Task Suspension in Agent Systems

(Draft Paper)

Berndt Farwer

Department of Computer Science, Durham University, Durham, DH1 3LE, U.K.

Abstract

We discuss the similarity of a recent approach to task suspension in agent programming languages with an earlier approach to formalising preemption using a class of Petri nets, called M-nets. We argue that the theory of agent programming would benefit from adopting certain features of the Petri-net approach, and thus making further results for Petri nets applicable in the agent domain.

1 Introduction

Preemption is widely used in a number of contexts, originally in the area of multi-tasking in operating systems. Informally, it subsumes any kind of abortion or suspension of computational processes. Certain classes of Petri nets can be augmented with a subnet that will allow internal and external preemption operations to be carried out on the net [6]. This can then be used to define a compositional algebra of preemption with a Petri net semantics.

Recently, *suspension* has been used in conjunction with multi-agent systems [9] where conditions are investigated, in which a goal or a plan can be suspended and later on resumed. Several widely used agent programming languages support the concept of suspension, e.g. Jadex [7] and Jason [2] have internal mechanisms to suspend plans or goals during the deliberation process. The approach of [9] extends this in a way that gives the programmer some control over suspension and thus adds an active component to suspension in the agent's reasoning. This can be used to optimise the planning of individual agents. It is natural in other areas of computer science to speak of suspension and resumption of plans, e.g. for (concurrent) systems in which

Email address: `berndt.farwer@durham.ac.uk` (Berndt Farwer).

22 October 2008

the preconditions of sub-tasks may be violated after the main task has been started. The precise reasons can be manifold, e.g. lack of resources, change of conditions in the environment. The study of preemption with respect to Petri nets may help give insights into resource handling and deadlock avoidance for multi-agent systems.

An extension of an algebra of high-level Petri nets with operations for suspension and abortion is presented in [6,5]. We discuss this algebraic representation in the light of agent programming and propose an new sub-class of Petri nets as a semantics for agent programs that can actively suspend tasks.

2 Background

The presentation in [9] pursues an *ad-hoc* approach to extend an existing agent programming language (CAN) to be able to handle suspension of tasks by adding specific plans to the agents plan library. In this way, the original programming language is left unchanged and the approach can fairly easily be adapted to other languages than CAN. Existing programs are not affected and continue to work as before. The approach, however, lacks a formal foundation that would handle resources or enable reasoning about deadlocks.

To the author's knowledge there has not been any attempt to employ methods known from concurrency theory and algebra, e.g. preemption semantics based on classes of Petri nets, to a more general framework in computer science or to the area of agent programming in particular. This paper extends an algebraic approach to the theory of preemption and adapts it for use with agent programming. Future work will focus on a further classification of types of suspension and methods of resumption, based upon a theory of compositional programs supporting preemption in a resource-sensitive setting.

3 An Algebra for Preemption

The algebra discussed in this section is based on the use of M-nets as a semantics for asynchronous, concurrent systems. The main ideas behind the algebra presented below are:

- A priority relation between actions or subgoals ensures that the appropriate steps needed to suspend a task are taken in a way that does not interfere with the remainder of the concurrent program in an undesired way.
- The concurrency of tasks (intentions) corresponds to concurrency of Petri net transitions in the semantics.

- An algebra of agent programs with priority (APP) corresponds to the algebra of priority M-nets, resp. its extension, preemptible M-nets (P/M-nets).
- The composition of agent programs in APP corresponds to composition of M-nets.
- Top-down refinement in APP corresponds to refinement of M-nets.
- Subgoals in the agent program can be represented by refinement in the corresponding M-net.
- The operation $\pi(\cdot)$ for preemption on M-nets has different orientations:
  - · internal vs. external preemption, representing internal suspension of intentions by the reasoner (i.e., the agent programming language interpreter) vs. programmed suspension (i.e., the agent's own behaviour represented explicitly in its plans);
  - · preemption is either *abortion* or *suspension*.
- The role of time in preemption is important: In an asynchronous system we have to ensure that after the start of suspension no action of the suspended intention will occur until the resumption of the intention (or task/goal).

