An agent-oriented programming approach for C++-based simulation models

Benjamin Herd

TR-16-02
September 2016

Department of Informatics, King’s College London

Abstract

Agent-based simulation (ABS) has shown great success for the study of complex adaptive systems which are hard or even impossible to analyse using conventional analytical techniques. For reasons of performance and flexibility, non-trivial ABS models are often implemented in general purpose programming languages like Java or C++. As a consequence of the generality of those languages, simulation-based agents are traditionally rule-based and thus rather ‘myopic’ in nature which critically limits their level of behavioural sophistication. In the more general area of multiagent systems, agent-oriented programming (AOP) has emerged as a powerful paradigm for the implementation of intelligent, practically reasoning agents. However, current AOP languages tend to prioritise expressivity over performance which critically limits their application in a time-critical simulation context.

The goal of this work is to address this problem and to bridge the gap between the world of C++-based simulation and the world of practically reasoning agents. We present a first version of an efficient and customisable C++-based BDI framework, equipped with a declarative AOP interface that allows for the implementation of BDI-based agents on a high level of abstraction without compromising on performance. The balance between efficiency and convenience of development is achieved by utilising advanced template metaprogramming techniques.

1 Introduction

A central problem that severely limits the complexity of many contemporary agent-based models is the low level of behavioural sophistication. Simulation-based agents are traditionally rather ‘myopic’ in nature, i.e. their behaviour is largely pre-determined and based on simple behavioural rules. Although such a representation of the simulated individuals may be sufficient for many modelling problems, it is certainly not for all of them. Particularly in the area of social simulation, where human beings are to be simulated, a higher level of behavioural sophistication may be required to truthfully represent the real-world entities and to reach a sufficient level of validity. Furthermore, in cases where only the environment that the agents operate in but not the actual rules that they follow are known, a higher level of autonomy with respect to the agents’ decision making can help to develop more realistic and interesting models.

The BDI architecture is a particularly popular example of a cognitive architecture. Its theoretical origins date back to the philosophical work of Michael Bratman, in particular his work on human practical reasoning [6]. BDI provides a convenient balance between cognitively realistic yet extraordinarily complex theories of human reasoning, and computationally efficient yet cognitively overly simple representations of human behaviour such as those mostly employed in contemporary agent-based simulation (ABS). The BDI architecture has proved very useful for the implementation of flexible, autonomous agents and serves as the conceptual basis of numerous dedicated agent-oriented programming (AOP) languages and frameworks. Unfortunately, in the ABS community, the BDI architecture is largely unknown. There are several reasons for that. First, due to its interdisciplinary nature, ABS is mainly conducted by non-computer scientists who are often not familiar with the work in the AI

\[1\] BDICA lore, desires, intentions.
community. Second, the implementation of a BDI interpreter with its complex internal logic is highly non-trivial, even for an expert. And, third, despite its light weight in comparison to other architectures, BDI is still fairly complex and computationally demanding. It was developed for the implementation of real embodied agents such as robots or unmanned vehicles rather than for the purpose of simulation, in which case the requirements on execution time and scalability are typically much higher. As a consequence, existing BDI implementations have prioritised usability and interoperability with existing software environments over performance, e.g. by building upon Java technology.

Most existing BDI interpreters are based on Java; support for C++ is still largely missing. We do, however, believe that C++ offers a number of features that make it interesting for simulation developers. First, C++ is arguably the most performant high-level programming language which renders it particularly useful for the development of simulation models that should run efficiently on a single machine. Second, with its support for template metaprogramming [1, 2], C++ provides a powerful built-in mechanism for the implementation of embedded domain-specific languages (EDSLs). In that way, it is possible to hide complex computations behind a convenient and easy-to-use interface yet without compromising on the runtime performance of the application. As described further below, we make extensive use of this technique for the development of a declarative interface, akin to that provided by AOP languages such as Jason or 2APL [8]. And, third, with several new standards being published within just a couple of years, C++ is currently experiencing an exciting evolution. With powerful new features that significantly improve the programming experience (e.g. advanced type inference and the auto keyword or lambda expressions), we believe it has a realistic chance of catching up with Java as a major general purpose ABS modelling language.

