The Isomorphism Between a Class of Place Transition Nets and a Multi-Plane State Machine Agent Model

Krzysztof Palacz, Ladislau Bölöni, and Dan C. Marinescu
Computer Sciences Department
Purdue University,
West Lafayette, IN, 47907
email: {palacz,boloni, dcm}@cs.purdue.edu

November 19, 1999

Abstract

Recently we introduced a multi-plane state machine model of an agent, released an implementation of the model, and designed several applications of the agent framework. In this paper we address the translation from the Petri net language to the Blueprint language used for agent description as well as the translation from Blueprint to Petri Nets. The simulation of a class of Place Transition Nets is part of an effort to create an agent–based workflow management system.

Contents

1 Introduction

Recently, we proposed a multi – plane state machine agent model [2] and released a component – based architecture for building agents. Bond agents were used to construct a network of PDE solvers [16], to support adaptive resource allocation for multimedia applications [13], to discover resources in a wide-area distributed object system [12] and for several other projects including the workflow management system discussed briefly in Section 5 of this paper.

Our agent design has unique features consequences of our philosophy to represent the behavior of the agent as a *multi-plane state machine* as discussed in Section 2.

- Behavior embedded into a data structure. The overall behavior of the agent is described by the blueprint. The strategy associated with a state defines the behavior in that state.
- Explicit concurrency. The agent structure is based on several state machines running concurrently.
- Dynamic agent modification, agent surgery.
- Integration of heterogeneous behavior models.
- Possibility of automated programming. While the automated programming at the fine grain level is still unrealistic at the current level of knowledge, the structural components of the model are coarse grain enough and their semantics is simple enough that we can envision that they can be created automatically based on the needs and circumstances.

A model for specifying behavior frequently used in embedded systems and UML [14] based systems is the statechart model [11] designed by David Harel. Our multiplane agent state machine can be seen as a different way of expressing the parallelism which in statecharts are expressed as *concurrent substates*. On the other hand we have chosen to use a simpler state machine than those used in state charts. For example, in statecharts transitions can have conditions and actions associated with them, while in our model, only states can generate actions and transitions are unconditional. The theoretical foundation behind this decision is that in our model the structural component (the multiplane state machine), the active components (the strategies) and the model of the world are clearly separated. If we introduced a condition on a transition, that would clearly be a boolean function on the model, thus making the state machine dependent on the model. If an action were associated with the transition that would either imply that actions can be generated outside strategies, or alternatively that there is a strategy which is not determined by the state vector. Both these semantics are expressed in our model by *inserting an intermediate node* between the source and destination, the strategy of these node than performing the desired action or evaluating the condition. Thus, the multiplane state machines in our model can be larger for the same task than the corresponding statechart, but they are easier to analyze and generate, because of the simpler semantics. On the other hand, real time systems are easier to specify in the statechart format. Another feature of statecharts, the possibility to define embedded substates is also missing in our system. Its functionality in most cases can be replaced by the state vector of the multiplane state machine. We are currently investigating the possible benefits of introducing substates in our model: although they improve the expressiveness of the system, they also introduce difficulties in implementing checkpointing, migration and agent surgery.

We adhere to the definition of Stan Franklin and Art Graesser [9]: *an* **autonomous agent** *is a system situated within and part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.* The **agent execution model** assumes that the agent has an explicit goal. Agents receive external events and generate actions. However the actions of an agent are determined by the pursue of its goal. Of course, events may trigger immediate responses, but agents perform actions even without any external input. An agent terminates its execution when its goal is accomplished.

The other execution models are: the **non-interactive (batch) execution model**, and the **reactive application execution model**, see Figure 1. The first is characteristic for traditional numerical simulation. The goal is to generate output results given a set of input data. All the inputs are available at the start of the application. The execution does not involve any interaction between the user and the environment from start to finish. In the reactive execution model, applications provide immediate responses to user inputs or external events. The application does not have an explicit goal, never takes the initiative and its termination is also a response to an external event. In absence of external events, the application is in an *idle cycle*. Examples are interactive programs like word processors, operating system shells, traditional servers e.g. file servers or http servers, embedded systems.

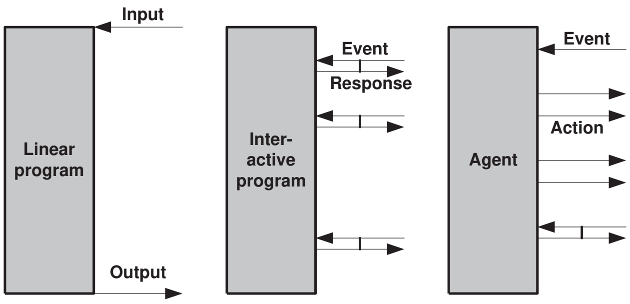

Figure 1: The execution model for non-interactive, reactive applications and agents. The first are characterized by a continuous progression from the input to the generation of output. Reactive applications are characterized by the well defined event-response pairs. The behavior of agents is characterized by the spontaneous generation of actions, although event-response pairs may also exist.

The execution model for agents is the more general and complex of the three models. Combinations of execution models are also possible, e.g. some modern servers like those used in airline ticket reservation or e-commerce perform a number of actions between the original request and the reply. These servers can be viewed as agents.

Since an agent can emulate every other execution model, some researchers are inclined to view every program as an agent. Although this approach may be sometimes useful, it does not lend itself to efficient implementations, the simpler the execution model the more optimal implementation is possible. For example we can optimize the response time of a stateless server far better than of an agent with a complex state.

