Programming Safe Agents in Blueprint

Alex Muscar
University of Craiova
Programmers are craftsmen, and, as such, they are only as productive as theirs tools allow them to be
Introduction
Agent Oriented Programming

- has been around for 20+ years
- was intended as a higher level alternative to OOP
- many regarded it as “a revolution in software”
Agent Oriented Programming

- has failed to gain wide traction
- is regarded as an experimentation tool for AI
- the community lacks focus
**Blueprint**

**Premise:** There is still place for a solution that is both high-level, yet practical

**Design goals:** an agent oriented programming language, focusing on concurrency, static safety, ease of use and extensibility
Defining terms

**agents:** computational entities that (i) have their own thread of control and can decide autonomously if and when to perform a given action; and (ii) communicate with other agents by asynchronous message passing
Defining terms

concurrency: the composition of independently executing entities
Defining terms

*consistency: data consistency*, rather than logical consistency

*The ‘A’ in A.C.I.D.*
Defining terms

**scalability**: (i) the ability of the runtime to gracefully handle a growing number of agents executing concurrently; and (ii) the ability of the language to gracefully handle growing code bases.
Background and Motivation
Why another language?

Blueprint’s development was motivated by my experience:

- developing the prototype of a dynamic negotiation mechanism in Jason
- teaching agent technologies to undergraduate students using JADE
Jason’s advantages

- high level
- domain oriented
- most popular AOPL
Jason’s advantages

- active community
- regularly updated
- good documentation
Jason’s disadvantages

- limited in scope
- latently typed
- exotic syntax
Jason’s disadvantages

• slow interpreter
• not scalable
JADE’s advantages

- manifestly typed
- scalable
- most popular agent framework
JADE’s disadvantages

- lacks expressivity
- syntactic noise
Comparing Jason and JADE

<table>
<thead>
<tr>
<th></th>
<th>Jason</th>
<th>JADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency support</td>
<td>⃝</td>
<td>⃝</td>
</tr>
<tr>
<td>Language scalability</td>
<td>Ⓞ</td>
<td>Ⓝ</td>
</tr>
<tr>
<td>Safety</td>
<td>Ⓞ</td>
<td>Ⓝ</td>
</tr>
<tr>
<td>Expressivity</td>
<td>Ⓝ</td>
<td>Ⓞ</td>
</tr>
<tr>
<td>Extensibility</td>
<td>Ⓞ</td>
<td>Ⓝ</td>
</tr>
</tbody>
</table>

This is the route chosen by Jason, which runs on the JVM. But there is a tension between Jason's high level semantics and JVM's lower-level features so developers using Jason frequently need to use an "escape hatch": they need to drop into Java to develop parts of their agent system. This is a dangerous route since the developer might be tempted to use the path of least resistance, and develop the better part of the agent system in Java (this issue is also covered in [75]). One of the goals of BLUEPRINT is to make the native libraries easy to access from the language itself, and the language expressive enough to make their use comfortable.

These criteria will be the same ones that guide BLUEPRINT’s design.

Table 2.1 shows the comparison of Jason and JADE according to the above criteria. Backing up the intuition built by 2.1, JADE has full ratings in the criteria related to safety and language scalability. This is expected since it benefits from the years of efforts pour into the Java programming language and the JVM. Conversely, Jason ranks fairly low on safety and scalability related issues, but it shines in expressivity, being very domain oriented. This is indeed a point where JADE suffers, being overly verbose.
Monadic Foundations for Concurrent Agents
Monads

- originated in category theory
- structures that represent computation
- usually composed of a type constructor and two operations
Why monads?