The purpose of this work is to reconcile the flexibility and performance C++ with the convenience and high level of abstraction of declarative agent-oriented programming. This is to be achieved by integrating features of both worlds into a common BDI framework that is tailored to the specific requirements of multiagent-based simulations. We put a particular focus on efficiency, flexibility, and usability. A solution is efficient if it provides a high level of performance and if it can be integrated into an existing simulation model without much effort; it is flexible if it does not force the simulation programmer to be constrained to the BDI architecture in its full functional complexity exclusively, but allows him to vary the level of an agent’s cognitive sophistication depending on his current needs; and, it is usable if it provides a convenient interface to the programmer which hides most of the functional complexity of the underlying BDI interpretation cycle. In this paper, we present BDI4MABS, a C++-based BDI framework for the purpose of multiagent-based simulation. It consists of two central parts:

1. An efficient and customisable BDI interpreter.
2. An embedded domain specific language that facilitates the formulation of the agent logic through a declarative interface.

The paper is structured as follows. We start with an overview of AOP in Section 2, followed by a motivating example in Section 3; the design rationale and implementation details of the BDI framework are given in Section 4, followed by an evaluation in Section 5; the paper concludes with a summary in Section 6.

2 Agent-oriented programming

One of the most influential agent-oriented programming languages is AgentSpeak(L) [14]. An AgentSpeak(L) program consists of beliefs, desires (or goals), triggering events and actions. Programs in AgentSpeak(L) are written using a subset of first-order logic. The syntax is similar to conventional logic programs and augmented by a number of symbols in order to express achievement, testing, sequencing and implication. An agent in AgentSpeak(L) is specified by its knowledge base (beliefs), its goals and a list of available plans. A plan consists of a head and a body. A triggering event specifies when the plan is invoked (e.g. when a belief or desire has been added or removed). A plan that matches a given event is called relevant for that event. Plans are context-sensitive, that is, specific conditions need to be fulfilled in order for a plan to be executable. Conditions are specified within the context section of the plan’s head. A plan that both matches a given event and satisfies the context conditions is called applicable. Finally, the body specifies the single steps that need to be executed when the plan is being pursued. These steps can be either actions or the addition or removal of further facts and desires to an agent’s knowledge base. Beliefs describe an agent’s current state, its knowledge about itself, about other agents, and about the world in general. The facts that represent the goal states that the agent wants to bring about are its desires. In order to achieve a desired goal, an agent adopts a plan from the library and commits itself to the pursuit of the
respective goal. The plans that have been adopted by the agent thus represent its intentions. Beliefs, desires and intentions are not represented explicitly through modular formulae; instead, they are ascribed to agents implicitly at design time. The deliberation cycle of AgentSpeak(L) programs follows the BDI deliberation scheme. In general, the following steps can be distinguished:

1. Perceive the environment and updating the belief base accordingly.
2. Select an event to handle.
3. Determine all relevant plans.
4. Determine all applicable plans and update intentions accordingly.
5. Select an intention for execution.
6. Execute one step of the chosen intention and modify the intention stack and the belief base accordingly.

Several practical interpreters for AgentSpeak(L) or variants thereof have been developed [12, 3], the most notable of which is Jason [5]. Jason represents a hybrid approach that combines the convenience of a declarative, logic-based language with the flexibility of a general-purpose programming language (Java). The agent logic is formulated in a declarative way, more complex procedural logic can be formulated in Java and referred to from within the declarative code.

Although AgentSpeak(L) and its concrete implementations have mostly been used for the implementation of embodied reasoning agents, it has recently also been suggested as a tool for agent-based simulation development [4]. However, due to its complex internal dynamics, performance remains a critical bottleneck and its usefulness for the implementation of large-scale and performance-critical simulations remains limited. Despite various calls for higher cognitive sophistication in agent-based modelling and simulation, the BDI architecture is still mostly unknown in the simulation community. The major technical reason is complexity. The complex dynamics in the BDI reasoning cycle render the implementation of simulation models with hundreds, thousands, or even millions of agents largely infeasible.

One attempt to bridge the gap between the world of cognitive agents and the agent-based modelling community is BDI for NetLogo [16]. The authors use lists and reporters, two of NetLogo’s central features to mimic some of the dynamics of the BDI architecture. For example, beliefs are represented as lists containing the type and the content. Intentions are also represented as lists with two entries, intention name and intention part, the former representing function calls and the latter relating to reporters returning true once the goal has been achieved. Although NetLogo is a great language for quick model development, it is fairly slow and thus not suitable for large-scale simulation models. Furthermore, models have to be written using NetLogo’s own domain-specific language which may be a limiting factor, especially if the model requires complex calculations. Another attempt to bring closer together the two areas of AOP and ABS has been made by Padgham et al. [13] who integrated JACK [17] into Repast, a popular Java-based agent-based modelling framework [7]. According to the authors, the main reason for using JACK instead of developing an entirely new BDI interpreter on top of Repast is to leverage the maturity of JACK. As a consequence, the main focus of the work is on the synchronisation between the two platforms rather than on the development of a new approach. Due to the reliance upon two separate comprehensive frameworks, the approach is relatively heavyweight and thus not ideal for large-scale simulations running on a single machine.

