CHAPTER 31

Making MATSim Agents Smarter with the Belief-Desire-Intention Framework

Lin Padgham and Dhirendra Singh

31.1 Basic Information

Entry point to documentation:

`http://matsim.org/extensions` → bdiintegration

Invoking the module:

See `http://matsim.org/extensions` → bdiintegration

Selected publications:

Padgham et al. (2014)

31.2 Introduction

In this chapter, we introduce a MATSim extension allowing a developer to program (some of) an agent's decision-making in a BDI (Belief Desire Intention) system, while actual actions and environment percepts occur within MATSim.[1] This allows sophisticated modeling of agents within a BDI framework, using the concepts of goals, hierarchical abstract plans (containing sub-goals)

[1] This work was supported by the ARC Discovery DP1093290, ARC Linkage LP130100008 and Telematics Trust grants. We would like to thank Agent Oriented Software for use of the JACK BDI platform and Kai Nagel, Todd Mason, Sewwandi Perera, Edmund Kemsley, Oscar Francis, Daniel Kidney, Andreas Suekto, Qingyu Chen, and Arie Wilsher for their contribution to the BDI platform integration framework and to these applications.

How to cite this book chapter:

Padgham, L and Singh, D. 2016. Making MATSim Agents Smarter with the Belief-Desire-Intention Framework. In: Horni, A, Nagel, K and Axhausen, K W. (eds.) *The Multi-Agent Transport Simulation MATSim*, Pp. 201–210. London: Ubiquity Press. DOI: http://dx.doi.org/10.5334/baw.31. License: CC-BY 4.0

and percepts (information from the environment), as well as information about the current situation. For example, we used it to model residents in a bushfire[2] evacuation, as well as an incident controller in an evacuation scenario. The residents may receive information about the bushfire from the fire simulation, as well as warnings and messages from the incident controller agent. They may well have to pick up children, check on neighbors and communicate with other family members, etc. Their plans enable decision-making, which will result in actions executed within MATSim.

In standard MATSim usage, intelligence within individual agents' behavior arises from co-evolutionary algorithms in the replanning phase. This is based on agents evaluating—via a scoring function—the plan they have executed during a given day and modifying this to obtain a new plan, until all agents have acceptable plans; the system then reaches a stable state. This approach, however, only works for applications where one can assume that the agents adjust and refine their behavior over many iterations, to eventually obtain their standard modus operandi. For applications such as emergency management, agents must react immediately to the situation as it evolves, doing so in an "intelligent" manner.

The chapter on Within-Day Replanning introduces two approaches to the mobsim component which address the need to be more reactive to an evolving situation. The first allows a centralized MATSim process to identify sets of agents that should have their plans modified, then runs one or more processes to adjust agents' plans. The second rewrites the agent, so that instead of following a specified plan, the agent invokes a decision-making process at all possible decision points. By integrating a BDI agent platform with MATSim (Padgham et al., 2014), we allow autonomous individual decision making to be programmed in specialized and powerful systems developed specifically for this purpose, balancing reactive behavior and goal-based commitment. Different BDI platforms have different strengths, but are, in general, based on a simplified psychological/philosophical view of how people behave, facilitating a high level specification of complex human behavior. These systems have been demonstrated to be very efficient for building complex applications (Benfield et al., 2006). Provided the appropriate system interface support is developed, any BDI system can be coupled to MATSim, as described here. Until now, we have used three different BDI systems, for which the system level interface is available. The decisions made in the BDI system are then inserted into the relevant agents' MATSim plans, allowing the MATSim agents to operate in the same efficient manner as in standard MATSim.