• F#’s computation expressions are syntactic sugar for monads
• they are an elegant way of expressing the composition of concurrent computations
• they have been thoroughly studied
A closer look at concurrent computations

- they start *now* and they will finish sometime *in the future*
- we need to react when a concurrent computation ends
- we need to combine concurrently running computations
A closer look at concurrent computations

• the reaction to the completion of a concurrent computation is its continuation

• look at plans as being split in two: the actions ran thus far and the actions that are still to be executed

• a promise that a set of actions will get executed at some point

• this hints at a way of composing plans
The \textit{Promise} monad

A promise for a value of type $\alpha$ is a function which receives a handler that can be called with the value of the promise, and it produces a value of type $\beta$. The \textit{type constructor} for the Promise monad, $M_{\text{promise}}$, is defined as:

\[
M_{\text{promise}} = (\alpha \rightarrow \beta) \rightarrow \beta
\]
The *Promise* monad

The *unit operation* takes a value and returns a promise that will pass the value as an argument to the promise’s handler:

\[
\text{unit}_{\text{promise}} = \lambda x. \lambda k. k \ x
\]
The Promise monad

The bind operation takes a promise and a continuation of the promise, and returns a promise that will invoke the continuation in a context where the result of the promise is available:

$$\text{bind}_{\text{promise}} = \lambda m. \lambda k. \lambda c. \text{run} \ m \ (\lambda x. \text{run} \ (k \ x) \ c)$$

where run is a function that executes a promise with the given callback.
Conclusions

- the *Promise* monad is actually the well known *CPS* monad
- we can use monads to structure concurrent plans
- we can employ the same strategy as F#: use monads internally and perform code rewrite
The Blueprint Language
Blueprint is meant to be

- high level (e.g. agents, plans)
- safe (e.g. static types, channel protocols)
- easy to learn (e.g. C-like syntax)
- easy to use for concurrent applications
From revolution to evolution

- take a step back and look at agents as an evolution of the OOP and the Actors model
- concurrently executing agents with reactive behaviours
- respects Shoham's definition of AOP as a specialisation of OOP in the sense of the Actor model
The road to Blueprint

- agents are reactive and autonomous entities
- send messages asynchronously to mitigate deadlocks
- react to incoming events serially in order to avoid race conditions
- use monads to structure compose computations
Communication channels

- agents use *bidirectional* and *asymmetric* channels to exchange messages
- messages sends are asynchronous (i.e. non-blocking), while receives are synchronous (i.e. blocking)
- preserve message ordering
- they belong to exactly one agent
- they are introduced by the **chan** keyword
Channel endpoints

- an *exporting* endpoint, and an *importing* endpoint
- the exporting endpoint is used by the owner of the channel, while the importing endpoint can be handed off to other agents
- each endpoint has an *ordered, unbounded* message queue
- channel endpoints are first order entities (i.e. they can be passed as arguments and returned as values)
agent Account(init: int, impChan: BankAccount.Imp) {
    chan c = BankAccount.make()
    bel balance = init

    plan Start() {
        val msg = <-c.Exp.operation;
        match msg {
            case deposit(amount):
                val currentBalance = balance.take()
                balance.put(currentBalance + amount)
            case withdraw(amount):
                val currentBalance = balance.take()
                balance.put(currentBalance - amount)
            case transferTo(acc, amount):
                acc <- deposit(amount);
                val currentBalance = balance.take()
                balance.put(currentBalance - amount)
        }
    }
}
agent Account(init: int, impChan: BankAccount.Imp) {
    chan c = BankAccount.make()
    bel balance = init

    plan Start() {
        val msg = <-c.Exp.operation;
        match msg {
            case deposit(amount):
                val currentBalance = balance.take()
                balance.put(currentBalance + amount)
            case withdraw(amount):
                val currentBalance = balance.take()
                balance.put(currentBalance - amount)
            case transferTo(acc, amount):
                acc <- deposit(amount);
                val currentBalance = balance.take()
                balance.put(currentBalance - amount)
        }
    }
}
agent Account(init: int, impChan: BankAccount.Imp) {
    chan c = BankAccount.make()
    val balance = init

    plan Start() {
        val msg = <-c.Exp.operation;
        match msg {
            case deposit(amount):
                val currentBalance = balance.take()
                balance.put(currentBalance + amount)
            case withdraw(amount):
                val currentBalance = balance.take()
                balance.put(currentBalance - amount)
            case transferTo(acc, amount):
                acc <- deposit(amount);
                val currentBalance = balance.take()
                balance.put(currentBalance - amount)
        }
    }
}
Channel protocols

