) externally visible behaviour of the agent, and (b) the (current) constraints imposed on agents within its context. An agent has a specification as normal, and (initially) extracted from this specification is a distinct specification of behaviour relevant/visible to its context and another relevant/visible to its content. But crucially, the X_SPEC is likely to be comprised of properties (such as “abilities”) that a context can expect to be exhibited by the agent, whilst the N_SPEC is likely to be comprised of less concrete behaviour relating to communication, goals, norms.

- CODE/SPEC is the description of the agent’s behaviour, which could be a formal specification, or could be concrete code;
- X_SPEC is the specification of the current guaranteed behaviour of this agent visible within the agent’s context; and

- N_SPEC is the specification of the current constraints of the agent, as viewed by the agent's content.

Initially, at least, both N_SPEC and X_SPEC are just partial views representing aspects of the overall agent specification, SPEC. Importantly, the behaviour of the agent, as visible by its context may depend upon the agent's content and context. For instance an agent may offer an ability to its context based on the abilities of its content agents. Similarly an agent may gain an ability due to one context it is a member of and so be able to offer that ability to other contexts.

Thus we require the properties below, capturing the satisfiability of the specifications and the fact that any advertised behaviour of an agent must be implied by the agent, its contents or its context:

$$\exists \mathcal{M}.\ \mathcal{M} \models_{\mathcal{L}} \bigwedge_{i \in Content} \mathsf{X\_SPEC}_i \bigwedge_{i \in Context} \mathsf{N\_SPEC}_i \wedge \mathsf{SPEC} \tag{1}$$

and

$$\vdash_{\mathcal{L}} \bigwedge_{i \in Content} \mathsf{X\_SPEC}_i \bigwedge_{i \in Context} \mathsf{N\_SPEC}_i \wedge \mathsf{SPEC} \Rightarrow \mathsf{X\_SPEC} \tag{2}$$

Where $\mathsf{X\_SPEC}_i$ is the X_SPEC of agent i and similarly $\mathsf{N\_SPEC}_i$ is the N_SPEC of agent i. We require, for the present at least, that the formulae contained in X_SPEC and N_SPEC are Horn clauses to limit the possibilities for contents and contexts contradicting each other.

Meanwhile an agent's internal constraints can also be derived from its specification but may also be affected by its context (for instance an agent may wish to pass on a goal from its context to its contents) so we also require:

$$\vdash_{\mathcal{L}} \bigwedge_{i \in Context} \mathsf{N\_SPEC}_i \wedge \mathsf{SPEC} \Rightarrow \mathsf{N\_SPEC} \tag{3}$$

If we have CODE rather than a formal specification, then we must simply check

$$\tau(\mathsf{CODE}) \models_{\mathcal{L}} \Big(\big(\bigwedge_{i \in Content} \mathsf{X\_SPEC}_i \bigwedge_{i \in Context} \mathsf{N\_SPEC}_i \big) \Rightarrow \mathsf{X\_SPEC} \Big)$$

$$\tau(\mathsf{CODE}) \models_{\mathcal{L}} \bigwedge_{i \in Context} \mathsf{N\_SPEC}_i \Rightarrow \mathsf{N\_SPEC}$$

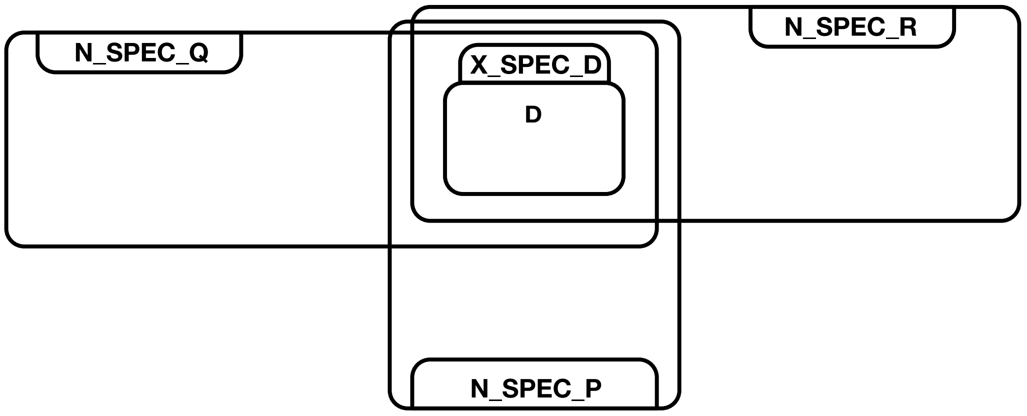
A consequence of our system is that an agent's overall visible behaviour may depend upon its context, for instance it may receive a goal from its context. We write $\mathcal{XB}$ for the overall behaviour of an agent given its current content and context and define this as:

$$\mathcal{XB}_a \equiv \bigwedge_{x \in Context(a)} \mathsf{N\_SPEC}_x \wedge \mathsf{X\_SPEC}_a$$

$\mathcal{XB}$ describes the behaviour of an agent that we can deduce knowing only its 'advertised' behaviour and the contexts in which it is placed. This allows us to reason about agent behaviour in differing contexts without, necessarily, being aware of the agent's internal specification or composition.

So, for instance the overall visible behaviour of the agent D in the diagram below is

$$\mathcal{XB}_{\mathsf{D}} \equiv \mathsf{N\_SPEC\_P} \wedge \mathsf{N\_SPEC\_Q} \wedge \mathsf{N\_SPEC\_R} \wedge \mathsf{X\_SPEC\_D}$$

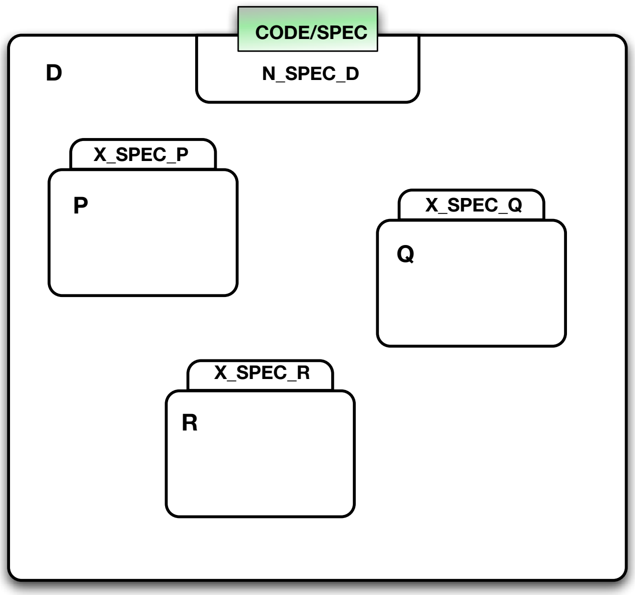


Similarly we define the internal behaviour, $\mathcal{IB}$, of an agent as

$$\mathcal{IB}_a \equiv \bigwedge_{x \in Content(a)} \mathsf{X\_SPEC}_x \wedge \mathsf{N\_SPEC}_a$$