31.3 Software Structure

Our framework supports independent execution of MATSim and the BDI platform, with synchronization via the infrastructure provided. They can either run within a single process (in separate synchronized threads, or sequentially in a single thread), or in two separate processes (synchronizing using inter-process communication, such as sockets). The former is, of course, considerably more efficient. Conceptually, for every MATSim agent whose decision making is to be carried out in the BDI system, a BDI agent must be created. The BDI counterpart can be regarded as "the brain" associated with the MATSim agent. It is possible to have BDI agents with no MATSim counterpart and vice versa. For example, in our bushfire application, the incident controller has no MATSim agent, as he does not move on the road network. He receives information about the fire and has some static location information; his role in the simulation is to issue warnings and evacuation advisories, which, in turn, affect the resident agents. There may also be MATSim agents that do not have a BDI counterpart. For example, in a taxi modeling application, there may be MATSim

[2] Bushfire is the Australian term for what is otherwise known as a wildfire or forest fire.

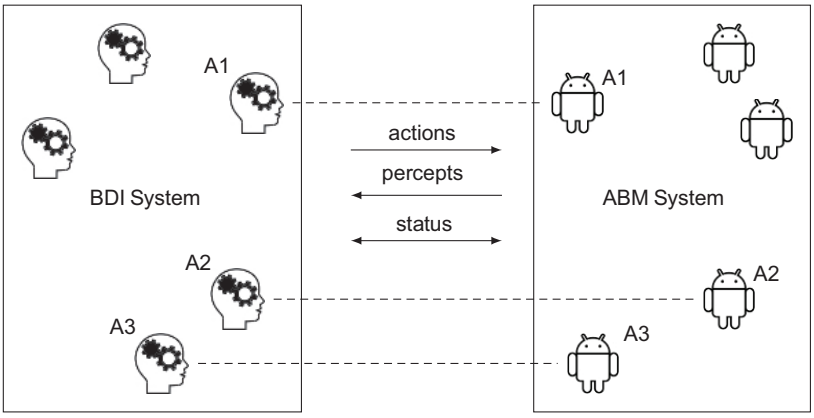

Figure 31.1: Conceptual BDI-ABMS integration architecture.

Source: Figure adapted from Padgham et al. (2014, Figure 1) distributed under the Creative Commons Attribution Non-Commercial License

agents using the road network, but with no need for complex decision-making modeling; these may exist only within MATSim.

Figure 31.1 shows the two parallel systems' basic architecture and the information passed between them at each time step.

The structure of the data components passed between the MATSim agent and its BDI counterpart is shown in Table 31.1 and consists of *BDI Actions*[3], *Percepts* and *Queries*. As indicated in Figure 31.1, BDI-actions are always initiated by the BDI system. Their status field, however, can be modified by both systems. When a BDI action such as `DriveTo(loc)` is decided by the BDI agent, the BDI system sets the status of this action as "INITIATED". MATSim will then set its status to "RUNNING", which will probably remain in this state for several steps. When the `loc` destination is reached, the MATSim routine will set the status to "PASSED" and the BDI system will continue reasoning about the next stage of agent behavior. If desired, the MATSim routine can also detect situations which should be conveyed as "FAILED" and pass this to the BDI counterpart. For example, if there is a BDI action to meet at a location and time and the MATSim agent is delayed in traffic, the BDI action implementation in MATSim can be programmed to detect the missed deadline and set the status to "FAILED", at which point the BDI agent will attempt failure recovery (as part of the BDI infrastructure). The BDI system can also set the status to "ABORTED"—for example, if information arrives requiring a different action—in which case, it is canceled within MATSim. The BDI system can also set status to "SUSPENDED", though this is not currently implemented.

To manage BDI actions, we provide a `MatsimAgentManager` class responsible for updating BDI actions status for all agents. At each step, the `MatsimAgentManager.updateActions(...)` function identifies (from the information package supplied by the BDI system) all agents initiating, aborting, or suspending actions. These are the agents which may require their MATSim plans to be modified. For each agent that has some action with s status "INITIATED", the action is passed to the agent's action handler class `MatsimActionHandler` via a call to `MatsimActionHandler.processAction(agentID, actionID, params)`. This function, based on the action, calls an appropriate helper function that performs required modifications to the MATSim plan and other relevant bookkeeping, to ensure that success and failure are observed (via

[3] We call these actions BDI Actions to distinguish them from actions in the ABMS (Agent-Based Modeling and Simulation) which may include lower level or additional actions.