This paper is organized as follows. In Section 2 we outline the agent model. In Sections 3 and 4 we discuss the translation to/from Petri Nets from/to Blueprint, then in Section 5 we outline the architecture of an agent-based workflow enactment engine.

2 A Multi-Plane State Machine Agent Model

Bond is a Java based distributed object system and agent framework. It is composed of a core level, containing the object model and message oriented middleware, a service layer containing distributed services like directory and persistent storage services and the agent framework, providing the basic tools for creating autonomous network agents together with a database of commonly used strategies which allow developers to assemble agents with no or minimal amount of programming.

2.1 Bond Core

At the heart of the Bond system there is a Java Bean-compatible component architecture. Bond objects extend Java Beans by allowing users to attach new properties to the object during run-time, and offer a uniform API for accessing regular fields, dynamic properties and JavaBeans style `setField/getField`-defined virtual fields. This allows programmers the same flexibility like languages like Lisp or Scheme, while maintaining the familiar Java programming syntax.

Bond objects are network objects by default: they can be both senders and receivers of messages. No post-processing of the object code as in RMI or CORBA-like stub generation, is needed. Bond uses *message passing* while RMI or CORBA-based component architectures use *remote method invocation.*

The system is largely independent from the message transport mechanism and several communication engines can be used interchangeably. We currently provide TCP-based, UDP-based, Infospheres-based, and, separately, a multicast engine. Other communication engines will be implemented as needed. The API of the communication engine allows Bond objects to use any communication engines without the need to change or recompile the code. On the other hand, the properties of the communication engine are reflected in the properties of the implemented application as a whole. For example the UDP based engine offers higher performance but does not guarantee reliable delivery.

All Bond objects communicate using an agent communication language, KQML [8]. Bond defines the concept of *subprotocols*, highly specialized, closed set of commands. Subprotocols generally contain the messages needed to perform a specific task. Examples of generic Bond subprotocols are *property access* subprotocol, *agent control* subprotocol or *security* subprotocol. An alternative formulation would be that subprotocols introduce a *structure in the semantic space of the messages.*

Subprotocols group the same functionality of messages which in a remote method invocation system would be grouped in an *interface.* But the larger flexibility of the messaging system allows for several new techniques which are difficult to implement in the remote method call case:

- The subprotocols implemented by objects are properties of the object, so two objects can use the property access subprotocol implemented by every Bond object, to find the common set of subprotocols they can use to communicate.
- An object is able to control the path of a message and to delegate the processing of the message to subcomponents called *regular probes.* Regular probes can be attached dynamically to an object as needed.
- Messages can be intercepted before they are delivered to the object, thus providing a convenient way to implement security by means of a fire wall, accounting, logging, monitoring, filtering or preprocessing messages. These operations are performed by subcomponents called *preemptive probes* which are activated before the object in the message delivery chain.
- Subprotocols, like interfaces, are grouping some functionality of the object, which may or may not be used during its lifetime. A subcomponent called *autoprobe* allows the object to instantiate a new probe, to handle an incoming message which can not be understood by the existing subcomponents attached to an object.
- Objects can be addressed by their unique identifier, or by their *alias.* Aliases specify the services provided by the object or its probes. An object can have multiple aliases and multiple objects can be registered under the same alias. The latter enables the architecture to support *load balancing* services.

These techniques can be implemented through different means in languages which treat methods as messages, e.g. Smalltalk. In Java and C++ they can be implemented at compile time, not at runtime, e.g. using the delegation design pattern. Techniques from the recent CORBA specifications e.g. the simultaneous use of DII, POA, trading service and others, also allow to implement a similar functionality, but with a larger overhead, and significantly more complex code.

2.2 Bond Services

Bond provides a number of services commonly used found in distributed object systems, like directory, persistent storage, monitoring and security. Event, notification, and messaging services, which provide message passing services in remote method invocation based systems are not needed in Bond, due to the message-oriented architecture of the system.

Some of Bond services perform differently than their counterparts in other middleware systems, like CORBA. For example, Bond never requires explicit registration of a new object with a service. Finding out the properties of a remote object, i.e. the set of subprotocols implemented by the object, is done by direct negotiation amongst the objects. The directory service in Bond combines the functionality of the naming and trading services of other systems and it is implemented in a distributed fashion. Objects are located by a search process which propagates from local directory to local directory. The directories are linked into a virtual network by a transparent *distributed awareness* mechanism, which transfers directory information by piggybacking on existing messages as discussed in the previous Section.

Compared with the naming service implementations in systems like CORBA or RMI, which are based on the existence of a name server, this approach has the advantage that there is no single point of failure, and the distributed awareness mechanism reconstitutes the network of directories even after catastrophic failures. However, a distributed search can be slower than lookup on a server, especially for large networks of Bond programs. For these cases, Bond objects can be registered to external directories, either to a CORBA naming service through a gateway object, or to external directory services using LDAP access.

2.3 Bond Agents

The *Bond agent framework* is an application of the facilities provided by the Bond core layer to implement collaborative network agents. Agents are assembled dynamically from components in a structure described by a multi-plane state machine, [2]. This structure is described by a specialized language called `blueprint`. The active components (*strategies*) are loaded locally or remotely, or can be specified in interpretive programming languages embedded in the blueprint script. The state information and knowledge base of the agents are collected in a single object called *model of the world* which allows for easy checkpointing and migration of agents. The multiplane state machine describing the behavior of agents can be modified dynamically, thus allowing for *agent surgery*.