So, for instance the overall internal behaviour of the agent D in the diagram below is

$$\mathcal{IB}_{\mathsf{D}} \equiv \mathsf{X\_SPEC\_P} \wedge \mathsf{X\_SPEC\_Q} \wedge \mathsf{X\_SPEC\_R} \wedge \mathsf{N\_SPEC\_D}$$



5.2 Autonomy

In the above, we have assumed that any agent's behaviour is directly affected by the N_SPEC formula of any context in which it resides. However, 'real' agents are autonomous and need not permit external agents to have direct control over them. Inevitably the need to reason about groups of agents must reduce their autonomy. Our formalisation requires that we can prove the following.

$$\mathsf{SPEC} \vdash_{\mathcal{L}} \forall x \in Context.\ \mathsf{N\_SPEC}_x \tag{4}$$

i.e., that the agent is specified in such a way that, whatever context it finds itself in, it can guarantee that the formulae in that context will govern its behaviour.

However it is not necessary for agents to give up all autonomy. For instance we typically envisage that N_SPEC may contain a joint goal for an agent's content. The individual agent's are required to adopt the goal but there are no constraints imposed on how actively they pursue the goal, nor how they go about achieving the goal. Similarly formulae in N_SPEC may be expressed using the sometime operator $\Diamond\phi$ leaving agents autonomy about when to fulfill the formula or may be expressed probabilistically and so forth if the logic allows. Lastly agents retain the autonomy to enter and leave contexts and thus agree autonomously to accept or reject the constraints of any given N_SPEC. (See Section 7 for further work in this area.)

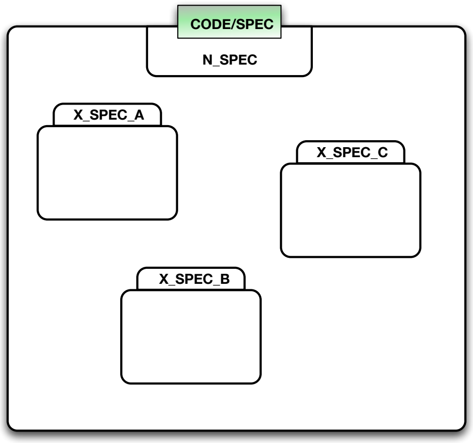
5.3 Simple Examples

5.3.1 Example 1

We next consider a number of simple examples. First, we provide N_SPEC with a simple communications formula which guarantees messages broadcast within the system will eventually reach their recipients;

$$\forall_{x \in context}.\ \forall_{ag \in content_x}.\ send(\psi) \Rightarrow \Diamond receive_{ag}(\psi)$$

We then show how the temporal behaviour of the multi-agent system is affected by this. Later, we turn to examples concerning formulae in N_SPEC that capture *joint goals*, *joint beliefs*, etc.



Here:

- X_SPEC_A is **start** $\Rightarrow talk$
- X_SPEC_B is $\square(hear \Rightarrow \lozenge snore)$
- X_SPEC_C is $\square(annoy \Rightarrow \bigcirc shout)$
- N_SPEC is

$$\square \left[\begin{array}{rcl} talk & \Rightarrow & \lozenge hear \\ snore & \Rightarrow & \lozenge annoy \end{array} \right]$$

The combined internal behaviour, $\mathcal{IB}$, of the multi-agent system is then

$$\begin{array}{rcll} \textbf{start} & \Rightarrow & talk & \wedge \\ \square(hear & \Rightarrow & \lozenge snore) & \wedge \\ \square(annoy & \Rightarrow & \bigcirc shout) & \wedge \\ \square(talk & \Rightarrow & \lozenge hear) & \wedge \\ \square(snore & \Rightarrow & \lozenge annoy) & \end{array}$$

which implies

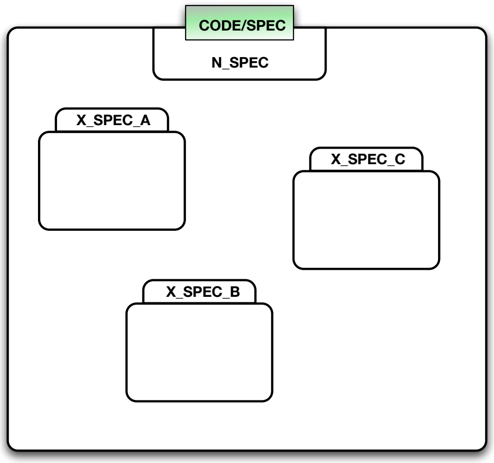
$$\textbf{start} \Rightarrow \lozenge shout\,.$$

Thus, effectively, the agents interact together, via the context, in order to eventually make $shout$ true. Note, however, that if the same agents described by X_SPEC_A, X_SPEC_B, and X_SPEC_C, move to a different context *without* the communications formulae provided by N_SPEC, then we can not prove $\lozenge shout$.

Furthermore, in the diagram above, $\lozenge shout$, could validly appear in the X_SPEC of the whole multi-agent system as the system can achieve this.

5.3.2 Example 2

Next we consider an example where the specifications are given in terms of agent *beliefs* as well as time.



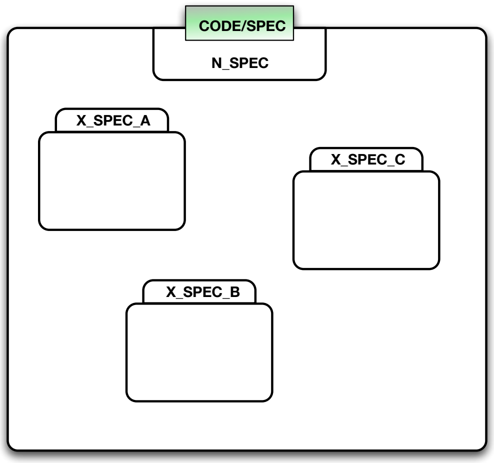
Here we ignore communication aspects (assuming that communication will occur as required, in particular that agents communicate their beliefs to each other) and specify the agents by:

- X_SPEC_A is **start** $\Rightarrow B\psi$
- X_SPEC_B is $\square((B\varphi \wedge B\psi) \Rightarrow B\xi)$
- X_SPEC_C is ....
- N_SPEC is persistent shared belief $\square B\varphi$

Again the agents combine together and, utilising the shared belief provided by the context, eventually produce $B\xi$.