3 Motivating example: a simple party scenario

We introduce use as our running example a simple simulation model originally presented by Padgham et al. [13]. The behaviour of an individual agent is shown graphically in Figure 1. Here, the agent aims to purchase a present for a party. In order to be able to do so, he first needs to get money, either from the ATM or from a friend. Getting money from the ATM requires two further actions: walking to the ATM and withdrawing cash; getting money from a friend also requires two actions: walking to the friend’s place and collecting the money. Once the money is available, the agent needs to walk to the shop and purchase the present. In order to make things more interesting, we further assume that there is a wide range of products from which the agent has to choose. Each product has a certain price and the agent has an internal threshold according to which he rules out products that are too expensive. From the remaining set of products, the agent selects one randomly and inversely proportionally to the price of the product — the cheaper the product, the more likely it will bought by the agent.
From a conceptual point of view, it would be convenient to represent each product option as a separate plan, as indicated in Figure 1. However, as the set of products grows larger, the manual implementation of plans becomes infeasible. BDI4MABS facilitates the formulation of plans over large sets through *parametrised goals, plans*, and *actions*. The choice is then made as part of the plan selection process by means of *utility evaluation*. In that way, the selection process in the case of large domains of values becomes conveniently hidden behind a declarative BDI-like interface.

### 4 An agent-oriented programming framework for C++

We start our exploration of BDI4MABS with an overview of requirements and desired features in Section 4.1, followed by a description of the declarative interface in Section 4.2; implementation details of the BDI interpreter are given in Section 4.3.

#### 4.1 Requirements

Following a thorough requirements analysis, a number of key features for the framework have been identified [9]; the central ones are described below.

**Modelling beliefs:** It must be possible to model an agent’s beliefs in a simple yet flexible way. In particular, it should be easy to integrate key-value attributes of different types commonly used in ABS into the deliberation process such that they can be easily accessed from within an action.

**Modelling goals:** It must be possible to model an agent’s individual goals in a declarative way. We distinguish between *parametrised* and *unparametrised*, conventional goals. Parametrised goals contain an additional type parameter together with a textual name that can be used to access the value of a particular sequence of values (e.g. a list of products), for each of which an instance of the particular goal is created internally. Unless labelled *parallel* (see further below), a goal needs to be satisfied immediately in order for the superordinate plan to continue successfully.

**Modelling conventional plans:** It must be possible to model an agent’s available plans in a declarative way. Following the notation of Jason, this includes (i) the *triggering condition* (i.e. the goal), (ii) the *context*, and (iii) the *plan body* (containing subgoals as well as actions).
Modelling parametrised plans: In order to facilitate the formulation of large option sets, i.e. large sets of plans which only differ in one particular parameter, it must be possible to specify parametrised plans, i.e. plans that are parametrised with a container of possible options and that are instantiated automatically.

Modelling actions: It must be possible to model the actions that an agent is capable of performing. In order to provide a high level of flexibility, actions may contain arbitrary procedural logic; on the other hand, they should be referable from within plans in a declarative way. Furthermore, in the presence of parametrised plans, actions themselves also need to be parametrisable.

The framework should further allow the developer to specify the way in which decisions during the deliberation process are to be made. In particular, we want to support the following three types of selection:

Event selection: The user should be able to specify an event selection mechanism that selects from the event store the next event for execution.

Plan selection: The user should be able to specify a plan selection mechanism. This involves both the selection of relevant plans and the selection of applicable plans. Plan selection should also allow for the integration of utility functions, an important aspect of many ABS models.

Intention selection: The user should be able to specify an intention selection mechanism that selects from the intention store the next intention for execution.

Finally, the environment that agents are situated in and that they react to plays an important role in the development of an ABS. The environment itself is not part of the deliberation process and its dynamics are thus beyond the control of the BDI architecture. However, since agents need to perceive the environment as part of the BDI deliberation process, an appropriate interface needs to be provided by the framework.

Modelling the environment: It should be possible to integrate the environment in which the agents are situated into the deliberation process. The logic of the environment is to be provided by the developer; in order to provide the highest level of flexibility, the framework should not impose any constraints upon the nature of the environment and allow for its formulation in a high-level language (C++). Nevertheless, it needs to provide an interface through which it can be perceived by the agents as part of their deliberation process.

