DOI 10.12694/scpe.v14i3.859
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 14, Number 3, pp. 181–187. http://www.scpe.org © 2013 SCPE

JOIN PATTERNS FOR CONCURRENT AGENTS (POSITION PAPER)

ALEX MUSCAR*

Abstract. Even though the Agent Oriented Programming (AOP) paradigm has been introduced over 20 years ago, the following is still modest. We believe this is partially due to the lack of the community to address such real world issues as concurrency and distribution.

In this paper we take a look at the possibility of combining elements from the Join Calculus, namely *join patterns* with an AOP programming language, in our case Jason. We discuss some of the implications, advantages and possible drawbacks.

Key words: asynchronous programming, agent-oriented programming, concurrency

1. Introduction. Since the introduction of *Agent Oriented Programming* (AOP) two decades ago [14] the world of software has evolved considerably. The uprise of mobile and distributed applications has made the complexity of software increase exponentially. Another important change is that the number of cores in the modern processors is increasing steadily, giving rise to new opportunities. In the meantime, the AOP community has made little progress in making the agent paradigm a viable solutions of these kinds of problems. We believe that this is a cultural issue: the AOP community has been closed to external influences, thus it has not been able to take advantage of research efforts in other fields. This has started to change recently though, with efforts to make agent languages more oriented towards real world problems [12]—the authors propose a new agent language called SimpAL which addresses some shortcomings of existing agent languages such as weak type systems. The need for better agent languages is also clear from experience reports such as [11], which describes the experience of using Jason[1] [4], one of the best known agent languages, for developing a medium software project, and the problems faced by the author. Another relevant and recent effort in the area on adapting agent languages to real world issues has been presented in [6]—the authors have ported the Jason language to run on the Erlang virtual machine, a highly scalable and concurrency oriented platform. For an in-depth review of agent languages we refer the reader to [1].

In this paper we propose to take a look at how the *Join Calculus* [7], a process calculus for distributed systems, can be integrated with AOP languages, namely with Jason. More specifically, we propose to integrate *join patterns* [7] in the *triggering events* of Jason plans (see sec. 2). We have chosen Jason because it is one of the most popular and mature AOP languages. As of the Join Calculus, we have chose it since we feel join patterns integrate well with Jason. The author of [9] shows how the Join Calculus relates to *Constraing Handline Rules* (CHR), another language from the logic programming language family. Elements from the Join Calculus have also been incorporated in research branches of popular languages such as C# [2], leading to interesting results.

The rest of the paper is structured as follows: in sections 2, and 3 we make short overviews of Jason and the Join Calculus. In section 3.1 we introduce the proposed approach together with a working example. We extend the discussion in section 4. In section 5 we talk about some problematic aspects of the proposed approach. We finally conclude the paper in section 6.

2. An Overview of Jason. In order to make the code samples in this paper easier to understand, in this section we are going to give some details on the syntax of Jason. We will also very briefly and informally look at the Jason semantics.

2.1. The Syntax of Jason. Jason language has the notions of plans, beliefs, and goals. A program is made out of three sections:

- **Beliefs** are very similar to facts in Prolog [5] with the operator `&` replacing `,` in conjunctions. The belief base can be manipulated with the aid of the `+` and `-` operators, in the *body* of the plan. They

*University of Craiova, Bvd. Decebal, Nr. 107, 200440, Craiova, Romania (muscar@gmail.com). This work was partially supported by the strategic grant POSDRU/CPP107/DMI1.5/S/78421, Project ID 78421 (2010), co-financed by the European Social Fund - Investing in People, within the Sectoral Operational Programme Human Resources Development 2007 - 2013.

[1] http://jason.sourceforge.net/

add, and delete respectively, a belief from the belief base. Since these operations usually come in pairs, a deletion followed by an addition, the shortcut operator `-+` is used for this purpose;
- **Goals** are introduced by the `!` operator[2];
- **Plans** which follow the generic form `triggering_event : context <- body.`, are intended to handle goals. *Triggering events* match goals and message. The only relevant triggering events for this paper are goal addition and message arrival, denoted by atoms with the `+!` and the `+` prefixes.