Components of The Data Package Provided to Specific Agents Via The Interface:	
Component Type	Component fields
BDI action	< *instance_id*, *action_type*, *parameters*, *status* >
Percept	< *percept_type*, *parameters*, *value* > (parameters and value may be complex objects)
Query	< *query*, *response* >
BDI Action Status:	
State	Description
INITIATED	Initiated by BDI agent and to be executed
RUNNING	Being executed, set by the simulation agent
PASSED	Completion detected and set by the simulation agent
FAILED	Failure condition detected and set by the simulation agent
DROPPED	Aborted by the BDI agent
SUSPENDED	Temporarily suspended by the BDI agent

Table 31.1: Data Passed Between The BDI and ABMS Systems

appropriate MATSim callbacks) and that status is reported back to the BDI system. For example, for a `DriveTo` action, a `processDriveTo(agentID, loc)` function is executed to determine the leg associated with `loc`, obtain a route using the MATSim router and insert this into the MATSim agent's plan. The standard MATSim execution then follows this plan at each subsequent step. If the `processAction` function returns a success status indicating that the action was handled successfully, then `updateActions` changes the status for this action to "RUNNING"; otherwise, it sets it to "FAILED."

Sometimes, a running action can also fail in the ABMS for some reason. For instance, a `DriveTo (loc)` action could fail due to a road-closure in a bushfire evacuation simulation. While this functionality is supported by our infrastructure, it has not yet been used in the applications we have built with MATSim. Failing actions will soon be added for some applications. Aborting and suspending are also not currently implemented for MATSim. This would be accomplished by having appropriate functions declared which reset the plan contents of the agent to a 'holding state' (activity with infinite end time), maintaining the removed contents of a suspended plan in some data structure for eventual resumption.

Percepts capture information identified as necessary for the BDI agent's reasoning. Typically, this is any information leading to triggering of a BDI-goal, or causing an executing goal/plan to be re-evaluated. Approaching a destination is one example. MATSim callbacks are used to capture the relevant information within MATSim; this is then provided to the BDI counterpart via our infrastructure. The appropriate MATSim event is caught with `AgentActivityEventHandler. handleEvent(event-type)`. The `handleEvent(event-type)` function then first checks whether the agent receiving the event is one registered for a percept that triggers with this event type, and if so, calls the appropriate function to calculate the percept's value and add it to the percept container for that agent, to be sent to the BDI system. Termination conditions (PASSED and FAILED) of BDI actions are also similarly detected.

Instead of passing back the percept in these cases, the relevant action and its status is edited and passed back. For example, a BDI action `DriveTo(loc)` should succeed when the agent reaches the link closest to this location. To achieve this, we implement `handleEvent(PersonArrivalEvent)`, which will then trigger for every agent arriving anywhere. If the agent has a current (`DriveTo`) BDI action being monitored, then `arrivedAtDest(agentID,loc)` is called to ascertain whether the

PersonArrivalEvent caught does match the link closest to the coordinates of the desired destination. If it does, the action status of that DriveTo action for that agent is changed to PASSED and the action is removed from the monitoring list.

This approach conveniently uses MATSim callback infrastructure. However, we note that it will generate an event that must be processed any time any agent arrives anywhere, although most will not be an arrival at a desired destination. This is a substantial overhead; we may eventually consider collecting (some) percepts and state information for determining action status, in a separate, more efficient global processing at the end of the step.

Queries are defined for any information that the BDI system may want to request from MATSim during its reasoning process. Typically, queries are based on plans' context conditions, which must be evaluated to determine if a plan is applicable. Each query structure must be defined and the code must be supplied on the MATSim side to call the relevant functions to provide the response. Similar to the MatsimActionHandler class, we have a MATSimPerceptQueryHandler class containing a queryPercept(agent,query,response) function. This function then uses the query string received to extract the percept type and make a specific function call to obtain and provide the results. For example, if an agent agentID sends a queryPercept(agentID, ''RequestLocation agentX'', loc) query to request the location loc of some agent agentX (possibly itself), then the queryPercept function will execute the clause:

```
if percept_type = "RequestLocation"
   loc = getLocation("agentX")
```

The agentID of the requesting agent, obtained from the data package, is always provided to the query response function, in case it is required, although in this case it is not. Queries can be made at any point during the BDI execution and are answered immediately. They have no effect on the MATSim simulation.