The *behavior* of the agent is described by the *actions* the agent is performing. The actions are performed by the strategies either as reactions to external events, or autonomously in order to pursue the *agenda* of the agent. The current state of the multiplane state machine (described by a *state vector*) is specifying the strategies active at a certain moment. The multiple planes are a way of expressing parallelism in Bond agents. A good technique is to use them to express the various facets of the agents behavior: sensing, reasoning, communication/negotiation, acting upon the environment and so on. The *transitions* in the agent are modifying the behavior of the agent by changing the current set of active strategies. The transitions can be triggered by internal events or from external messages - these external messages form the *control subprotocol* of the agent.

Strategies, having limited interface requirements are a good way to provide code reuse. The Bond agent framework provides a strategy database, for the most commonly used tasks, like starting and controlling external agents or legacy applications. A number of base strategies for common tasks like dialog boxes or message handlers are also provided, which can be sub-classed by developers to implement specific functionality. External algorithms, especially if written in Java are usually easily portable to the strategy interface.

2.4 Using Planes to Implement Facets of Behavior

The behavior of an agent is often *multifaceted*, it consists of several loosely coupled aspects. A full-featured agent may exhibit several facets:

- **Reasoning.** Agents with reasoning abilities usually run in a cycle, generating new true statements based upon current knowledge.
- **Visual interface.** Most agents present a visual interface and interact to the user by: (a) presenting a part of the knowledge of the agent (i.e. a part of the model) in a visual format, and (b) collecting user interface events from the user.
- **Reactive behavior.** Agents react to external events e.g. messages, user interface events, signals, etc.
- **Active behavior.** Agents perform actions in pursuit of their agenda even without external events.

In most cases, a separation of these facets is possible, and the relative independence of the facets justifies their separate treatment. For example the various steps taken by an agent to pursue its goal are changes in its active behavior, but these changes may not necessarily lead to a change in its reactive behavior, the look of the user interface, or the reasoning process of the agent.

We argue that the multi-plane state machine structure provides an elegant way to express the multifaceted behavior of an agent, every plane expresses a facet of the behavior of the agent. There are no restrictions on the nature and behavior of planes, so the agent designer can create the structure most suitable to the problem at hand. However, the independence of facets is relative, significant interdependence existing between them. In the multi-plane state machine structure, the interdependence amongst planes is captured by the fact that (a) all planes share a common model (knowledge) and (b) transitions triggered by one plane are applied to the whole structure, providing a signaling mechanism amongst planes.

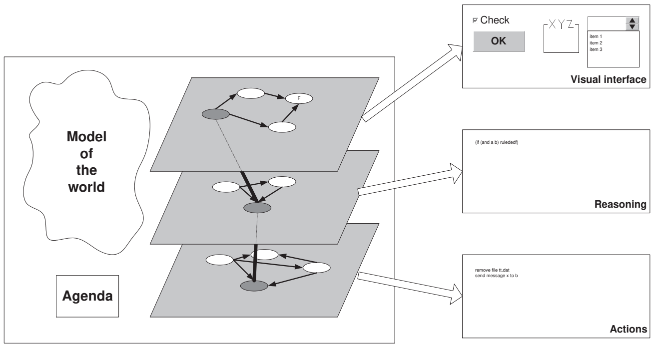

Figure 2: Different plans of the agent represent different facets of the behavior of the agent. In this case we have a plane controlling the visual interface, a plane for reasoning and a plane for performing actions.

Figure 2 represents an example of an agent that presents a visual interface to the user, performs a background reasoning, and takes actions. These three facets of the behavior are reflected into an agent with three planes, each one dedicated to a facet.

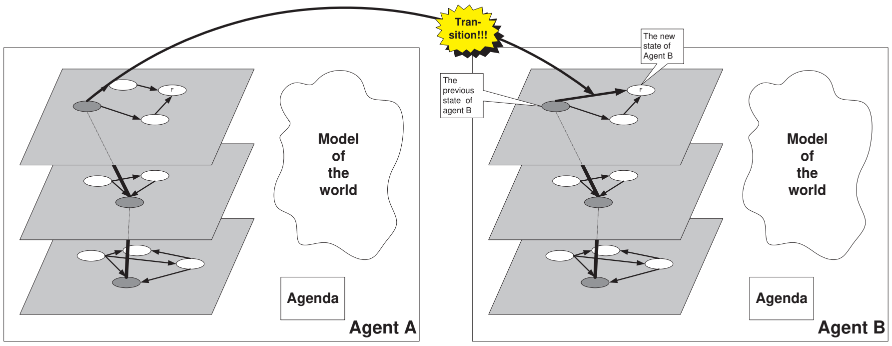

Figure 3: Example of remote control using messaging. The message sent by agent A triggers a transition on the first plane of agent B thus changing its behavior.

A significant part of the inter-agent communication can be described as *control*: the behavior of the *controlled* agent is changed as a result of an action of a *controller* agent.

In terms of our structure, the behavior of the agent is described by the state vector, and it can be changed by transitions, which alter one or more states in the state vector. One way to trigger transitions is by an external message.

In our implementation of the model, transitions can be triggered by KQML messages. Messages are grouped in *subprotocols* which is determined by the multi-plane state machine of the agent, which in its turn is described by the blueprint. For consistency reasons not all transitions in the multi-plane state machine can be triggered remotely, because the semantics of certain transitions, for example `success` depends upon the current strategy.

Figure 2.4 present an example of use. Agent A wants to change the behavior of agent B, by changing the strategy on the first plane. Thus it sends a message labeled with the given transition. The transition will be performed on all the planes of agent B, if there is a transition with the specific label. In our case, only on the first plane it is a transition with the label as specified, thus the only state from the state vector which is changed is the state of the first plane.