4.2 The declarative interface

Due to the complexity of the underlying reasoning mechanism, AOP can be tricky. In order to facilitate the formulation of AOP applications, it is thus crucial to relieve the programmer from having to deal with the intricacies of the underlying interpreter. Both Jason and 2APL are nice examples of how a subdivision between a convenient and easy-to-use interface and the complex dynamics of the reasoning process can be realised by extending the actual reasoner with a domain-specific language for the formulation of the agent logic. To this end, BDI4MABS has been equipped with a embedded domain-specific language (EDSL) that aims to mimic some of the declarative aspects of AOP languages like Jason or 2APL yet without compromising on the efficiency of the underlying C++ implementation. In particular, the user should be enabled to formulate beliefs, actions, events, and plans in an easy and intuitive way. Starting with the environment, the framework elements that constitute the EDSL are described below. For illustration, we contrast the C++ entities with their respective counterparts in Jason.

Environment: The environment can be realised as an arbitrary C++ object. In that way, the programmer is given maximum flexibility with respect to the logic. The only requirement is that the environment class has to provide a function perceive that the framework uses during the perception stage. The structure of the class is shown below.

```cpp
struct Environment {
    template <class AS>
    void perceive(AS& as) {
        // the environment logic goes here
    }
};
```

The perceive function expects the agent’s state as an argument. In order to allow for the highest level of flexibility, the type of the state can be provided by the user.
**Beliefs:** It is common in agent-based modelling to support a wide range of different beliefs. In Jason, beliefs can be specified irrespective of their type which is handled automatically through type inference, as shown below.

\[
\begin{align*}
\text{priceThreshold}(10). \\
\text{price(“prodA”,} 42). \\
\end{align*}
\]

In order to support an equivalent level of flexibility, BDI4MABS supports *heterogeneous belief bases* that allow for storing values of different type. The user defines the belief base by passing it the data types that it is supposed to handle. For our party scenario example, we require three data types: an integer representing an agent’s price threshold, a set containing all available products, and a dictionary containing products and their prices.

```java
using BB = BeliefBase<int, set<int>, map<int, float>>;
```

Beliefs represented by simple data types (int, string, etc.) can then be added through a common interface (realised through overloading the `addBelief` function), irrespective of their type.

```java
int thr = 10; set<int> products = {1,2,3,4};
bb.addBelief("priceThr", thr);
bb.addBelief("products", products);
```

Similarly, beliefs can be queried in a unified and convenient way.

```java
int thr; set<int> products;
bb.getBelief("priceThr", &thr);
bb.getBelief("products", &products);
```

In order to add/remove items to/from a container type, we can obtain a pointer to the container through which the element can be added/removed. This is particularly efficient since the data structure does not have to be copied in memory.

```java
set<int>* products;
bb.getBeliefP("products", products);
products->insert(42);
```

**Goals:** In Jason, goals are represented as literals and prepended as required in the specific context (e.g. to describe goal addition or removal). In BDI4MABS, since goals are to be processed at compile time, they need to be represented as types. Goals do not exhibit any functionality on their own, so there is no need for a concrete goal to inherit from a base class. Goals can thus be represented as simple types. The agent’s goal library can then be described as a simple typelist.

```java
struct GetPresent {};
struct GetMoney {};
using Goals = list<GetPresent, GetMoney>; // static list of goals
```

**Actions:** In Jason, actions are implemented in plain Java. To that end, the Jason-internal environment classes that need to be subclassed in order to implement a custom environment provide a function `executeAction` which can be used to implement the action logic. An example implementation for a custom action `walkToShop` is shown below.

```java
public boolean executeAction(String agName, Structure action) {
    if (action.getFunctor().equalsIgnoreCase("walkToShop")) {
        try {
            // custom action logic goes here
        } catch (Exception e) {
            System.err.println("Error: "+ e.getMessage());
        }
    }
}
```

The so-defined action can then be referred to from within plans by its name. In the following Jason plan, action `walkToShop` is used as the second item of the plan body.

```java
+!getPresent : true <-
!getMoney;
walkToShop;
purchasePresent.
```
In BDI4MABS, actions represent executable entities and are thus implemented as simple function objects (called ‘functors’ in C++). The framework distinguishes between basic and parametrised actions. Instances of basic actions can be created by simply inheriting from class BAction and overwriting the function call operator. Using the example from above, this can be achieved as follows.

```cpp
struct WalkToShop : public BAction {
    template <class AS>
    bool operator()(AS& as) const { /* custom logic */ }
};
```

A parametrised action (PAction) is particularly powerful if the same function is to be executed for a range of values. In our party scenario, a 'Buy Product' action is required for each individual product. Since products are represented as integers, 'Buy Product' can be realised as a parametrised action with int as parameter type.

```cpp
struct BuyProduct : public PAction<int> {
    template <class Env, class AS>
    bool operator()(Env&, AS& as) const { /* custom logic */ }
};
```