Internal actions allow agents to call methods from the underlying platform. They are qualified by their package name (like in Java, e.g. `package.action`). Predefined actions do not have a package name, but they retain the leading period (e.g. `.send`). For further details we direct the interested readers to [4] for further details.

2.2. Informal Jason Semantics. Jason uses the concept of *beliefs*, which represent the agent's view of itself and the world, *goals*, which represent states in which the agent wants to be, and *plans*, which are "recipes" for action. An agent is driven forward by a *reasoning cycle* in which the agent: (i) perceives the environment, and updates its beliefs, (ii) receives messages, (iii) selects an event and the plans that can handle it, and (iv) executes (a step of) the plan. Note that running plans are organised as *intentions*, which are stacks of currently executing plans. The Jason interpreter uses a single OS thread for executing all the intentions, thus a single intention is active at any given time. While this is a (very) simplified description of the reasoning cycle, it has all the information that we need.

3. Combining Join Calculus and Practical Reasoning Agents. In this section we take a brief look at the Join Calculus and how join patterns can be integrated in Jason in order to allow intra-agent concurrency.

The core Join Calculus is similar to a call-by-value lambda calculus enriched with concurrency primitives [7]: `run`, which starts a concurrent evaluation of an expression in a new *process*; the | operator for parallel composition of processes; and *join patterns* which serve for process synchronisation. Processes are represented as unique names with arguments, akin to compound terms in Prolog.

Since the Join Calculus is based on the *Chemical Abstract Machine* (CHAM) [3], its semantics is defined in terms of *reaction rules*. In the case of the Join Calculus, a reaction rule is composed from a join pattern and a join body, and has the general form $J \triangleright B$, where J is the join pattern, and B is the join body. A join pattern is constructed from a list of processes. The semantics of a join pattern is as follows: whenever all the processes in a join pattern are active in the program, they are consumed, and the processes in the join body are added to the set of active processes. For an example see sec. 3.1.

For an in-depth overview we refer the reader to [7]. Also note that this is only the core of Join Calculus. The following sections will refer to the richer flavour of Join Calculus implemented by the JoCaml system [10].

3.1. Join Patterns. *Join patterns* are a declarative way of defining synchronisation patterns between *channels* (or *ports*). Listing 1 shows the implementation of an unbounded buffer using a join pattern [7].

```
put(x) & get() = reply x to get
```

Listing 1: A simple unbounded buffer implemented using join patterns

The succinctness of the code is due to the semantics of the join patterns. As mentioned in sec. 3, in order to *activate* the pattern, messages must be sent to on both channels. When a pattern is activated its body is executed. In a scenario where multiple producers would send values on the `put` port none of them would block, because `put` is *asynchronous*. If a consumer were to call `get` and and no prior `put` message was sent to the buffer, then the caller would *block* until such a message is sent. This is because `get` is a blocking (or *synchronous*) channel. This is specified by returning a value on this port in the body of the pattern. When a join pattern is activated, the values that triggered the pattern are *consumed*. This is an important issue to which we will return later.

[2]To be precise the `!` introduces *achievement* goals. Jason also has *test* goals introduced by the `?` operator, but they are not yet supported by our dialect yet, and as such relevant for the purpose of this paper.

3.2. Declarative Beliefs and Goals. As mentioned earlier, Jason features declarative beliefs and goals. Listing 2 show a part of a coffee maker agent written in Jason. While not complete, the listing is enough to show the flavour of developing agents in Jason.

4. Declarative Concurrency for Agents. We believe that by combining the declarative approach of specifying agents by using knowledge, beliefs, and goals on one side and join patterns on the other we can get a stronger language that maintains its declarative nature, but which also gains declarative concurrency. Assuming a mental state similar to the one in listing 2[3], listing 3 shows the implementation of a coffee maker agent in Jason extended with join patterns.