The use of messages as information control is used in most multi-agent implementations using Bond. One example is the network of agents for solving partial differential equations [16] where external transitions are used to synchronize the results between independent solver agents.

Another important aspect of agents is that they can cooperate towards achieving certain goals. We argue that every agent cooperation can be described in terms of knowledge sharing. In our structure this means that a part of the model of an agent is copied to the model of another agent.

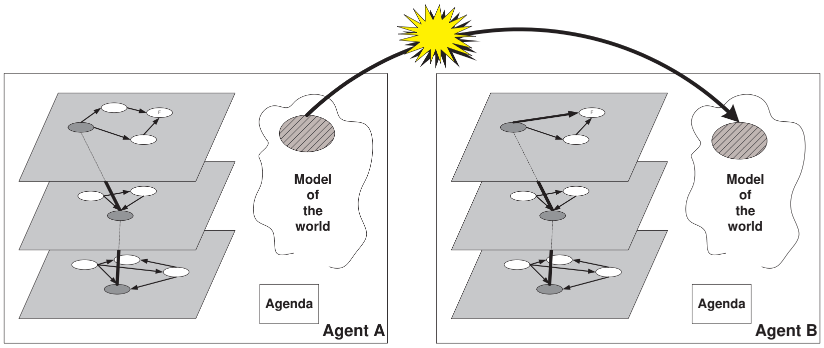

Figure 4: Example of inter-agent cooperation using knowledge sharing. Agent A pushes part of its model into the model of agent B.

There are basically two ways of sharing knowledge, depending upon the agent initiating of the process:

- **push mode** an agent copies part of its model to the model of the other agent.
- **pull mode** an agent copies a part of the model of the remote agent into its own model.

Security and semantic concerns related to the information sharing models must be addressed. We have to determine what part of the model will be shared, the identities of the agents, the confidence level in the shared knowledge and so on.

Our implementation, the Bond system contains support for information sharing at the communication layer level, and contains various mechanisms enforcing security for inter-agent cooperation [10]. Figure 2.4 presents an example of cooperation through knowledge sharing using the push mode. Agent A is pushing a part of its model to the model of agent B.

Although the knowledge sharing model applies to any possible cooperation model, it may not be the best way of dealing with some cooperation patterns. For example, a service negotiation would be implemented as a successive series of very small information interchanges. Obviously, this case is better handled if we view the negotiation as a message interchange and apply the results of recent research in this area . In our model we may create several negotiation planes such that multiple negotiations may be conducted simultaneously by the agent.

Cooperation through knowledge sharing is normally used when an agent starts a new agent and passes part of its knowledge to the newly created agent. One example is the PDE solver agent system mentioned before where partial results of computations are communicated in this way.

3 Simulating a Class of Place-Transition Nets on the Bond Multi-Plane Finite State Machine

The most straightforward way of simulating execution of a Petri net using a Bond agent involves allocating a separate plane for each transition in the net. This corresponds to the fact that each transition can fire whenever it is enabled, independently of others, thus requires its own "thread of control" and in Bond a "thread of control" is assigned to each plane (whether or not it corresponds to an actual Java thread). Hovewer, this approach is disadvantageous because the state transition mechanism present in Bond cannot be used explicitly. Also the actual degree of parallelism in the simulated Petri net is in most cases much smaller than the number of transitions present in it. We present here an alternative approach that is applicable to a large class of Petri nets. It is not our goal to build a general-purpose simulation tool but rather a framework for exection and coordination of software components, however, it is convenient to think about this framework as a Petri net simulation environment.

Let us first consider Petri nets with lowest possible degree of parallelism: the *S-nets*. In these nets $|\bullet t| = 1 = |t \bullet|$ for every transition t, i. e. each transition has exactly one incoming arc and outgoing arc. If the initial marking of an S-net has exactly one token and the net is strongly connected then the net can contain only one token at any time (thus it immediately follows that it is safe). In fact the set of all possible firing sequences of an S-net with the only token initially in place p_0 is equal to the language accepted by a finite state automaton $(S, A, p_0, \{p \in S :| p\bullet |= 0\})$ where the mapping A is defined as

$$A : S \times T \mapsto S,\ A(p,t) = q,\ (p,t),(t,q) \in F.$$

In effect it can be simulated by a single-plane Bond finite state machine containing states corresponding to Petri net places (we shall refer to them as p-states) as well as transitions (further called t-states) and transitions corresponding to Petri net arcs.

If a Petri net is not an S-net but can be partitioned in several disjoint S-nets, each of the S-nets can be simulated on a separate Bond plane. For connected nets such partition is not possible, however, we can attempt to *cover* the original net with S-nets. It is convenient to introduce here the following definitions and a theorem quoted from [7].

Subnets. Let (S, T, F) be a Petri net and $X \subseteq S \cup T$. The triple $(S \cap X, T \cap X, F \cup (X \times X))$ is the *subnet* of N generated by X.

S-components. Let N' be the subnet of a net N generated by a nonempty set X of nodes. N' is an *S-component* if

- $\bullet s \cup s\bullet \subseteq X$ for every place s of N
- N' is a strongly connected S-net.

S-covers. Let C be a set of S-components of a net. C is an *S-cover* if every place in the net belongs to an S-component of C. A net is covered by S-components if it has an *S-cover*.

Theorem Free choice, bounded and safe nets have S-covers.

Thus the set of S-coverable nets contains an important class of Petri nets possessing properties that are desirable in distributed and concurrent systems design. There exist efficient algorithms for finding S-covers; the algorithm used in Bond is taken from [6] and runs in $O((|S||T|)^2)$.