5.3.3 Example 3

If we use a logic characterising *goals* we can also use contexts to represent shared goals, as follows.



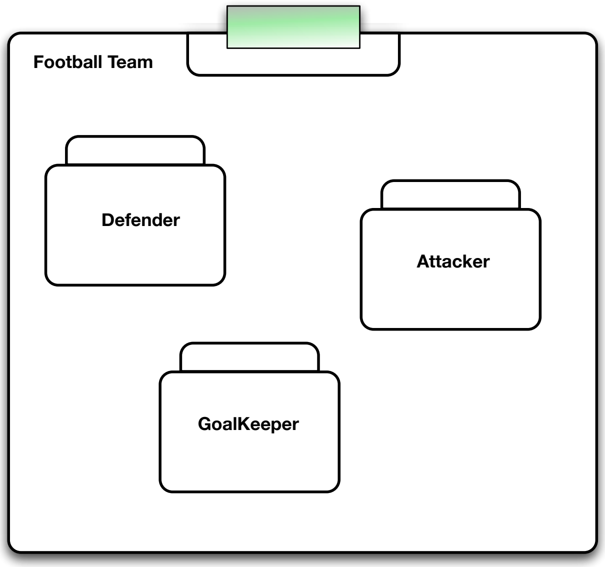
Again (ignore communications aspects):

- X_SPEC_A is $G\psi \Rightarrow G\varphi$.
- X_SPEC_B is....
- X_SPEC_C is....
- N_SPEC is shared goal $G\psi$.

Here the combined behaviour is $G\varphi$ as the shared goal combines with SPEC_A to generate this. Note that we might also carry out a form of *goal decomposition* here. If SPEC_B is $\square(\alpha \Rightarrow \psi)$ then we can decompose the goal $G\varphi$ deriving the new (sub) goal $G\alpha$.

5.3.4 Example 4

We will now consider two examples which look at the use of internal and external specifications in the modular decomposition of agents. Consider designing a football team agent. We might start with a simple design in which the Team contains Defender, Attacker and GoalKeeper agents which each offer certain abilities to the team.



For example (where 'A' is the "ability" operator)

$$\text{X\_SPEC}_{\text{Defender}} = A\,\mathit{defend}$$

$$\text{X\_SPEC}_{\text{Attacker}} = A\,\mathit{attack}$$

$$\text{X\_SPEC}_{\text{GoalKeeper}} = A\,\mathit{gkeep}$$

while $\text{SPEC}_{\mathit{Football\ Team}}$ should contain the formula

$$[A\,\mathit{defend} \land A\,\mathit{attack} \land A\,\mathit{gkeep}] \Rightarrow A\,\mathit{play\_football}$$

and

$$\text{X\_SPEC}_{Football\ Team} = A\, play\_football$$

If the content agents behave as required then we can use the Football Team agent as expected without worrying about the internal make-up of the team. Specifically, $\text{X\_SPEC}_{\text{Football Team}}$ characterises the externally visible behaviours. Moreover, assuming the content agent have the right abilities then

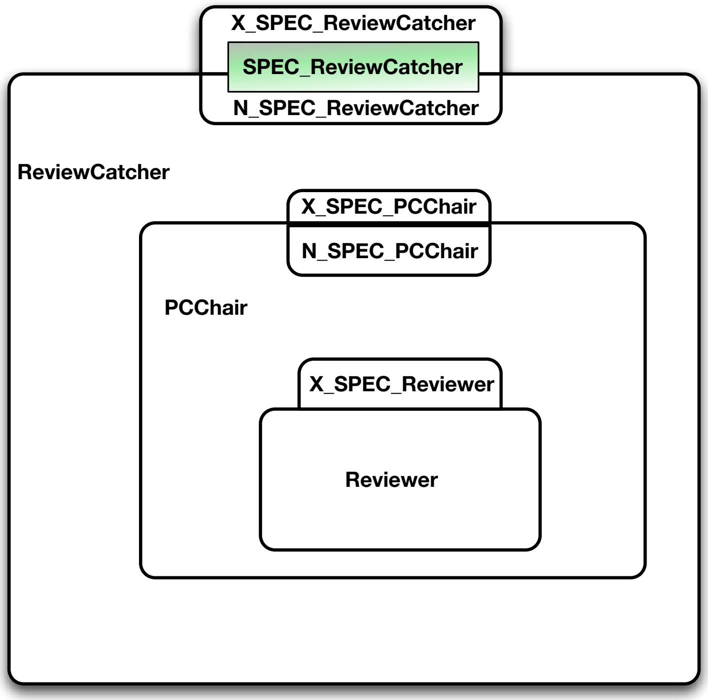
$$\text{X\_SPEC}_{\text{Defender}} \wedge \text{X\_SPEC}_{\text{Attacker}} \wedge \text{X\_SPEC}_{\text{GoalKeeper}} \wedge \text{SPEC}_{\text{Football Team}} \Rightarrow \text{X\_SPEC}_{\text{Football Team}}$$

So (2) holds as required.

5.3.5 Example 5

We conclude this example to show how the formalisation allows reuse of modular components and organisations in the maintenance phase of a project.

Let us consider an example from [32] of a conference system. In this design a PC_ Chair acts in the role of a Review_Catcher who assigns papers to reviewers and ensures there are no conflicts of interest. The PC_ Chair agent contains a number of PC Member agents who review the papers assigned to them



The X_SPEC and N_SPECs of these agents are defined as follows

$$\begin{aligned}\text{X\_SPEC\_ReviewCatcher} \quad = \quad & receive(paper,\ CONF) \Rightarrow \Diamond review(paper) \wedge \\ & B\, reviewed(a,\ paper) \Rightarrow B\neg author(a,\ paper)\end{aligned}$$

where $CONF$ is the name of the conference. So the ReviewCatcher role guarantees that it will eventually review all papers and that it will not believe a paper has been reviewed by its own author.

$$\begin{aligned}\text{N\_SPEC\_ReviewCatcher} \quad = \quad & B\, reviewed(a,\ paper) \Rightarrow B\, \neg author(a,\ paper) \wedge \\ & G\,(reviewed(a,\ paper) \Rightarrow \neg author(a,\ paper))\end{aligned}$$

This contains the additional constraint, useful internally, that it never generates a goal for an author to review their own paper.