**Events:** In Jason, events that may appear in the body of a plan are described by simply prepending the goal with the appropriate event specifier, as shown below.

```plaintext
!g // achievement goal
!!g // parallel achievement goal
?g // test goal
```

As with goals described above, events are represented as types in BDI4MABS. For performance reasons, test goals are not explicitly supported; if the belief base is to be queried, this is best done within a basic action. The framework supports both goal addition and goal removal. In addition, a distinction between sequential and parallel (i.e. background) execution is made. Events are described as simple types, as shown below. They can then be used within plan bodies, as described in the next paragraph.

```plaintext
Seq<BGoal<GetMoney>>; // sequential addition of goal 'GetMoney'
Par<BGoal<GetMoney>>; // parallel addition of goal 'GetMoney'
Rem<GetMoney>; // removal of goal 'GetMoney'
```

In addition to basic goals, BDI4MABS also supports the notion of parametrised goals. A parametrised goal can be seen as a template that gets instantiated for a range of given values. For example, in the party scenario, the goal 'Buy Present' is parametrised by the set “products” of integers representing the products in the agent’s belief base, as shown below.

```plaintext
Seq<PGoal<BuyProduct, "products", int>>, Rem<GetMoney>;
```

The addition of a parametrised plan causes each relevant plan for the goal (in our example, this is just the plan 'Buy Product') to be instantiated exactly once for each element in the set “products” and to be passed the respective product as a parameter.

**Plans:** In Jason, plans are described in a purely declarative way by specifying the triggering event, the context, and the plan body. The plan body itself is simply a sequence of formulae, i.e. actions or subgoals.

```plaintext
+!GetMoney // triggering event
:true <- // context
  walkToATM; // basic action
  withdrawCash; // basic action
  !shop. // subgoal
```

In BDI4MABS, given the definition of goals, actions, and events, plans can be formulated in a way that is as declarative as in Jason.

```plaintext
using PGetMoneyFromBank = Plan<
  GetMoneyFromATM, // plan name
  GetMoney, // triggering event (goal addition)
```

Note that, at the current stage, only goal addition is supported as a triggering event for a plan.
The plan body unfolds as:

```c++
list<
  // plan body
  WalkToATM, // basic action
  WithdrawCash // basic action
  Seq<BGoal<Shop>>, // sequential subgoal
>>;
```

The agent’s plan library can then be described as a simple typelist.

```c++
using PlanLibrary = list<
  PBuyPresent,
  PGetMoneyFromATM,
  PGetMoneyFromFriend,
  PBuyProduct>;
```

In Jason, the applicability of a plan is described by its context. In BDI4MABS, the user can specify a dedicated applicability function. For example, in our party scenario, an agent may only consider a product if its price is less than or equal to a given threshold (i.e. the maximum price that the agent is willing to pay). This can be formulated as an applicability functor as follows (since the plan that the functor refers to is parametrised, the functor itself is also parametrised).

```c++
template <>
struct applicable<PBuyProduct, int> {
  template <class AS>
  bool operator()(AS& as, int const& i) {
    unsigned int thr;
    as.getBB().getBelief(g_sPriceThr, thr); // price threshold
    as.getBB().getBeliefP(g_sPrice, price);
    return price[i] <= thr;
  }
};
```

Finally, BDI4MABS supports different plan selection mechanisms. In our example, the agent selects products based on their price: the cheaper the product, the more likely it will be bought. Utility evaluations of that kind are very common in ABS. In order to facilitate this process, BDI4MABS provides a utility evaluation selector which selects plans based on their relative utility. The utility calculation itself can be provided by the user (similar to the applicability functor described above) as shown below.

```c++
template <>
struct utility<PBuyProduct, int> {
  template <class AS>
  float operator()(AS& as, int const& param) {
    unsigned int thr;
    as.getBB().getBelief(g_sPriceThr, thr);
    as.getBB().getBeliefP(g_sPrice, price);
    float p = (*price)[param];
    return (thr-p);
  }
};
```

Agent state: Finally, the overall agent state needs to be described. In Jason, this happens implicitly through the definitions in the `asl` file, i.e. the file that contains the agent logic. In BDI4MABS, the agent state needs to be described explicitly. As with the definition of plans, BDI4MABS allows for a purely declarative description, as shown below.

```c++
using AS = AgentState<
  Beliefs, // belief base
  PlanLibrary, // plan library
  Goals, // goals that can be handled
  list<Add<BGoal<GetPresent>>>, // list of initial goals
  RTApplicability, // applicability checking mechanism
  UtilityEvalSelector> // plan selection mechanism
```