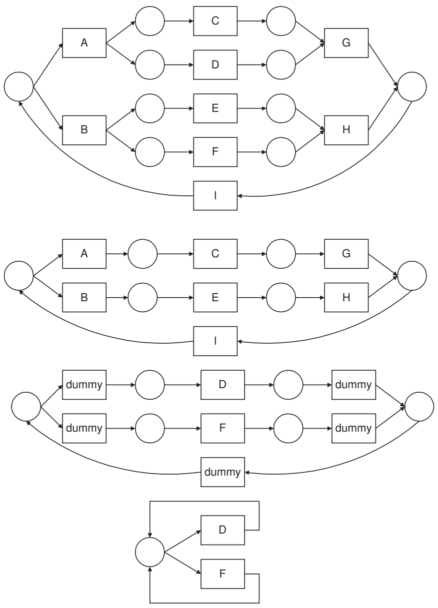

Figure 5: Example of the S-decomposition procedure. The top diagram represents the original net, the following two depict the two S-components with "dummy" transitions selected. The bottom diagram represents the second S-component after the dummy transitions have been contracted.

Thus in order to simulate a S-coverable Petri net we introduce a separate Bond plane for each of the S-components obtained from the decomposition procedure as in case of an S-net. However, since the decomposition procedure does not produce disjoint components, several Bond states may correspond to one node in the original Petri net, in particular several t-states may correspond to one transition which may have an action associated with it, which is clearly undesirable. To alleviate this problem we simply label all but one of the t-states corresponding to one Petri net transition (which in turn corresponds to one activity) as "dummy" and associate with them a strategy that once started immediately terminates reporting success.

In order to be able to simulate the Petri nets on the Bond multi-plane state machine we also have to add synchronization functionality to the strategies assigned to p-states. The decision when to fire a Petri net transition has to be made before the corresponding t-state is entered thus it is the role of the strategy associated with a preceding p-state to perform that decision. Even though in general a transition can have many preceding places, each transition has just one preceding place in an S-net, and although a transition can belong to many S-components, it will be marked as "dummy" in all but one of them, thus the p-state in charge of synchronization can be determined unambiguously. The strategy for this state has to wait until all the other p-states corresponding to the places preceding the given transition are entered and succeed. Additionally it can wait until a trigger condition necessary for firing of the transition is satisfied.

The "dummy" transitions can eventually be removed without changing the behavior of the system. For each of these transitions its following place q is merged with its preceding place p. Each strategy waiting until q is entered has to be modified to wait until p is entered. This process can be iterated until no further improvements can be achieved (see Fig. 5 for an example).

4 Mapping Bond Blueprints to Petri nets

Performing dynamic modifications of agent definitions calls for "sanity-checking" of the modified definitions. Since there exist well 'established Petri net analysis methods it is desirable to develop techniques of translating blueprints into a Petri net representation. In the absence of coordination among planes this translation is trivial since each plane is an independent finite state machine (for each state S add transitions τ, f_τ, places in_τ and out_τ and arcs $(in_\tau, \tau), (\tau, out_\tau), (out_\tau, f_\tau), (f_\tau, out_\tau)$, for each transition from state S_i to S_j add a transition τ_{ij} and arcs $(out_i, \tau_{ij}), (\tau_{ij}, in_j)$ as well as a transition from each out_τ place to a special *softstop* place. The existence of the f_τ transitions accounts for the fact that an action can be executed many times in a given state before the strategy supplies the name of transition to perform or an external message is received).

If synchronization among planes does take place it is difficult and impractical to attempt a translation to the Petri net domain since the strategies can execute arbitrary Java code and thus might employ any *ad hoc* synchronization schemes e. g. by inspecting shared variables etc. However, it is possible to perform such a translation if synchronization is introduced in a controlled fashion by providing necessary primitives and isolating the locations where these primitives can be invoked (note that since all strategies have access to the same shared memory (the model), this approach does not prevent the strategies from synchronizing using other methods either intentionally or unintentionally, but merely encourages programming discipline). In order to achieve this we shall require that all blueprints be in the normalized form that is all the states on each plane can be divided into t-states with one incoming and outgoing transition and arbitrary strategies associated and p-states with arbitrary number of incoming and outgoing transitions but restricted strategies. These strategies can only evaluate expressions containing model variables never changed by strategies on other planes and perform a transition based on that evaluation. Note that the translation to Petri Nets presented in the previous paragraph can be used as a base for a blueprint functionally equivalent to the original blueprint if the transitions f_τ as well as all the transitions to the *softstop* place are neglected, the strategies for each state S are rewritten to result in `success` and to write the actual Bond transition to be performed in a designated model variable and subsequently associated with Bond states representing transitions τ, dummy strategies are associated with states representing places τ_{in} and transitions τ_{ij} and special strategies are associated with places τ_{out}, whose only function is to perform a transition based on the value of the designated model variable for the preceding state.

The only synchronization construct that we make available is the `waitfor(p)` primitive, whose function is to block the execution of the plane in the context of which it is being invoked until the plane containing the p-state p enters this state. This primitive is allowed to appear only in p-state strategies. This condition is easily expressible in terms of Petri nets: if `waitfor(p)` appears in a p-state q then the Petri net corresponding to this blueprint has to be augmented with a new place r such that $|\bullet p| = |\bullet r|$ and $|q \bullet| = |r \bullet|$.

5 An Agent-Based Workflow Management Model

Workflow systems are designed to automate complex activities consisting of many dependent tasks. Workflow management systems, WFMS, are widely used to automate business processes [1], and there is growing interest in their application to data-intensive scientific and engineering problems [18]. There is ample evidence that the business use of workflows increases productivity and improves the quality of the products of an organization and we should expect similar benefits for scientific and engineering applications, e. g. for tasks that involve data collected from many sites and intricate computations to extract useful information from the data.