The most important operations of the proposed algebra are

- sequential composition $(a; b)$;
- parallel composition $(a \parallel b)$;
- choice $(a \,\Box\, b)$;
- "if not a then b" $(a \rhd b)$.

Further operations include iteration, synchronisation, and renaming.

The syntax is given by

The following sections introduce the concepts of M-nets (Section 3.1) and the agent programming language CAN (Section 4).

3.1 M-nets

M-nets[1] are a class of compositional high-level Petri nets with typed places and guarded transitions. The net components carry label annotations with the respective type information. M-Nets form a well-studied category with nice compositional features. They have been used as a semantics of the concurrent $B(PN)^2$ programming language, including constructs, such as parallel composition, iteration, guarded commands, handshake communications, communication though buffered channels, and shared variables [1]. A variant of M-nets, called priority M-nets, has been introduced to add features needed for the introduction of a preemption operator.

[1] Modular multi-labelled nets

Let Tok be an enumerable set of types (or colours) and let Var be an enumerable set of variable names with $Var \cap Tok = \emptyset$.

Definition 1 *An M-net is a tuple $N = \langle P, T, W, \iota \rangle$, with $P \cap T = \emptyset$, $W \in P \times Var \times T \ \cup \ T \times Var \times P^{\oplus}$, and $\iota : P \cup T \to Tok \cup Var$, where $\iota(p) \subseteq Tok$ is the* type *of place $p \in P$ and $\iota(t)$ is a boolean expression over $Tok \cup Var$, the* guard *for transition $t \in T$.*

Ensuring that we deal only with finite types, will yield a model that is strictly less powerful than Turing machines. Priority M-nets add a pairwise priority relation $\prec \subseteq T \times T$ on transitions to M-nets. $t_1 \prec t_2$ means that t_1 has lower priority that t_2, hence the execution of t_2 would be preferred over the execution of t_1. Note that $\prec$ is not assumed to be transitive.

An M-net

- having at least one entry place (that has only outgoing arcs) and one exit place (that has only incoming arcs);
- whose entry and exit places have a singleton type ($\{\bullet\}$);
- having only transitions with at least one input place and one output place

is called *ex-good* . The class of ex-good M-nets forms an algebra [1] with the following operations:

- Refinement, $N[\mathcal{X} \leftarrow N']$
- parallel composition, $N \parallel N'$, defined as $1N \cup 2N'$, i.e., disjoint juxtaposition
- sequential composition, $N; N'$, defined as $(1N \cup 2N') \oplus \otimes\{(1N)^\bullet, {}^\bullet(2N')\}$
- choice, $N \Box N'$, defined as $(1N \cup 2N') \oplus \otimes\{{}^\bullet(1N), {}^\bullet(2N')\} \oplus \otimes\{(1N)^\bullet, (2N')^\bullet\}$
- iteration, $[N * N' * N'']$, i.e., one execution of N, followed by an arbitrary number (possibly 0) of executions of N' and finally one execution of N''
- renaming, $N[f]$
- synchronisation, $N \,\mathbf{sy}\, A$, adding a new transition for pairs of transitions in N labeled with the appropriate synchronisation symbols (A and $\bar{A}$)
- restriction, $N \,\mathbf{rs}\, A$, i.e. removal of all transitions with action label A or $\bar{A}$
- scoping, $[\![A : N]\!]$, essentially synchronisation followed by restriction
- asynchronous link, $N \,\mathbf{tie}\, b$

In the above, N, N', N'' are M-nets, A is a synchronous communication symbol, b is an asynchronous link symbol, f is a renaming function defined on synchronous communication symbols an asynchronous link symbols, and $\mathcal{X}$ is a hierarchical symbol. For details of the definitions and properties of these operators, see [1] and [4].

Klaudel and Pommereau [5,6] extend M-nets with a priority relation over transitions, so that they can use this new class of M-nets as a semantics for preemption.