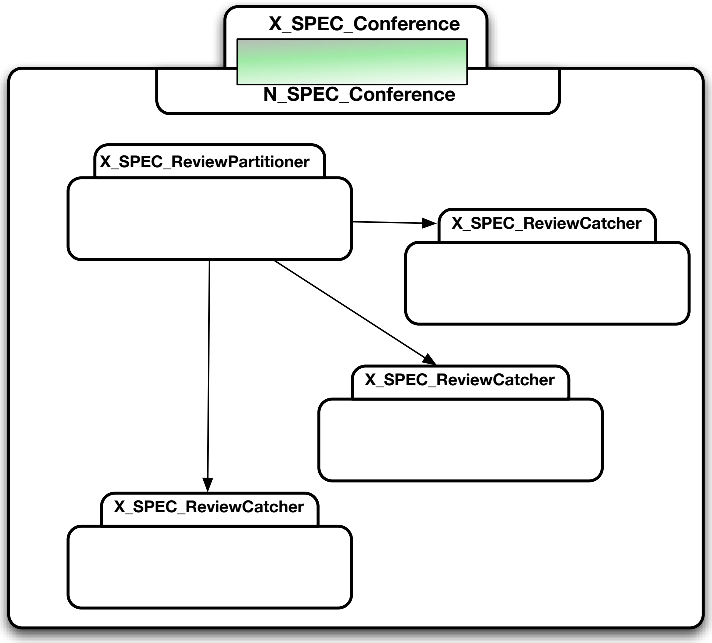
$$\begin{aligned}\text{X\_SPEC\_PCChair} \quad &= \quad received(paper,\ conf) \Rightarrow \Diamond review(paper) \\ \text{N\_SPEC\_PCChair} \quad &= \quad received(paper,\ conf) \Rightarrow \Diamond \exists a \in Content.\ G\, reviewed(a,\ paper)\end{aligned}$$

So the PCChair advertises that it will eventually review any paper and internally it assigns papers to reviewers by generating an appropriate goal.

$$\begin{aligned}&\text{X\_SPEC\_Reviewer} \;=\\ &\quad (name(NAME)\ \wedge\ G\,reviewed(NAME,\ paper)) \Rightarrow \Diamond review(paper) \wedge\\ &\quad G\,(reviewed(NAME, paper) \Rightarrow \Diamond B\ reviewed(NAME,\ paper))\end{aligned}$$

The reviewer advertises its name and guarantees to review papers it receives.

As the conference grows the designer decides to add a new layer to the organisation. (S)he introduces the role of ReviewPartitioner who assigns papers to areas. These papers are then assigned to a number of agents who are enacting the ReviewCatcher role.



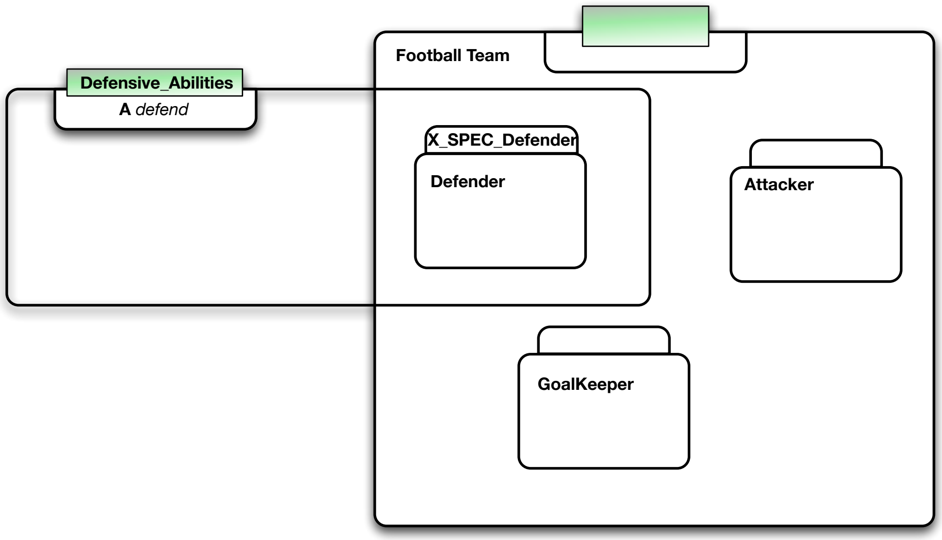
The X_SPEC and internal composition of the ReviewCatchers are as above, except that instead of being initialised with the name of the conference, $CONF$ above, they are initialised with the name of an area. The agent system is reused in its entirety. At this point, therefore, the designer needs only worry about the additional structure required and designs new X_SPECs and N_SPECs as follows:

$$\begin{aligned}\text{X\_SPEC\_CONF} &= && receive(paper,\ CONF) \Rightarrow \Diamond review(paper) \wedge\\ & && Breviewed(a,\ paper) \Rightarrow B\neg author(a,\ paper)\\ \text{N\_SPEC\_CONF} &= && receive(paper,\ CONF) \Rightarrow Bhave(paper) \wedge\\ & && send(paper,\ AREA) \Rightarrow \Diamond receive(paper,\ AREA)\\ \text{X\_SPEC\_ReviewPartitioner} &= && B\,have(paper) \Rightarrow \Diamond send(paper,\ AREA)\end{aligned}$$

Together with the existing agents and roles this continues to guarantee the overall desired behaviour of the system.