Typically, WFMS support processing of documents, of mail, or are concerned with coordination of legacy and newly developed applications. There exists a variety of commercial offerings, usually not developed from ground up as WFMS but built on top of applications designed for other uses, such as image-processing, e. g. FileNet's WorkFlo, relational database management systems, e. g. IBM's FlowMark and computer supported collaborative work e. g. Lotus Notes. The interest in WFMS led to the creation of the Workflow Management Coalition [5] that brings together designers and users of WFMS systems.

5.1 Workflow Management Systems

A workflow system consists of several components: workflow/process definition module, the workflow engine(s), support and monitoring tools, various client applications. Most existing systems are RDBMS-based and, in theory, can take advantage of the reliability, availability and scalability features of the underlying database system. However, in practice workflow systems require advanced transaction models not supported by most commercial DBMS. Consequently, an Workflow Management System built on top of the DBMS does not exhibit desired properties [1]. Moreover, many of the tasks that a workflow system might be used to automate are non-transactional in nature (e. g. interaction with humans, integration of applications that do not provide transactional semantics support), which further limits the benefits of basing a workflow management system on a DBMS ([19]). It is also worth noting that the typical usage profile of a WFMS is dissimilar to those of a RDBMS. RDBMS systems usually are subject to many read requests and relatively few updates while in the case of WFMS the queries are not very elaborate (do not require sophisticated data query language features) and updates occur with higher frequency.

Uses of agents in WFMS have been discussed in several publications e. g. in [3, 15]. Usually in WFMS implementations agents act as personal assistants performing actions on behalf of the workflow participants and/or facilitating interaction with other participants or existing WFMS.

Recently we proposed an agent–based architecture for workflow enactment and for the monitoring components of a workflow management system. In particular we concentrate on the use of agents as *case managers*: autonomous entities overlooking the processing of single units of work. Our assumption is that an agent–based implementation is more capable of dealing with dynamic workflows and with complex processing requirements where many parameters influence the routing decisions and require some inferential capabilities. We also believe that the software engineering of workflow management systems is critical and instead of creating monolithic systems we should assemble them out of components and attempt to reuse the components.

5.2 Interoperability and Meta–Models

As a part of its standardization effort, the Workflow Management Coalition defined a common meta-model for describing the workflow process definition and a textual format for the interchange of workflow process definitions – the Workflow Definition Language (WfDL) ([4]). Some authors believe that the semantics of workflows is still not well defined in WfMC standards, and Petri nets should be considered a lingua franca for workflows . The entities defined by the model WfDL include:

(a) workflow participants representing sets of resources or humans that can act as performers of various activities in the process definitions;
(b) workflow applications which may be invoked to support or wholly automate the processing associated with each activity;
(c) process activities representing logical, self-contained units of work to be processed by either participants or applications;
(d) transitions defining flow of control between activities with optional conditions associated with them; participant definitions; and
(e) workflow relevant data, information passed among activities that can influence the course of execution of the workflow process.

In this paper we shall be mostly concerned with the entities describing flow of control in the workflow process definition, that is activities and transitions. The design choice of the authors of the WfDL specifications was to include as much of the routing information as possible in the activity definition and retain the transition definitions as simple routing assignments. In effect each activity description has three sections,
(1) A "prologue" defining the behavior of the activity if multiple incoming transitions exist, referred to as the join type of the activity,
(2) The definition of the actual activity, and
(3) An "epilogue", that describes the behavior when multiple outgoing transitions exist, referred to as the split type of the activity. An activity A's split (join) type is `AND` if many activities following (preceding) A are to be performed in parallel; activity A's split (join) type is `XOR` if only one of the specified activities is to be performed after (before) A.

Some authors believe that the semantics of workflows is still not well defined in WfMC standards, and Petri nets should be considered a lingua franca for workflows .

5.3 Bond Workflow Management Architecture

An objective of the workflow management framework discussed in this paper is to build case manager agents based upon a static description of an workflow. There are several ways of specifying a workflow. Several existing commercial products use a Petri-Net based definition language or design tool, but the Workflow Definition Language is becoming an industry standard. While we intend to base our framework on Petri nets concepts and possibly maintain interoperability with the above mentioned products, we would like to provide a translation facility from the Workflow Definition Language to the Petri net based representation. This representation requires further transformations to obtain a Bond blueprint.

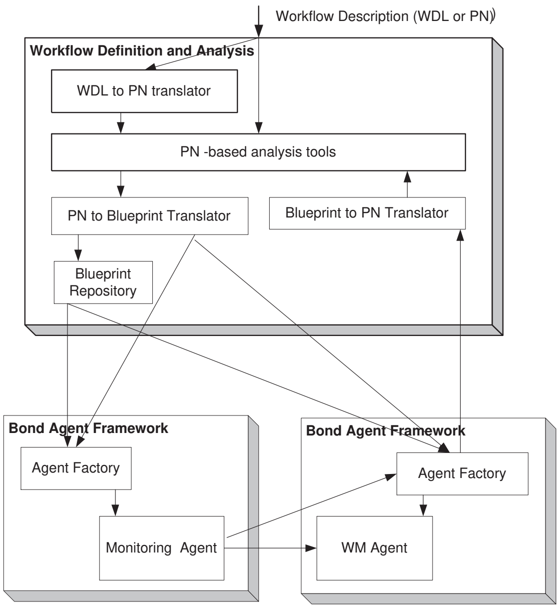

Figure 6: Workflow management in Bond

Figure 7 illustrates the definition and execution of a workflow in Bond. Our design supports dynamic workflows and complex monitoring. The workflow management agent originally created from a static description can be modified based upon the information providing by the monitoring agent. Several workflows may be created as a result of mutations suffered by the original workflow [17]. Once the new blueprint is created dynamically, it goes through the analysis procedure and only then it can be stored in the blueprint repository. The distinction between the monitoring agent and the workflow management agent is blurred, if necessary they can be merged together into a single agent.