Definition 2 *Let $N = \langle P, T, W, \iota \rangle$ be an M-net and let $\prec \subset T \times T$ be a pairwise priority relation, then $N^{\prec} = \langle N, \prec \rangle$ defines a* priority M-net.

The operations on M-nets are straightforwardly extended to priority M-nets. An additional preemption operation $\pi()$ is introduced in [5], so that $\pi(()N^{\prec})$ is the priority M-net $N^{\prec}$ extended with a sub-net that handles preemption. This leads to the definition of *preemptible M-nets*:

Definition 3 *A priority M-net $N^{\prec}$ is called* P/M-net *if either N is an* ex-*good M-net and $\prec = \emptyset$ or N is formed iteratively using the operations preemption, refinement, parallel composition, sequential composition, choice, iteration, synchronisation, restriction, scoping, asynchronous linking, or renaming on priority M-nets.*

The augmented net can now be subject to the usual Petri net analysis and established techniques, for instance for deadlock avoidance, can be employed.

4 CAN

CAN is a high-level agent programming language. It is similar to AgentSpeak [8,3] and similar BDI agent systems [2]. CAN's explicit goal construct captures both the declarative and procedural aspects of a goal. In CAN, when a plan fails, another applicable plan (if any) is automatically attempted. This equates to the default failure handling mechanism typically found in most BDI language interpreters.

In CAN, plans are written $e : c \leftarrow P$, where e is an *event*, c is a *context condition*, and P is the *program* or *plan body*. The plan bodies of CAN programs take the following form:

$$\begin{aligned} P := a \mid +b \mid -b \mid ?\,\phi \mid !\,e \mid P; P \mid P \parallel P \mid \\ \text{Goal}(\phi, P, \phi) \mid P \rhd P \mid (\!|\{\psi : P, \ldots, \psi : P\}|\!) \mid \text{nil} \end{aligned}$$

Here, a is an action, b is a belief, e is an event, and ψ and ϕ are logical formulae over agent beliefs. $\psi : P$ represents a guarded plan, and $(\!|\{\psi_1 : P_1, \ldots, \psi_n : P_n\}|\!)$ denotes a set of such plans from which the agent can choose. $P_1 \rhd P_2$ will try to execute P_1 and only start executing P_2 if P_1 fails.

In [9], Thangarajah *et al* define a transition system whose states (*basic agent configurations*) are $\langle B, P \rangle$, consisting of a belief base B and a current intention

[2] BDI stands for the *belief, desire, intention* paradigm

P. Methods, i.e. plans, for suspending and resuming goals are then added to an existing program. These methods are given in the CAN syntax and represent regular plan bodies. The semantics, however, gives way to very inefficient processing of these methods, so that an implementation closer to the M-net semantics is desirable.

Proposition 4 *The operational semantics of CAN can be faithfully modeled by a subclass of M-nets, giving rise to a more efficient implementation and to compositionality.*

Task instances are introduced and the agent program is augmented in [9] with a set of rules to handle suspension. Furthermore, each plan has to be transformed so that it becomes aware of the additional information, i.e. beliefs, that is introduced to initiate the suspension of a task.

Section 5 explains the idea for a Petri-net-based semantics for CAN.

5 M-nets as CAN Semantics

We modify the standard operations on M-nets and priority M-nets to accommodate agent specific needs and add some new operations. A summary of the new and modified operations is given below. We introduce the class of agent M-nets (a-M-nets) based on ex-good priority M-nets: Each a-M-Net has a single token in a distinguished *start place* marked *entry*. It furthermore has two distinguished exit places, marked *exit* and *fail*.[3] It is assumed that once a token reaches one of the exit places no other tokens remain in the net. This property is easily shown for the simplest kind of a-M-net, the action-M-net, shown in Figure 1. This represents an action, as found in agent plans, and has the possibilities of successfully executing the action and failing. By induction, the property also holds for composed a-M-nets.