5.4 Contexts Adding Behaviours

We now return to the use of contexts. While, in the Football Team example, we might implement the Defender, Attacker, etc, agents directly, we can alternatively utilise other contexts:



Here, the agent D might not be capable of defending by itself but, when in the Defensive_Abilities context, it can utilise that context's beliefs, abilities, etc. Thus, $\mathsf{SPEC}_{\mathsf{Defender}}$ might *not* explicitly contain "$A\,defend$". However, $\mathsf{N\_SPEC}_{\mathsf{Defensive\_Abilities}}$ contains $A\,defend$. So, (4) guarantees that Defender satisfies

$$\mathsf{SPEC}_{\mathsf{Defender}} \vdash_{\mathcal{L}} \forall x \in Context.\ \mathsf{N\_SPEC}_x$$

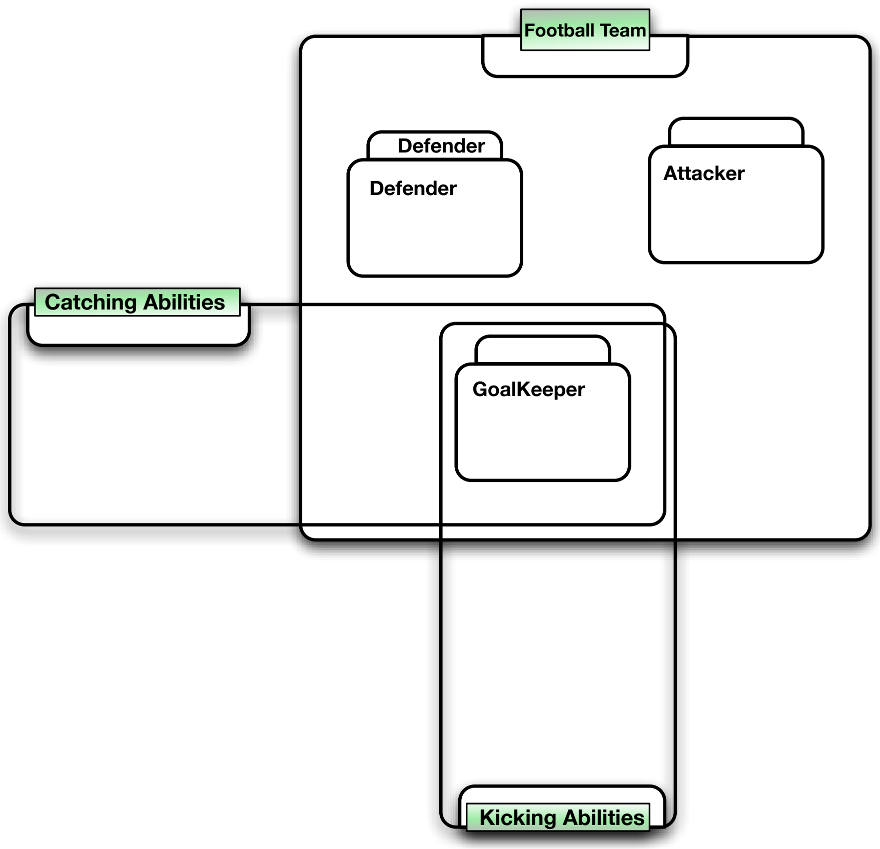
So

$$\mathsf{SPEC}_{\mathsf{Defender}} \wedge \ \mathsf{Defensive\ Abilities} \in Context \rightarrow A\ defend$$

This allows Defender to validly advertise $A\,defend$ in its X_SPEC which can then be utilised by the football team.

Note that, once in the Defensive_Abilities context, the Defender always has the defend ability but, unless that is advertised in its X_SPEC the Football team can not reason about nor make use, explicitly, of that ability.

Agents can have many contexts:



As long as the agent, defined within overlapping contexts, behaves as expected then other agents it interacts with do not care how its behaviour is constructed.

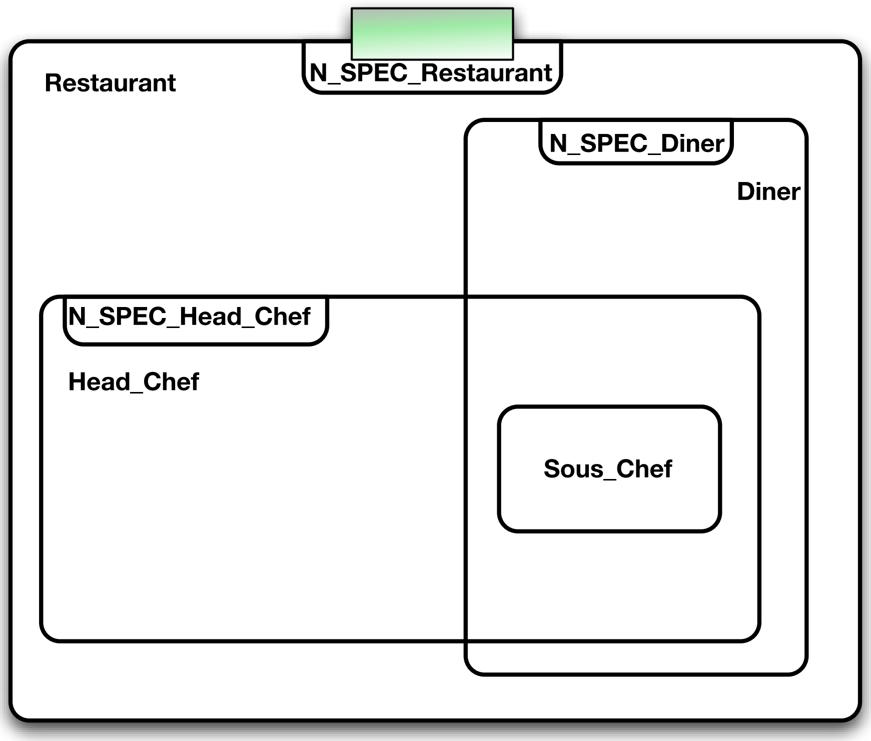
In the previous example, contexts were represented as conferring abilities. It is more likely that plans offered by the context would enable the content agent to offer an ability. However, contexts can represent many different aspects, e.g:

- locations, providing sensory input;
- themes/styles, providing preferences;
- organizations, providing goals and beliefs;
- institutions, providing beliefs and norms; or
- locations, proving neighbourhood information.

5.5 Contexts Constraining Behaviours

Our final example in this section demonstrates how a context agent can be a normative influence on its contents, by applying constraints on their behaviour. The intuition of agents adapting their behaviour according to the changing context of their actions is central not only to our proposed architecture but also to the multi-agent paradigm in general.