6 Conclusions

The multi-plane state agent model we have proposed can be mapped easily into the Petri Net computational model. We came to this realization while developing agent-based workflow management systems. The current realization of this agent model, the Bond Agent Framework, is a component based architecture where strategies encapsulating the behavior associated with individual states are combined together into a coherent system with a clear agenda.

The isomorphism between agents and net models could probably be exploited by a variety of applications besides workflows. At the same time it is beneficial to the agent community to have access to a more rigorous analysis methodology for static and dynamic properties of agents. Petri Net analysis tools including reachability, invariants, incidence matrix could provide answers to many intriguing questions related to agent design.

In this paper we presented the translation from the Petri Net language to the Blueprint Agent Description Language and from Blueprint to Petri Nets.

7 Acknowledgments

The work reported in this paper was partially supported by a grant from the National Science Foundation, MCB-9527131, by the Scalable I/O Initiative, and by a grant from the Intel Corporation.

References

[1] G. Alonzo, D. Agrawal, A. El-Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems. Technical report, IBM Almaden Research Center, 1997.

[2] L. Bölöni and D. C. Marinescu. A Multi – plane State Machine Agent Model. Technical Report CS TR 99-027, Computer Sciences Department, Purdue University, 9 1999.

[3] J. W. Chang and C. T. Scott. Agent–based Workflow: TRP Support Environment (TSE). *Computer Networks and ISDN Systems*, 28:1501, 1996.

[4] Workflow Management Coalition. Interface 1: Process definition interchange process model, 11 1998. WfMC TC-1016-P v7.04.

[5] Workflow Managment Coalition. Workflow managment coalition terminology and glossary. Technical Report WFMC-TC-1011, Workflow Managment Coalition, 1996.

[6] D. Lee D, T. Nishimura, and S. Kumagai. Structural and behavioral analysis of state machine allocatable nets based on net decomposition. *IEICE Trans. Fundamentals*, E79-A(3):399–408, 1993.

[7] J. Desel and J. Esparza. *Free Choice Peri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.

[8] Tim Finin et al. Specification of the KQML Agent-Communication Language – plus example agent policies and architectures, 1993.

[9] S. Franklin and A. Graesser. Is it an agent, or just a program? In *Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages*. Springer Verlag, 1996.

[10] R. Hao, K. Jun, and D. C. Marinescu. Bond system security and access control models. In *roceedings of IASTED Conference on Parallel and Distributed Computing*, pages 520–524. ACTA PRESS, 1998.

[11] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts. In *2nd IEEE Symposium on Logic in Computer Science*, 1987.

[12] K. Jun, L. Bölöni, K. Palacz, and D. C. Marinescu. Agent-based resource discovery. Technical Report 034, Department of Computer Sciences, Purdue University, 1999.

[13] K. Jun, L. Bölöni, D. Yau, and D. C. Marinescu. Intelligent QoS support for an adaptive video service. Technical Report 033, Department of Computer Sciences, Purdue University, 1999.

[14] Object Management Group. *OMG Unified Modeling Language Specification.*

[15] P. D. O'Brien and M. E. Wiegand. Agent based process management: applying intelligent agents to workflow. *Knowledge Engineering Review*, 13:2, 9 1998.

[16] P. Tsompanopoulou, L. Bölöni, D. C. Marinescu, and J. R. Rice. The Design of Software Agents for a Network of PDE Solvers. In *Workshop on Agent Technologies for High Performance Computing, Agents 99*, pages 57–68. IEEE Press, 1999.

[17] W. M. P. van der Aalst and T. Basten. Inheritance of Workflows. An approach to tackling problems related to change. (draft).

[18] J. Wainer, M. Weske, G. Vossen, and C. B. Medeiros. Scientific workflow systems. In *NSF Workshop on Workflow and Process Automation: State of the Art and Future Directions*, May 1996. http://wwwmath.uni-muenster.de/math/inst/info/u/dbis/Weske/papers/wwvm96.html.

[19] D. Worah and A. Sheth. What Do Advanced Transaction Models Have to Offer for Workflows. In *Proceedings of the International Workshop on Advanced Transaction Models and Architectures (ATMA), Goa*, August 1996.