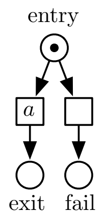

Fig. 1. A simple action-M-net

[3] a-M-nets are special cases of ex-good M-nets with just one entry place and exactly two (distinguishable) exit places. It is easy to see that every ex-good M-net can be transformed into an equivalent ex-good M-net with just one entry place.

Figures 2 to 4 schematically show some operations on a-M-nets and a reduction to a simpler equivalent net, were possible. To simplify the illustrations, arc, place, and transition inscriptions have been omitted. The reduction works by removing $p \in P$ and $t \in T$ from the net together with their arcs, whenever $p^\bullet = \{t\}$ and there is a $p' \in P$, such that $t^\bullet = \{p'\}$. A new arc (t', p) is introduced for each $t' \in {}^\bullet p$.

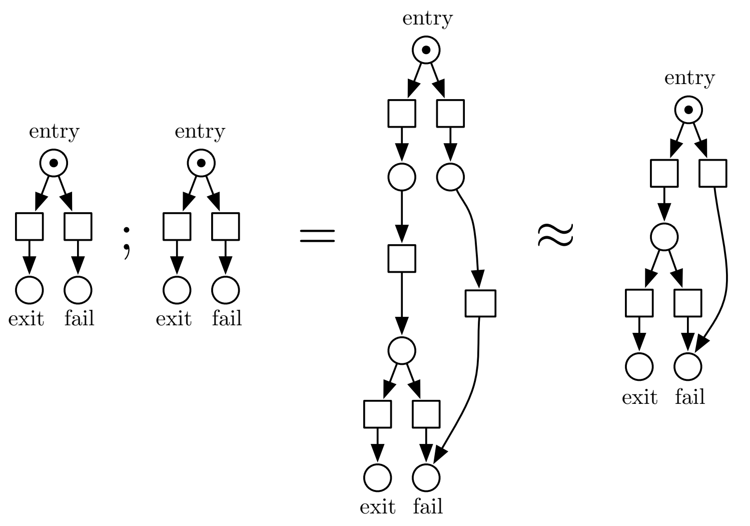

Fig. 2. Operation ; on action-M-nets

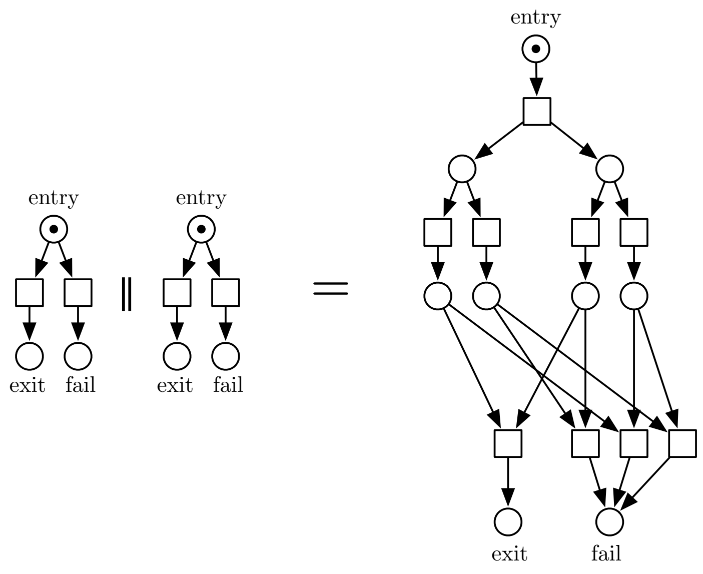

Fig. 3. Operation ∥ on action-M-nets

Other CAN operations can be modelled with similar ease. Embedding M-nets within an object-net formalism would naturally extend the semantics with