The example concerns a restaurant, its head chef, a sous chef and a customer. Clearly all agents involved would like to be assured that this particular restaurant satisfied (eventually) all customer orders. However, restaurant customers can be fussy about their food, and head chefs are equally particular about their menus. The relationships between agents in our restaurant system are depicted in the following diagram



The initial design provides the restaurant with a content specification that ensures all orders are eventually served, i.e.

$$\mathsf{N\_SPEC}_{\mathsf{Restaurant}} = \quad G\ order \Rightarrow \Diamond serve(meal).$$

This, is adopted by the Head_Chef — and subsequently the sous chef — who fulfills their role by providing a menu of meals to which its content agent, the Sous_Chef, must adhere. (We assume that there is some appropriate, but different, mechanism that makes the menu available to the Diner.)

$$\mathsf{N\_SPEC}_{\mathsf{Head\_Chef}} = \quad G\ order \Rightarrow \bigcirc\big(serve(pizza) \vee serve(risotto) \vee serve(steak)\big)$$

A Diner expresses their constraints in their N_SPEC, in this case the vegetarian constraint;

$$\mathsf{N\_SPEC}_{\mathsf{Diner}} = \square\big(contains(x, meat) \Rightarrow \neg serve(x)\big)$$

We are now able to verify that, not only will the restaurant satisfy all orders, but it will do so whilst adhering to the requirements of each diner.

$$\forall diner \in context_{\mathsf{Restaurant}}.\ \mathcal{XB}_{\mathsf{Restaurant}} \Rightarrow \mathsf{N\_SPEC}_{\mathsf{Diner}}$$

We envisage that preferences (see Section 7) would be communicated in a similar way.

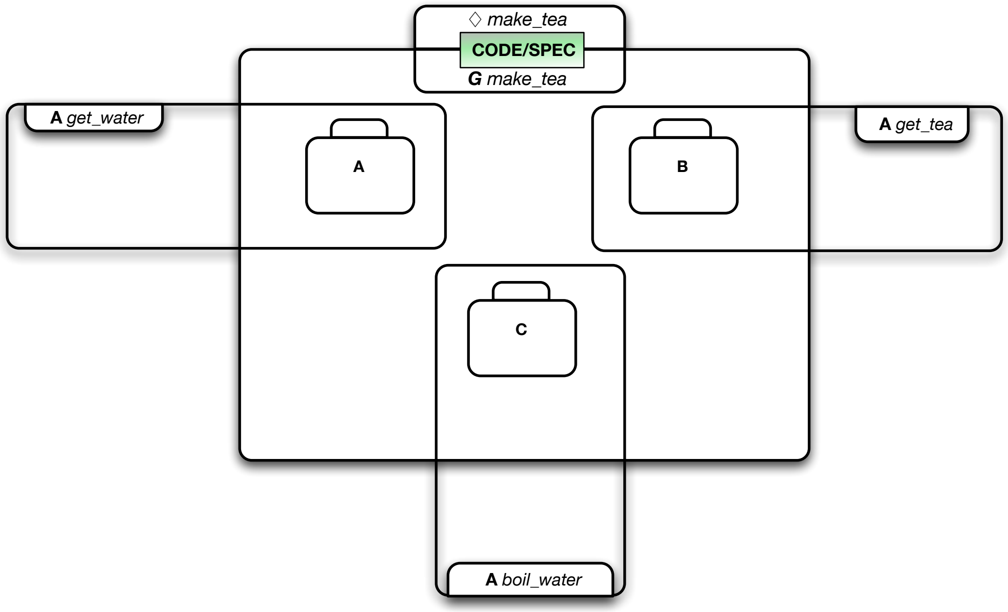
6 Dynamic Specifications for Dynamic Agents

So far we have discussed our notation as a static, design-time system. The proof obligations we have identified are used to ensure that complex, modular multi-agent systems with many contexts and contents still fulfil their advertised specifications. However the existence of the identified proof obligations also allows dynamic reasoning about agents in a changing environment. We consider this primarily in the context of an executable system where agents freely move between contexts and modify their X_SPEC and N_SPEC accordingly. However similar reasoning could be used within a design tool where the system could automatically update parts of the specification as the design changed.

So far we have considered the formal specification of multiple agents embedded within agent contexts. However a vital aspect concerns the behaviour that becomes available to agents as they *enter* and *leave* contexts.

6.1 Leaving Contexts

Consider the following example of a tea making agent which is composed of three sub-agents who gain vital skills from their over-lapping contexts.



We assume that the agent's SPEC includes the formula

$$[A\ get\_water \land\ A\ boil\_water \land\ A\ get\_tea] \Rightarrow\ \Diamond make\_tea$$

We also assume that the internal agents are exporting the abilities from their role contexts to their X_SPECs.

What happens if agent 'A' leaves this agent's content? At the point when 'A' leaves, and if there is no other agent that can fulfill A's *get_water* role, then the tea making agent can no longer prove

$$\text{X\_SPEC}_{\text{B}} \land\ \text{X\_SPEC}_{\text{C}} \land\ \text{SPEC} \Rightarrow\ \Diamond make\_tea$$

and so can no longer advertise X_SPEC $= \ \Diamond make\_tea$. However, we *can* prove

$$\text{X\_SPEC}_{\text{B}} \land\ \text{X\_SPEC}_{\text{C}} \land\ \text{SPEC} \Rightarrow\ (A\ get\_water \Rightarrow \Diamond make\_tea)$$

It could therefore dynamically modify it's X_SPEC to $A\ get\_water \Rightarrow \Diamond make\_tea$

Thus, this agent system, if provided with the ability to *get_water*, will then be able to *make_tea*.

Without a goal to make tea, $G\ make\_tea$, the agent specification remains as above. However, with the goal $G\ make\_tea$ the agent must *actively* look to find ways to achieve the goal, e.g. by first tackling a sub-goal of

$$G\ (A\ get\_water).$$

This could involve communicating with other agents to recruit a new content agent.

It seems probable that, given the formulae appearing in X_SPEC and N_SPEC are restricted to Horn clauses, if the contents of SPEC are kept simple and avoid, for instance, appeal to recursion, then only simple procedures could be required to maintain the validity of X_SPEC with respect to changing content. Indeed within the right logical framework it should be possible to maintain the validity of X_SPEC without the need to redo proofs.