- declarative mechanisms of enforcing proper message exchange between agents
- specify the flow of the data between the communicating entities (i.e. the order, and direction in which messages are sent)
- introduced by the `proto` keyword
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}

proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
Channel protocols

• protocols are designed from the perspective of the agent initiating the interaction (i.e. the exporting endpoint)

• there is no need to specify the dual protocol since it can be automatically derived by swapping direction specifiers
Concurrency and beliefs

- channels and protocols are a good way to control inter-agent concurrency
- we need a way to control intra-agent concurrency as well
- use synchronised mutable variables (mvars)
mvars

- one-place buffers which can be in one of the two states: empty or full
- two basic operations: take, and put
- calling take on a full mvar immediately returns the value and marks the mvar as empty
- If a take call is issued on an empty mvar, the calling thread of execution is blocked until the mvar becomes full
- the semantics of the put operations are similar
mvars

• the locks are not directly manipulated by the programmer, instead this is the job of the underlying implementation

• given the relatively low level, blocking nature of mvars (when compared to message passing), the risk of deadlock is still present
Beliefs as mvars

- Blueprint implements all beliefs as mvars
- beliefs are introduced by the `bel` keyword
- beliefs have two methods: `take()` and `put()`
Formal model sketch

- the semantics is defined via a CPS transform to a core language
- the core language is a small functional language
Proto $\pi$ ::= proto $id \{\sigma_0 \ldots \sigma_n\}$ Protocol definition
State $\sigma$ ::= $id : \overline{\pi} \gg \overline{id}$ Protocol state
MsgFlowExp $\mu$ ::= $\mu_0 \rightarrow \ldots \rightarrow \mu_n$ Message flow expression
MsgExp $\mu$ ::= $in \ id(id_0 \ldots id_n)$ Message receive expression
TargetStates $\sigma$ ::= $id_0 \lor \ldots \lor id_n$ Target states
::= $out(id_0 \ldots id_n)$ Message send expression
Plan $p$ ::= plan $id(p_0 \ldots p_n) \{e\}$ Plan definition
Meth $m$ ::= def $id(p_0 \ldots p_n) \{e\}$ Method definition
Stmt $s$ ::= val $id = e$ Value binding
::= var $id = e$ Variable binding
::= $e_0; \ldots; e_n$ Sequence
Exp $e$ ::= $n$ Numeral
true Boolean literal
false Boolean literal
"s" Literal
$id$ Reference
$e.id$ Field reference
$e[i]$ Array element reference
$e_1 \ op \ e_2$ Binary operator
$e(e_0 \ldots e_n)$ Function call
$e$ Channel receive
$e_1 \leftarrow e_2$ Channel send
$e_1 := e_2$ Assignment
$\epsilon$ Empty expression
\[
[\text{plan } id (p_0 \ldots p_n) \{ e \}] \equiv \text{let } id = \lambda (p_0 \ldots p_n) . \lambda \kappa . [e]
\]

\[
[e_{\text{plan}}()] \equiv \lambda \kappa . \text{asyncstart} (e); \kappa()
\]

\[
[e_1; e_2] \equiv \lambda \kappa . [e_1] (\lambda (). [e_2] \kappa)
\]

\[
[a] \equiv \lambda \kappa . a; \kappa()
\]

\[
[id := \text{take}(e)] \equiv \lambda \kappa . \text{suspend} (e, \lambda (). \text{set}(id, \text{take}(e)); \kappa())
\]

\[
[\text{put}(id, e)] \equiv \lambda \kappa . \text{put}(id, e); \text{signal} (id); \kappa()
\]