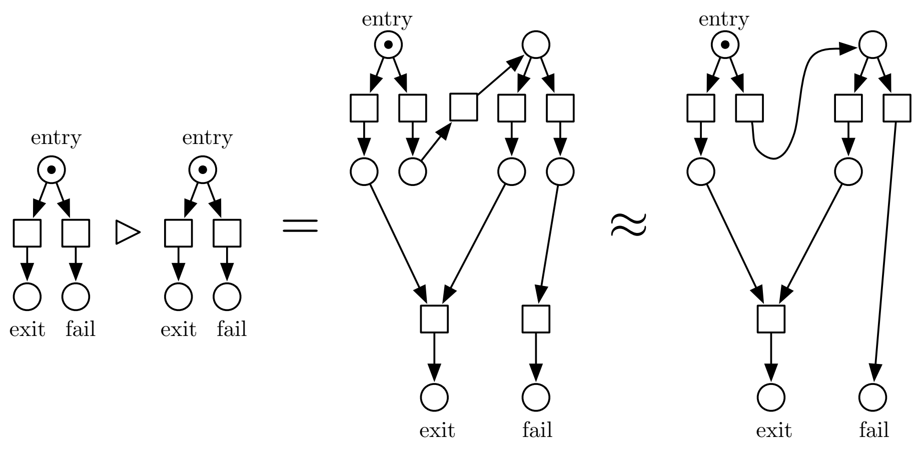

Fig. 4. Operation ▷ on action-M-nets

notions of environment and mobility. This will give rise to a semantics that can inherently deal with the important issues in agent programming, namely concurrency, resources, and location.

6 Conclusion

In this paper we have used a class of Petri nets to give a true concurrent semantics to preemtible/suspendible systems. Since Petri nets are inherently asynchronous, this semantics is very well suited in the area of agent systems, where agents are assumed to act autonomously on a distributed system and communicate asynchronously over this distributed network.

The discussion in this paper has remainder largely informal, so the next major step towards a theory of suspension for agent programming is to formalise the ideas further and establish the formal system as a foundation for the extension of well-known agent programming languages. Using our theory, based on M-nets, as a foundation would instantly lead to compositionality, which is a prerequisite for successfully using suspension methods for multi-agent reasoning.

Petri nets as semantics have much to offer for agent programming, especially when it comes to plans involving resources. It can be important for resources to be released for use by other plans in the case of goal suspension. On the other hand, this may lead to yet further deadlock situations. It is clearly necessary for agent programming to study the consequences of dealing with resources in different ways. Results on Petri nets (including strategies for deadlock avoidance) can help in gaining a better understanding of how to deal with these issues in agent reasoning with bounded resources.

Future work will be carried out on formalising the presentation given in this extended abstract, in particular with respect to the use of resources in the suspension of agent programs.

References

[1] Eike Best, Wojciech Fraczak, Richard P. Hopkins, and Elisabeth Pelz. M-nets: An algebra of high-level petri nets, with an application to the semantics of concurrent programming languages. *Acta Informatica*, 35(10):813–857, 1998.

[2] Rafael H. Bordini and Jomi F. Hübner. *Jason: A Java-based interperter for an extended version of AgentSpeak*, 2006. Available from http://jason.sourceforge.net.

[3] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason.* Wiley, 2007. ISBN: 978-0-470-02900-8.

[4] Raymond Devillers, Hanna Klaudel, and Robert-C. Riemann. General parameterised refinement and recursion for the m-net calculus. *Theoretical Computer Science*, 300(1-3):259–300, 2003.

[5] Hanna Klaudel and Franck Pommereau. A concurrent and compositional petri net semantics of preemption. In *Integrated Formal Methods: Second International Conference, IFM 2000, Dagstuhl Castle, Germany, November 2000. Proceedings*, volume 1945 of *Lecture Notes in Computer Science*, pages 318–337. Springer-Verlag, 2000.

[6] Hanna Klaudel and Franck Pommereau. A class of composable and preemptible high-level petri nets with an application to multi-tasking systems. *Fundamenta Informaticae*, 50(1):33–55, 2002.

[7] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A BDI reasoning engine. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications.* Springer, 2005.

[8] Anand S. 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.

[9] John Thangarajah, James Harland, David Morley, and Neil Yorke-Smith. Suspending and resuming tasks in BDI agents. In *Proceedings of AAMAS'08*, pages 405–412, 2008.