These mechanisms would also be useful in agent organisations with roles. Typically an organisation defines its roles and then seeks agents to enact them. Returning to the example of the football team. The Defender, Attacker and Goal Keeper agents might all be represented as roles enacted by identical Player agents such that

$$\begin{aligned}\text{X\_SPEC}_{\text{Player}} = &\\ &G\ defend \Rightarrow A\ defend \land\\ &G\ attack \Rightarrow A\ attack \land\\ &G\ gkeep \Rightarrow A\ gkeep\end{aligned}$$

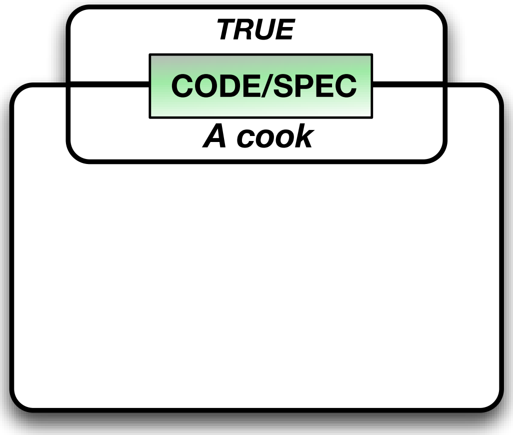
The Defender Role, for instance, would have an N_SPEC

$$\text{N\_SPEC}_{\text{Defender Role}} = G\, defend$$

In a dynamic situation a Player agent, on enacting the Defender Role would modify its X_SPEC to include $A\,defend$.

6.2 Entering Contexts

Consider a context that provides a specific *ability* or *plan*:



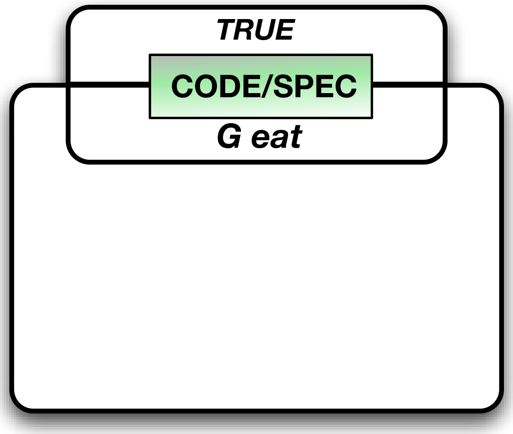
Any agent entering into this context will be able to access the ability to $cook$ within its computations.

Furthermore an agent, a, could, if it so chose, add $A\ cook$ to its X_SPEC. Since

$$\vdash_{\mathcal{L}} \text{N\_SPEC} \Rightarrow A\, cook$$

now holds for a. In practice heuristics will be used to decide exactly which deductions are placed in an agent's external specification as it enters and leaves contexts.

Similarly, a context might provide a particular goal to its members:



Suppose an agent, a, enters this context and reasons dynamically about the specifications. Firstly we require that the modified behaviour of a is consistent. Assuming, for the moment, that this is a's only context, a needs to check that

$$\exists \mathcal{M}.\ \mathcal{M} \models_{\mathcal{L}} G\ eat \wedge \text{X\_SPEC}_a$$

If a model could not be found then a could be prevented from entering the context.

Finally to go back to the tea making example from the previous section. Suppose the "tea making" agent currently advertises $A\,get\_water \Rightarrow \Diamond make\_tea$ in its X_SPEC. An agent enters its context which offers $A\,get\_water$ in its own X_SPEC. This would allow the tea making agent to validly alter its own X_SPEC so that it could once again offer $\Diamond make\_tea$.

7 Concluding Remarks

7.1 Summary

In this paper we have brought together formal specification, automated verification, dynamic agent organisations, and visual modelling in order to provide a simple, but semantically coherent, modular framework for designing and developing multi-agent systems. We believe this to be particularly useful in areas that are difficult for contemporary design methods to handle, namely: multiple overlapping/evolving contexts, dynamically evolving behaviours, and fault tolerance. The approach has a clear semantic basis and, in principle, allows a wide range of tools (such as proof checkers, theorem-provers, execution methods, synthesis procedures, etc.) to be applied. The fact that *everything* (including organisations and teams) is an agent means that these techniques can also be applied directly to complex organisations and teams themselves.

7.2 Future Work

Extensions, Application and Evaluation. Obviously, there are still many areas to be explored. For example, is the approach truly useful as a graphical design/development process? And, if it is, to what extent can proofs/checking be carried out in background? Or is it more appropriate as an approach to visual programming for multi-agent systems? Or are the visual aspects actually not very useful in practice?

We have advocated this view as a flexible design method for multi-agent systems, but clearly need to apply this in larger case studies and to evaluate its effectiveness. Will it turn out to just be another static design method for multi-agent systems? Or will the dynamic aspects really be useful? And, although we have in mind using executable agents specification, how far can we go beyond this? To general BDI languages? Or even general agent languages?

Tool Development. We already have tools for many parts of this framework: deductive verification tools for temporal and modal specifications; direct execution tools for agent specified using time and belief; a graphical representation of the content/context hierarchy; etc. However, all these need to be combined into a coherent and consistent toolset, especially before application to larger case studies.

Preferences. We have seen how contexts not only provides beliefs, goals, etc., to agents within it. In addition, context can provide preferences, in the form of *prefer* constraints. These do not have any direct logical meaning, but are used within the agent's deliberation process in order to select the most preferred option. Thus, if an agent has several options it can take (say a or b) and *prefer(b,a)* is present, then (assuming there are no other constraints) the agent will choose to do b. If an agent occurs within multiple contexts, then it is under the influence of multiple preferences. These aspects can be very useful in complex areas such as dynamic deliberation and dynamic reconfiguration [26, 25], and we aim to incorporate these within the design framework.

Flexible Autonomy. In the above, we have assumed that any agent's behaviour is directly affected by the $\mathsf{N\_SPEC}$ formula of any context in which it resides by ensuring that

$$\mathsf{SPEC} \vdash_{\mathcal{L}} \forall x \in Context.\ \mathsf{N\_SPEC}_x\,.$$

This is quite strong, and requires that the agent "gives away" some of its autonomy to its contexts. We intend to look at different variations of this, for example where the agent has a choice about whether to agree to the context behaviour of not, or is given a variety of strengths of agreement from which to choose.

References

[1] R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proc. 1st International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 1998.

[2] T. Ball and S. K. Rajamani. The SLAM Toolkit. In *Proc. 13th International Conference on Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 260–264. Springer, 2001.

[3] B. Bauer, J. P. Müller, and J. Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, 2001.

[4] S. Berezin, E. M. Clarke, A. Biere, and Y. Zhu. Verification of Out-Of-Order Processor Designs Using Model Checking and a Light-Weight Completion Function. *Formal Methods in System Design*, 20(2):159–186, 2002.