\[
[id := \text{recv}(e)] \equiv \lambda \kappa . \text{suspend} (id, \lambda (). \text{set}(id, \text{take}(id)); \kappa())
\]
\[
\text{Step:} \quad (\{e\} \cup A, Q, P) \leadsto (\{e'\} \cup A, Q, P), \text{ if } e \mapsto e'
\]
\[
\text{Suspend:} \quad (\{\text{suspend}(id, e)\} \cup A, Q, P) \leadsto (A, Q, P \cup \{id \to e\})
\]
\[
\text{Schedule:} \quad (A, \{e\} \cup Q, P) \leadsto (A \cup \{e\}, Q, P)
\]
\[
\text{Signal:} \quad (\{\text{signal}(id)\} \cup A, Q, P \cup \{id \to e\}) \leadsto (A, Q \cup \{e\}, P)
\]
\[
\text{Async-Start:} \quad (\{\text{asyncstart}(e)\} \cup A, Q, P) \leadsto (A, Q \cup \{e\}, P)
\]
Implementation considerations

• Blueprint is built on top of the CLR framework

• The CLR contains a performant *Virtual Machine* with a *Just In-time Compiler* and a *Garbage Collector*

• we use the thread-pool pattern for scheduling agent reactions to incoming messages
and a Garbage Collector (GC). The development of a VM that can be on par with the one in .NET or Mono would be a serious undertaking, and is out of the purpose of this thesis. As we will see in chapter 6, the performance of programs written for both .NET and Mono is on par with that of programs running in top of the JVM.

Figure 5.5: The Common Language Runtime high level overview.

Figure 5.5 shows a high level overview of the CLR. Languages targeting the CLR translate the source code to an intermediate representation called Common Intermediate Language (CIL), which is a stack based byte code, very similar to the JVM bytecode. CIL code is translated by the JIT to native code an executed. This approach makes the compiler easier to write, because CIL is higher level than machine code, and that code produced by multiple languages targeting the CLR can seamlessly interoperate, e.g. BLUEPRINT agents can use libraries written in C#.

4 This image is a freely licensed media file from the Wikimedia Commons.
The thread-pool pattern

• a model where a (possibly fixed) number of threads—called worker threads—is created in order to execute waiting tasks—usually stored in a queue

• a worker thread requests the next pending task, and if one is available it runs it to completion

• the thread may sleep or it may request another task once the current task has finished
5.6. IMPLEMENTATION DETAILS

Figure 5.6: An example of a simple thread-pool with waiting tasks (blue), running tasks (red), and completed tasks (yellow). In such situations, the thread-pool provides a mechanism to put the executing task in a special "waiting" queue, and free the worker thread to service another queued task. Waiting tasks are rescheduled when the event they are waiting for occurs.

Mapping plans to tasks

In order to allow the asynchronous execution of plans, we will employ a scheme similar to that presented in [97]. The essential aspects of the approach are illustrated in figure 5.3: we will perform a rewrite of the source code in CPS [97], and use a thread-pool for scheduling reactions to pending events, e.g. communication.

This model has proven quite effective: it has been employed by the F# programming language since 2007, and recently C# has adopted a similar solution. Instead of forcing the developer to write the code in explicit CPS, which is what Jason does by requiring separate event handlers for each message received, we will let the compiler take care of the plumbing.
The thread-pool pattern

- it scales well for I/O-bound tasks
- the performance degrades when it has a lot of CPU-bound tasks
Future Directions
• investigate *code reuse* (most probably via some form of inheritance of prototypic delegation)

• investigate an extension of the concurrency model, based on the *Join calculus*

• give a full formal account of the language
• define a mechanism similar to channel protocols to characterise agent *environments*

• further investigate the object capability model in the context of security in AOP

• develop a JVM backend for Blueprint

• develop tooling for the language (i.e. plugins for popular IDEs)
Thank you. Questions?