8 Appendix 1: An Example of a WfDL Definition

```
MODEL 'Simple'
        WPDL_VERSION "1.0"
        VENDOR "Purdue University"
        CREATED 1999-10-18

WORKFLOW 'Simple_Test'
        CREATED 1999-10-18
        NAME "Simple Workflow Test"

        ACTIVITY 'start'
                IMPLEMENTATION APPLICATIONS
                        TOOL_LIST 'Workflow.Run' ("ls")
                        END_TOOL_LIST
                SPLIT AND
        END_ACTIVITY

        ACTIVITY 'branch_a'
                IMPLEMENTATION APPLICATIONS
                        TOOL_LIST 'Workflow.Run' ("pwd")
                        END_TOOL_LIST
        END_ACTIVITY

        ACTIVITY 'branch_b' IMPLEMENTATION NO SPLIT XOR 'alt_a' 'alt_b'
```

```
        END_ACTIVITY

        ACTIVITY 'alt_a' IMPLEMENTATION APPLICATIONS 'Dialog.OkDialog'
        END_ACTIVITY

        ACTIVITY 'alt_b' IMPLEMENTATION APPLICATIONS 'Dialog.OkDialog'
        END_ACTIVITY

        ACTIVITY 'rendez_vous' IMPLEMENTATION NO JOIN XOR
        END_ACTIVITY

        ACTIVITY 'stop' IMPLEMENTATION NO JOIN AND
        END_ACTIVITY

        TRANSITION 't1_a' FROM 'start' TO 'branch_a' END_TRANSITION
        TRANSITION 't1_b' FROM 'start' TO 'branch_b' END_TRANSITION
        TRANSITION 't2_a' FROM 'branch_a' TO 'stop' END_TRANSITION
        TRANSITION 't2_b1' FROM 'branch_b' TO 'alt_a' END_TRANSITION
        TRANSITION 't2_b2' FROM 'branch_b' TO 'alt_b' END_TRANSITION
        TRANSITION 't3_b1' FROM 'alt_a' TO 'rendez_vous' END_TRANSITION
        TRANSITION 't3_b2' FROM 'alt_b' TO 'rendez_vous' END_TRANSITION
        TRANSITION 'z' FROM 'rendez_vous' TO 'stop' END_TRANSITION

END_WORKFLOW
END_MODEL
```

9 Appendix 2: Petri Net Representation and Translation to Blueprint

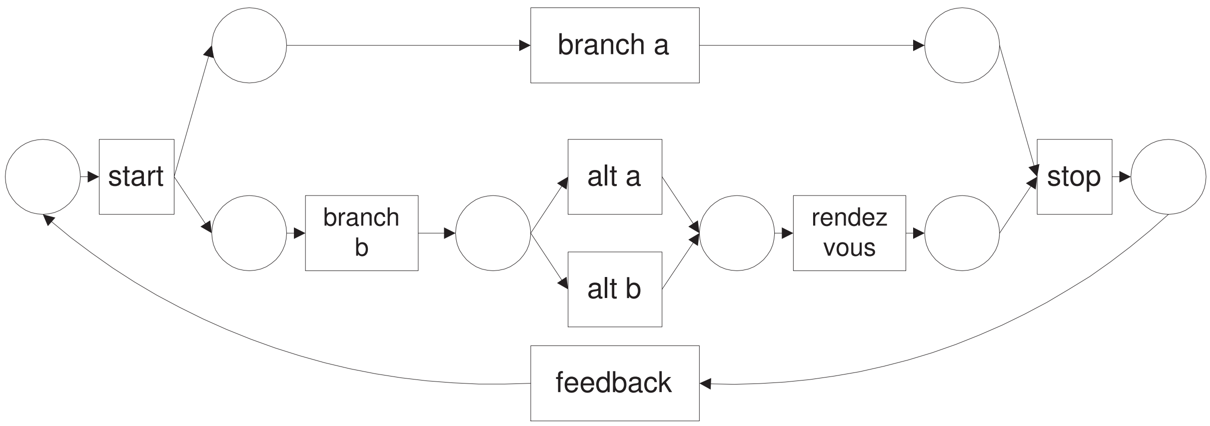

Figure 7: The workflow definition from Appendix 1 translated into a Petri net

```
import bond.agent.strategydb;
import bond.application.workflow.strategies;
create agent Workflow
plane plane_0
add state rendez_vous with strategy Util.WaitAndSucceed;
add state branch_b with strategy Util.WaitAndSucceed;
add state feedback with strategy Workflow.Feedback::feedback;
add state alt_b with strategy Dialog.OkDialog::alt_b;
```

```
add state alt_a with strategy Dialog.OkDialog::alt_a;
add state stop with strategy Util.WaitAndSucceed;
add state start with strategy Workflow.Run::start
model {
        arguments = ["ls"   ];
};
add state branch_b_split with strategy Workflow.Route::branch_b_split
model {
        out = ["alt_b", "alt_a"];
};
add state rendez_vous_split with strategy Workflow.Route::rendez_vous_split
model {
        out = ["stop"];
};
add state output with strategy Workflow.Route::output
model {
        out = ["feedback"];
};
add state input with strategy Workflow.Route::input
model {
        out = ["start"];
};
add state branch_b_join with strategy Workflow.Route::branch_b_join
model {
        out = ["branch_b"];
};
add state alt_a_split with strategy Workflow.Route::alt_a_split
model {
        out = ["rendez_vous"];
};
internal transitions {
        from alt_a_split to rendez_vous on enter_rendez_vous;
        from rendez_vous to rendez_vous_split on success;
        from branch_b_join to branch_b on enter_branch_b;
        from branch_b to branch_b_split on success;
        from output to feedback on enter_feedback;
        from feedback to input on success;
        from branch_b_split to alt_b on enter_alt_b;
        from alt_b to alt_a_split on success;
        from branch_b_split to alt_a on enter_alt_a;
        from alt_a to alt_a_split on success;
        from rendez_vous_split to stop on enter_stop;
        from stop to output on success;
        from input to start on enter_start;
        from start to branch_b_join on success;
}
end plane;
plane plane_1
add state branch_a with strategy Workflow.Run::branch_a
model {
        arguments = ["pwd"   ];
};
add state branch_a_split with strategy Workflow.Wait::branch_a_split
model{
        waitfor = "rendez_vous_split";
        goto = "branch_a";
};
```

```
internal transitions {
        from branch_a_split to branch_a on enter_branch_a;
        from branch_a to branch_a_split on success;
}
end plane;
end create.
```