The EDSL is realised through a combination of static polymorphism and template metaprogramming, as illustrated in further detail in the next section.
4.3 Implementation of the BDI interpreter

In order to manipulate the static types described above as part of the BDI reasoning cycle, BDI4MABS makes extensive use of both static polymorphism using template parameters and template metaprogramming (TMP) [2, 1]. In TMP, the C++ template system is used to formulate metaprograms, i.e. programs that operate upon types at compile time. TMP can be employed to perform structural transformations of the source code at compile time; this is particularly useful for the development of embedded domain-specific languages, as well as for the manipulation of types at runtime. For space limitation, we cannot provide full details here. We thus restrict our focus to one particular aspect of the BDI reasoning cycle – the selection of a plan for a given goal. For a more comprehensive description, please refer to the author’s thesis [9].

Both goals and plans are represented as types, the plan library of an agent is represented as a simple typelist. As described in Section 2, plan selection involves three steps: (i) the identification of relevant plans, (ii) the identification of applicable plans, and (iii) the selection of a plan for execution. All three steps are briefly described below. We start with the selection of relevant plans. Given a goal Goal, the set of relevant plans is the set of all those plans from the plan library that have the addition of Goal as their triggering event. Since both plans and goals are represented as types, the selection of relevant plans can be realised as a compile time function, as shown below³.

```cpp
using relevantPlans = Invoke<copy_if<
    PL,
    is_relevant<Goal>
>>;
```

`copy_if` represents a compile time function that accepts (i) the agent’s plan library PL (a static typelist) and (ii) another unary compile time function `is_relevant` that is applied to each item in the plan library and checks whether the respective plan is relevant⁴. If the relevance check is successful, then the plan is copied to the set of relevant plans, otherwise it is skipped. Remember that relevance checking happens at compile time. As a consequence, both the plan library and the resulting set of relevant are lists of types, and not lists of values!

As illustrated in Section 4.2 above, plans in BDI4MABS are purely declarative in nature and may contain further subgoals as part of their body. In order to make those plans executable, subgoals thus need to be replaced by appropriate plans; this recursive procedure is known as subgoal expansion. Since we assume that the full plan library is known at compile time, subgoal expansion can also happen at compile time. Given the set of relevant plans, subgoal expansion is done as follows.

```cpp
using newRelevantPlans = Invoke<transform<
    relevantPlans,
    expandPlanItem<PL,Applicability,Selector>
>>;
```

`transform` is a compile time function that accepts two arguments, (i) a compile time sequence (the set of relevant plans) and (ii) a metafunction (the recursive expansion function). It applies the metafunction to each element in the sequence and replaces it with the result of the function call (`transform` is thus equivalent to `map` in functional programming). The result of the operation is a set of fully expanded relevant plans.

The second step in the plan selection process involves checking the applicability of the relevant plans. The question whether a plan is applicable may depend on a number of aspects. It may be trivially simple (e.g. each plan is always applicable), in which case it can be decided at compile time; however, applicability may also depend upon runtime information (e.g. the agent’s beliefs), in which case it can only be decided at runtime. BDI4MABS supports both compile time and runtime applicability checking. We briefly describe the latter here; for a thorough description of supported compile time mechanisms, please refer to the author’s thesis [9]. In order to be able to manipulate the set of relevant plans of runtime, we first need to create instances of all relevant plans and store them in a vector. C++ vectors are homogeneous, i.e. it is not possible to store instances of different types in them. But each relevant plan is represented by its own type. A solution is to define a `variant` type which can be used as a basis for storing the differently-typed relevant plans in a single vector.

```cpp
using RP = Invoke<make_variant_over<relevant_plans>>;
std::vector<RP> relevantPlans;
```

³`Invoke<T>` is a convenience shortcut for `typename T::type` which essentially ‘executes’ the compile time calculation.
⁴This happens by simply comparing the given goal with the triggering goal of the respective plan. Both of them are known at compile time.
Applicability checking follows a pattern that is similar to that of relevance checking: the set of relevant plans needs to be iterated over and, for each plan, a checking function is called. If the result is positive, then the plan is copied to the set of applicable plans, otherwise it is skipped. The implementation is shown below.

```cpp
std::copy_if(
    relevantPlans.begin(),
    relevantPlans.end(),
    std::back_inserter(applicablePlans),
    [&as,&f,this](auto & p) {
        return apply_visitor(isApplicable<AS,Type>(as,f),p);
    });
```

The function is very similar to the relevance checking function shown further above. It accepts a sequence (the set of relevant plans) as well as another function that performs the actual applicability checking. The mechanism according to which applicability is determined is hidden in function `isApplicable` which can be overwritten by the user. In that way, custom applicability checking logic can be ‘injected’ into the framework. The overall result of the operation is the set of applicable plans. Despite their conceptual similarity, the crucial difference between relevance checking and applicability checking is that the former is a compile time function that operates on `types`, whereas the latter is a runtime function that operates on `values`.