We will now take an in-depth look at this version of the coffee maker agent. While in Jason the triggering events use predicates, Jason extended with join patterns uses a mixture of channels and predicates. In listing 3, channels are written with a bold font face, while predicates are written with a regular typeface. Some readers may already be wondering what happens with variables during the resolution process? How do they unify?

When a pattern is activated, the variables in the synchronous channel(s) are bound to the values that activated the pattern. The variables in the asynchronous channels are left unbound and they get values by substitution during the resolution process. For *every* possible substitution a test is made to see if the substitution matches values present in the channel—not necessarily the latest value, channels are scanned for the required value. When a match is found, the matching values are consumed, the free variables are bound, and the body of the pattern is executed. If there is no match, or the whole pattern can not be grounded the value in the synchronous channels is put back in the buffer and the body of the pattern does not execute.

Algorithm 1 shows the algorithm defined above. The channels are extracted to form a join pattern, $\{c\} \cup A_c$—c is the first channel in the pattern—while the predicates are extracted to form a linear sequence of clauses, $C = C_1, C_2, \ldots, C_n$. When the pattern is activated the mental state of the agent is used to find the *unifier* σ. Then we scan (i.e. look through the values in a channel's message queue without consuming them) the remaining asynchronous channels, using the Scan method, to see if we can find the values for the unifying substitution, using the Var function to extract the variables from the channel. If we find matches for every channel, then we execute the body, b, of the action pattern to which we apply σ. In the opposite case, we re-queue the value that triggered the computation in the synchronous channel, which we obtain by using the Val function, so that it can be used in future matches.

There are a couple of restrictions that we impose on our model:

- There must be *at most one* synchronous channel in a pattern (note that there can be patterns made only out of asynchronous channels); and
- Before a pattern can be activated all the variables in it must have a value.

There might be multiple *activable* patters at any moment. The method we presented would treat each pattern in turn, from top to bottom, using the previously illustrated algorithm to decide which pattern can be activated. Once a pattern is activated no other pattern will be considered in that step. This is similar to how Prolog defines predicates out of multiple clauses or to how functions are defined in Haskell, by defining multiple equations with the same name. In our case, the blocking channel can be considered the central part.

When we said, above, that the mental state of agent enhanced with join patterns is not exactly the same as that of the Jason agent we were referring to the fact that beliefs in the proposed language are represented as channels, rather than predicate. This has advantages as well as some drawbacks. The main advantage of this approach is that we have a uniform model, where the beliefs can be simply updated by sending messages, instead of receiving a message and duplicating the information in the belief base. The main drawback is that once a patter in which a belief takes part is fired, the value of the belief is lost. In the case of depleatable resources, such as water or coffee beans in the case of the coffee maker agent, this acts in our favour since we don't need to manually manage the resource (first add it, then retract it). But for more long-lasting beliefs this is undesirable. If we wish to keep this representation model, we need to find a way around this issue. We have come up with two very similar approaches:

- *Annotations* introducing metadata in the language (e.g. Java annotations or C# attributes). This could prove beneficial in the long run since this is a general approach which could be used in other cases

[3]This is not entirely the case, we will come back to this issue later

```
// Rules

neededFor(coffee, water).
neededFor(coffee, grounds).
neededFor(grounds, grounds, beans).

canMakeIt(M, P) :-
  canMake(M, Prods) &
  member(P, Prods).

// Beliefs

have(water).
have(beans).

canMake(maker, [coffee]).

// Initial goals

!have(coffee).

+!have(P)
<- // If we want to have some product P
   // then make it.
   !make(P).

...

+!make(P)
: // If we need R in order to make P
  neededFor(P, R) &
  // and we don't have it
  not(have(R)) &
  // but we can make it ourselves
  .my_name(Me) & canMakeIt(Me, R)
<- // then we just set a goal to have R
   !have(R).

+!make(P)
: // If we need R in order to make P
  neededFor(P, R) &
  // and we don't have it
  not(have(R)) &
  // but we know someone else who can
  // make it
  .my_name(Me) & canMakeIt(Maker, R) &
  Me ¯ Maker
<- // the we just tell that agent to make it
   .send(Maker, achieve, have(R)).

...
```