[5] R. H. Bordini, L. A. Dennis, B. Farwer, and M. Fisher. Automated Verification of Multi-Agent Programs. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 69–78, 2008.

[6] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Model Checking Rational Agents. *IEEE Intelligent Systems*, 19(5):46–52, September/October 2004.

[7] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying Multi-Agent Programs by Model Checking. *Journal of Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, March 2006.

[8] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley, 2007.

[9] G. Cabri, L. Leonardi, and M. Puviani. Methodologies for Designing Agent Societies. In *Proc. 2nd International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 529–534. IEEE Computer Society, 2008. `http://dx.doi.org/10.1109/CISIS.2008.65`.

[10] L. Cernuzzi and F. Zambonelli. Experiencing AUML in the GAIA Methodology. In *Proc. 6th International Conference on Enterprise Information Systems (ICEIS)*, pages 283–288, 2004.

[11] L. Cernuzzi and F. Zambonelli. Dealing with Adaptive Multi-agent Organizations in the Gaia Methodology. In *Proc. 6th International Workshop on Agent-Oriented Software Engineering (AOSE)*, volume 3950 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2006.

[12] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.

[13] E. M. Clarke and E. A. Emerson. Using Branching Time Temporal Logic to Synthesise Synchronisation Skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[14] P. R. Cohen and H. J. Levesque. Intention Is Choice with Commitment. *Artificial Intelligence*, 42(2-3):213–261, Mar. 1990.

[15] L. A. Dennis, B. Farwer, R. H. Bordini, M. Fisher, and M. Wooldridge. A Common Semantic Basis for BDI Languages. In *Proc. 7th International Workshop on Programming Multiagent Systems (ProMAS)*, volume 4908 of *LNAI*, pages 124–139. Springer Verlag, 2008.

[16] L. A. Dennis and M. Fisher. Programming Verifiable Heterogeneous Agent Systems. In *Proc. 6th International Workshop on Programming in Multi-Agent Systems (ProMAS)*, volume 5442 of *LNCS*, pages 40–55. Springer Verlag, 2008.

[17] L. A. Dennis, A. Hepple, and M. Fisher. Language Constructs for Multi-Agent Programming. In *Proc. 8th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA)*, volume 5056 of *Lecture Notes in Artificial Intelligence*, pages 137–156. Springer, 2008.

[18] M. Fisher. A Temporal Semantics for Concurrent METATEM. *Journal of Symbolic Computation*, 22(5/6), November/December 1996.

[19] M. Fisher. Representing abstract agent architectures. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V — Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg, 1999.

[20] M. Fisher. Temporal Development Methods for Agent-Based Systems. *Journal of Autonomous Agents and Multi-Agent Systems*, 10(1):41–66, Jan. 2005.

[21] M. Fisher. METATEM: The Story so Far. In *Proc. Third International Workshop on Programming Multiagent Systems (ProMAS)*, volume 3862 of *Lecture Notes in Artificial Intelligence*, pages 3–22. Springer Verlag, 2006.

[22] M. Fisher, C. Ghidini, and B. Hirsch. Organising Computation through Dynamic Grouping. In *Objects, Agents and Features*, volume 2975 of *Lecture Notes in Computer Science*, pages 117–136. Springer-Verlag, 2004.

[23] M. Fisher and T. Kakoudakis. Flexible agent grouping in executable temporal logic. In Gergatsoulis and Rondogiannis, editors, *Intensional Programming II*. World Scientific Publishing Co., Mar. 2000.

[24] J. C. García-Ojeda, S. A. DeLoach, Robby, W. H. Oyenan, and J. Valenzuela. O-MaSE: A Customizable Approach to Developing Multiagent Development Processes. In *Proc. 8th International Workshop on Agent-Oriented Software Engineering (AOSE)*, volume 4951 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2008.

[25] A. Hepple, L. A. Dennis, and M. Fisher. A Common Basis for Agent Organisations in BDI Languages. In *Proc. International Workshop on LAnguages, methodologies and Development tools for multi-agent systemS (LADS)*, volume 5118 of *Lecture Notes in Artificial Intelligence*, pages 171–88. Springer-Verlag, 2008.

[26] B. Hirsch, M. Fisher, C. Ghidini, and P. Busetta. Organising Software in Active Environments. In *Computational Logic in Multi-Agent Systems (CLIMA-V)*, volume 3487 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[27] J. Howse and S. Schuman. Precise Visual Modeling: A Case-Study. *Software and System Modeling*, 4(3):310–325, 2005.

[28] C. B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[29] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, Englewood Cliffs, NJ, 1986.

[30] O. Kupferman and M. Y. Vardi. μ-Calculus Synthesis. In *Proc. 25th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 1893 of *Lecture Notes in Computer Science*, pages 497–507. Springer, 2000.

[31] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley Professional, 2003.

[32] M. Luck and L. Padgham, editors. *Agent-Oriented Software Engineering VIII*, volume 4951 of *Lecture Notes in Artificial Intelligence*, Heidelberg, Germany, 2008. Springer-Verlag.

[33] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.

[34] Z. Manna and R. J. Waldinger. Toward Automatic Program Synthesis. *ACM Communications*, 14(3):151–165, 1971.

[35] J. Misra and K. M. Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[36] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[37] L. Padgham and M. Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons, 2004.

[38] A. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Agents Breaking Away — Proc. Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW)*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.

[39] A. S. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proc. First International Conference on Multi-Agent Systems (ICMAS)*, pages 312–319, San Francisco, USA, 1995.

[40] A. S. Rao and M. P. Georgeff. Modeling Agents within a BDI-Architecture. In R. Fikes and E. Sandewall, editors, *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, Cambridge, Massachusetts, Apr. 1991. Morgan Kaufmann.

[41] S. Schewe and B. Finkbeiner. Distributed Synthesis for Alternating-Time Logics. In *Proc. 5th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 4762 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag, 2007.

[42] S. Schneider. *The B-Method: An Introduction*. Palgrave, October 2001.

[43] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 1995.

[44] B. van Linder, W. van der Hoek, and J.-J. C. Meyer. Formalising Abilities and Opportunities of Agents. *Fundamentae Informaticae*, 34(1-2):53–101, 1998.

[45] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Organisational Rules as an Abstraction for the Analysis and Design of Multi-Agent Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):303–328, 2001.

[46] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing Multiagent Systems: The Gaia Methodology. *ACM Transactions on Software Engineering Methodology*, 12(3):317–370, 2003.