Once the set of applicable plans has been determined, one of them needs to be selected for execution. The point at which the selection takes place depends on the previous step: if applicability checking happened at compile time, then the plan selection may happen either at compile time or at runtime; if applicability checking happened at runtime, then the final selection can only happen at runtime, too. The actual mechanism according to which selection takes place is left to the user. A wide variety of selection mechanisms is possible. For example, a plan may be selected randomly, it may be selected using simple prioritisation, or, it may be selected following a sophisticated utility evaluation. Their activation happens through static polymorphism: the actual selection mechanism is represented as a type parameter `Selector`, a metafunction class that accepts the set of expanded relevant plans, an applicability checking mechanisms and two more parameters which are not relevant here. The metafunction class triggers applicability checking and performs the final selection. It is invoked as follows.

```cpp
using type = typename Selector::template apply<
    Applicability,
    newRelevantPlans,
    void,
    g_sEmpty
>;
```

This concludes the description of plan selection. For space limitation, we could only provide a very superficial description of the implementation here; for further information, please refer to the author’s thesis [9]. In the next section, we provide a performance evaluation of BDI4MABS with respect to both compile time and runtime consumption.

## 5 Evaluation

For the performance evaluation of the framework, we return to the simple party scenario introduced in Section 3 and further elaborated upon in Section 4.2 (the code can be found on the paper website [10]). For the first experiment, we focus on the impact of compile time and runtime computation on the performance.

For each scenario, we run the simulation model for 1000 ticks and with 10, 100, 1000, and 10000 agents, respectively. All experiments were performed on a Dell Latitude E6420 with 2 Intel® Core™ i7-2620M CPUs (2.7 GHz each), 8 GB of memory, and Linux Mint 17.1 Cinnamon 64-bit (kernel version 3.13.0-37-generic) as operating system. For profiling, the Linux tool `time` was used. It reports three different aspects of the runtime of an application: (i) the elapsed real time between invocation and termination (`‘real’`), (ii) the user CPU time (`‘user’`), and (iii) the system CPU time (`‘system’`). The numbers shown below are averages of 10 independent runs.

In the first experiment, we study the impact of population size on runtime. The results are shown in Table 1 (left). As expected, runtime increases linearly with the size of the population. As described in Section 4.2, BDI4MABS supports different applicability checking mechanisms. For example, if the criterion according to which a plan is to be marked as applicable is known at compile time, then applicability checking can be performed statically, i.e. at compile time. Table 1 (right) shows the results for the case where all plans are marked applicable.
<table>
<thead>
<tr>
<th>#agents</th>
<th>real(s)</th>
<th>user(s)</th>
<th>system(s)</th>
<th>#agents</th>
<th>real(s)</th>
<th>user(s)</th>
<th>system(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.09</td>
<td>0.08</td>
<td>0.00</td>
<td>10</td>
<td>0.09</td>
<td>0.08</td>
<td>0.00</td>
</tr>
<tr>
<td>100</td>
<td>0.74</td>
<td>0.74</td>
<td>0.00</td>
<td>100</td>
<td>0.60</td>
<td>0.60</td>
<td>0.00</td>
</tr>
<tr>
<td>1000</td>
<td>7.26</td>
<td>7.26</td>
<td>0.00</td>
<td>1000</td>
<td>5.97</td>
<td>5.96</td>
<td>0.00</td>
</tr>
<tr>
<td>10000</td>
<td>75.81</td>
<td>75.74</td>
<td>0.01</td>
<td>10000</td>
<td>61.60</td>
<td>61.49</td>
<td>0.04</td>
</tr>
</tbody>
</table>

Table 1: Influence of population size on execution time with compile-time applicability checking and runtime time plan selection (left) and runtime applicability checking and runtime time plan selection (right) for 1 product

at compile time. The numbers indicate that runtime can be reduced if some of the calculations can be performed at compile time.