Listing 2: A coffee maker agent

```
Make(X) & neededFor(X, Y) & have(Y)
<- .reply(Make, X).
Make(X) & neededFor(X, Y)  & neededFor(Y, Z) &
  have(Z)
<- Make(Z).
Make(X) & neededFor(X, Y) & canMake(Producer, Y)
<- .send(Make, have(Y)).
```

Listing 3: A coffee maker agent in Jason with join patterns

Algorithm 1 Jason with join patterns pattern activation algorithm

```
procedure ACTIVATE($c, A_c, C, b$)
    $\sigma \leftarrow$ UNIFY($C$, $metal\_state$)
    $activate \leftarrow True$
    for $c_a$ in $A_c$ do
        $value \leftarrow$ VAR($c_a$)$\sigma$
        if $\neg$ SCAN($c_a, value$) then
            $activate \leftarrow False$
        end if
    end for
    if $activate$ then
        $(b\sigma)()$
    else
        $c \leftarrow$ VAL($c$)
    end if
end procedure
```

as well. In this case an annotation `@keep` could be introduced to mark a channel whose value is to be kept even after a pattern has been activated; and

- *Special-purpose predicates* in the spirit of `bel` or `goal` from the GOAL language [8].

Irrespective of the design decision in this case a mechanism for manually consuming the value from a channel is needed. Listing 4 illustrates this issue. When an agent enters in a new environment (signaled by the runtime via the channel `EnterEnv`) it starts believing that it is in that environment (by means of the `inEnv` channel/belief in our example). When, later, the agent needs to know in which environment it's operating in and it queries its `inEnv` channel/belief, if the pattern matches the value of the environment will be consumed. This is clearly undesirable.

```
EnterEnv(Env) & canMake(X)
<- .send(Env, register(canMake(X))), inEnv(Env).

...

Make(X) & neededFor(X, Y) & inEnv(Env)
<- // Find out who can produce Y
   .send(Env, ask, query(canMake(Producer, Y)));
   // Start believing that Producer can make Y
   +canMake(Producer, Y)

...
```

Listing 4: Unintentionally consuming values

In listing 5 we illustrate both approaches outlined above. For the above mentioned reasons we favor the annotations approach.

```
...

// Specifying that a value is to be
// kept in the channel via annotations

Make(X) & neededFor(X, Y) & @keep inEnv(Env)
<-
// Find out who can produce Y
  .send(Env, ask, query(canMake(Producer, Y)));
// Start believing that Producer can make Y
  +canMake(Producer, Y)

// The same effect via special-purpose
// predicates

Make(X) & neededFor(X, Y) & keep(inEnv(Env))
<-
// Find out who can produce Y
  .send(Env, query(canMake(Producer, Y)));
// Start believing that Producer can make Y
  +canMake(Producer, Y)

...
```

Listing 5: Keeping values unconsumed

5. Open Issues. Besides the belief representation issue, which is a fundamental design decision for the language, there are a series of other open issues/design decisions:

- **Goal representation** Similarly to the belief representation issue, we have the goal representation issue. Again, we have two options: i) keep the predicate representation that Jason uses and introduce inconsistency in the language—different mechanisms are used to represent beliefs and goals; or ii) use channels, just like in the case of beliefs. Maybe a combination of the two is possible: since a goal is a desired state of the world, and an agent's view of the world is modeled by its beliefs it seems natural to represent goals as patterns of channels and predicates (just like action patterns). This is an important design decision and its effects need to be carefully considered;
- **Channel visibility** In the previous examples we have operated under the assumption that the capitalization of the channel names dictates their visibility: upper-cased channel names represent public channels, while lower-case names denote private channels;
- **Advertising capabilities** In listing 4 the agent exposed its capabilities by directly advertising them to the environment (i.e. `.send(Env, register(canMake(X)))`). While this works for a toy example it can get unwieldy to expose capabilities via this mechanism. One alternative would be to look at agents as abstract types that conform to an interface (like in the ML family of languages), and register that interface with the environment;
- **Communication** So far we haven't said anything about communication. What is the level at which agents communicate? In our previous examples agents can communicate at the knowledge level, by freely exchanging compound terms (functors). It is still not clear if this is the best approach, though it seems like it; and
- **Predicates in the body of a pattern** In listing 4 we used `.send(Env, query(canMake(Producer, Y)))` to find out which agent can make a certain item. While this is elegant, it's not entirely clear where the `Producer` variable gets bound (i.e. the semantics of the distributed/remote resolution semantics are not clear).

6. Conclusion and Future Work. Our approach combines declarative single agent specification with declarative concurrency between agents. This allows for truly concurrent, distributed systems specified in a declarative fashion. As far as we can tell, this is the first approach of its kind in the field. From the small examples we have provided it seems like an promising direction.

One of the main open issues at this stage is studying the formal underpinnings of such a system. How does the Join Calculus semantics work together with Jason's practical reasoning semantics. Intuitively there isn't much friction between the two, since each addresses a different concern. Still, representing beliefs as channels and goals as channels and predicates patterns, might prove problematic. In this case we can always fall back to the representation method used by Jason.

Some other, more distant, issues and open directions are:

- Adding a static type system to the language;
- Related to the previous point: using a statically typed *Knowledge Representation Language* (KRL) and studying ontology modeling by using features of the type system;
- Privacy and security issues—investigate the use of the *Principle of Least Privilege* (POLP) [13];
- Tooling for the environment (e.g. IDE, debugger, logger);
- Assessing the tool's use as a means of implementing *Ambient Intelligence* (AmI) scenarios.

This being said, we once again express our belief that this is a fresh, promising approach that, tackled properly, could produce valuable results for the agent community.

REFERENCES

[1] Costin Badica, Zoran Budimac, Hans-Dieter Burkhard, and Mirjana Ivanovic. Software agents: Languages, tools, platforms. *Comput. Sci. Inf. Syst.*, 8(2):255–298, 2011.

[2] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, September 2004.

[3] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pages 81–94, New York, NY, USA, 1990. ACM.

[4] Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[5] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog (4. ed.)*. Springer, 1994.

[6] Alvaro Fernandez Diaz, Clara Benac Earle, and Lars-Ake Fredlund. Erlang as an implementation platform for bdi languages. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, Erlang '12, pages 1–10, New York, NY, USA, 2012. ACM.

[7] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer, 2000.

[8] KoenV. Hindriks. Programming rational agents in goal. In Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H. Bordini, editors, *Multi-Agent Programming:*, pages 119–157. Springer US, 2009.

[9] Edmund S.L. Lam and Martin Sulzmann. Finally, a comparison between Constraint Handling Rules and join-calculus. pages 51–66.

[10] Louis Mandel and Luc Maranget. Programming in JoCaml — Extended Version. Research Report RR-6261, INRIA, 2007.

[11] Radek Píbil, Peter Novák, Cyril Brom, and Jakub Gemrot. Notes on pragmatic agent-programming with jason. In Louise Dennis, Olivier Boissier, and RafaelH. Bordini, editors, *Programming Multi-Agent Systems*, volume 7217 of *Lecture Notes in Computer Science*, pages 58–73. Springer Berlin Heidelberg, 2012.

[12] Alessandro Ricci and Andrea Santi. From actors to agent-oriented programming abstractions in simpal. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, SPLASH '12, pages 73–74, New York, NY, USA, 2012. ACM.

[13] Fred B. Schneider. Least privilege and more. *IEEE Security and Privacy*, 1(5):55–59, 2003.

[14] Yoav Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, March 1993.

Edited by: Dana Petcu
Received: Aug 30, 2013
Accepted: Oct 9, 2013