<table>
<thead>
<tr>
<th>#agents</th>
<th>real(s)</th>
<th>user(s)</th>
<th>system(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.023</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>100</td>
<td>0.11</td>
<td>0.107</td>
<td>0.00</td>
</tr>
<tr>
<td>1000</td>
<td>0.857</td>
<td>0.703</td>
<td>0.143</td>
</tr>
<tr>
<td>10000</td>
<td>12.947</td>
<td>9.35</td>
<td>3.563</td>
</tr>
</tbody>
</table>

Table 2: Influence of number of products (= applicable plans) on execution time with runtime applicability checking and runtime time plan selection for 1 agent

In the next example, we study the impact of parametrised plans on the overall performance of the framework. To that end, we increase the number of products being available. Remember that a separate plan is instantiated for each product. The resulting set of plans serves as an input to applicability checking and plan selection. The results are shown in Table 2. The numbers indicate that the framework scales linearly with the number of plans and performs well, even in the presence of large sets of plans.

<table>
<thead>
<tr>
<th>#agents</th>
<th>Scenario</th>
<th>real</th>
<th>user</th>
<th>system</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Baseline</td>
<td>0.353</td>
<td>0.287</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>BDI</td>
<td>6.131</td>
<td>5.797</td>
<td>0.317</td>
</tr>
<tr>
<td></td>
<td>Factor</td>
<td>17.37</td>
<td>20.2</td>
<td>5.3</td>
</tr>
<tr>
<td>100</td>
<td>Baseline</td>
<td>0.346</td>
<td>0.273</td>
<td>0.065</td>
</tr>
<tr>
<td></td>
<td>BDI</td>
<td>6.161</td>
<td>5.829</td>
<td>0.312</td>
</tr>
<tr>
<td></td>
<td>Factor</td>
<td>17.8</td>
<td>21.4</td>
<td>4.8</td>
</tr>
<tr>
<td>1000</td>
<td>Baseline</td>
<td>0.353</td>
<td>0.282</td>
<td>0.065</td>
</tr>
<tr>
<td></td>
<td>BDI</td>
<td>6.177</td>
<td>5.861</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>Factor</td>
<td>17.5</td>
<td>20.8</td>
<td>4.6</td>
</tr>
<tr>
<td>10000</td>
<td>Baseline</td>
<td>0.353</td>
<td>0.282</td>
<td>0.065</td>
</tr>
<tr>
<td></td>
<td>BDI</td>
<td>6.197</td>
<td>5.865</td>
<td>0.311</td>
</tr>
<tr>
<td></td>
<td>Factor</td>
<td>17.6</td>
<td>20.8</td>
<td>4.8</td>
</tr>
</tbody>
</table>

Table 3: Impact of BDI framework usage on compilation time (left) and runtime (right) for different population sizes, with runtime applicability checking and runtime time plan selection

The second experiment concerns the general compilation time and runtime overhead incurred by the framework. For that purpose, we developed a baseline scenario which contains the same functionality with respect to the agent’s behaviour but does not make use of the BDI framework. The code of the baseline scenario can be found on the paper website [10]. The results of the experiment are shown in Table 3. It is apparent that usage of the framework increases compilation time by a constant factor of $\approx 17–18$ and execution time by a constant factor of $\approx 30–40$. There is certainly room for improvement. We intend to investigate further optimisations as part of our future work.
6 Conclusions

In this paper, we described our efforts to address this problem and provide an implementation of the BDI architecture with a particular focus on its application in the context of agent-based modelling and simulation. Rather than sacrificing behavioural richness for the sake of efficiency, the main challenge was to combine the best of both worlds: BDI4MABS aims to be easily accessible and seamlessly capable of being integrated into an existing simulation environment whilst still providing a high level of performance at runtime. However, with almost two decades of research, the BDI architecture is of great complexity and it would be bold to claim that the developed framework offers the same functionality as established AOP languages and frameworks. The most important limitations of the framework are quickly summarised below.

**Event types:** At the moment, BDI4MABS only supports two types of event: goal addition and removal. As a consequence, the framework does not produce events when a change to the belief base occurs. It is thus currently not possible for an agent to react to belief addition or removal.

**Triggering events:** Only goal addition is supported as a triggering event for plans. It is currently not possible to formulate a plan that reacts to goal removal.

**Plan failure:** Due to the lack of goal removal as the triggering event of a plan, it is currently not possible to implement plan failure handling through contingency plans.

**Agent communication:** Although easily implementable as basic actions using plain C++ in the framework, building blocks for inter-agent communication are currently not integrated into the framework. The lack of agent communication mechanisms also excludes the possibility for plan exchange.

**Acknowledgments:** The author would like to thank Jeremy Gibbons and Michael Wooldridge for their helpful comments and suggestions, as well as Andrew Skates and the entire team of Sandtable Ltd in London for their support.

References