A number of commonly used BDI actions and percepts are defined as part of our integration infrastructure. New ones can be added as part of developing a specific application, as described in Section 31.4. This structure allows all high-level decision making to be carried out by individual agents, within the BDI-system, which is designed and optimized for this purpose with regard to both representation and execution. On the MATSim side, specified functions simply modify the agents' MATSim plans (in parallel, if desired), retaining the standard MATSim simulation execution where each agent just follows its MATSim plan. This approach allows for both simplicity and efficiency at the lower level.

31.4 Building an Application Using BDI Agents

We focus here only on what must be done to integrate BDI agent reasoning into MATSim. To learn about BDI design and development, we refer the reader to Padgham and Winikoff (2004), as well as the excellent "practicals" (tutorials) available as part of the JACK platform[4]. In Figure 31.2, we show part of a taxi agent design, in an application involving taxis operating within MATSim. Here, the percept ClosetoDest (potentially) triggers a plan GrabJob. Plans have context conditions which indicate whether or not they are viable in the current situation, as a response to a percept, or a way of achieving a goal. Let us assume, in this example, that the plan GrabJob has the context condition (Location(self,loc)) ∧ board.job.loc ∧ (distance(board.job.loc,loc) < 4km). Thus, the figure at the left of the diagram can be understood as the rule:

ClosetoDest ∧ Location(self,loc) ∧ board.job.loc ∧ (distance(board.job.loc,loc) < 4km) → GrabJob

[4] http://aosgrp.com/products/jack/

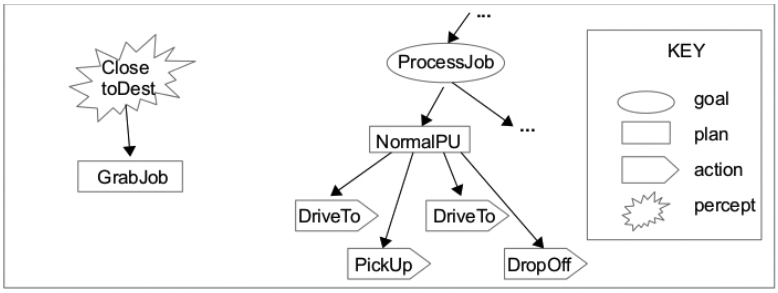

Figure 31.2: Excerpt of taxi design.

There are two pieces of information in this rule that must come from MATSim: first, the agent is close to its destination (`ClosetoDest`) and second, the agent's current location (`Location(self, loc)`). We could have MATSim send the agent location at every step. However, this is unnecessary overhead; instead, we send `ClosetoDest` as a percept. This requires the BDI agent to query its location to evaluate whether there are pending jobs whose location necessitates triggering some instance of `GrabJob`. This gives us an example of a percept and a query required in MATSim. On the right hand component in Figure 31.2, we see four different actions which will have a corresponding BDI-action on the MATSim side. We will focus here on the `DriveTo` action, but the `PickUp` and `DropOff` would be realized in a similar way, using MATSim activities rather than legs.

The following must usually be done:

- Every plan trigger which is information from MATSim must be defined as a percept.
- All information required from MATSim, that is not a trigger, must be defined either as a percept (and then stored locally), or as a query.
- All actions which should be executed in MATSim must be defined.

In the rest of this section, we describe exactly what must be provided in the MATSim application files for each of these to work as expected. Instructions and examples for the BDI application can be found in the integration repository (noted at start of chapter).

31.4.1 The `ClosetoDest` *Percept*

All functions for collecting percepts for the BDI system are defined in the `AgentActivityEventHandler` class. Perusal of existing functions can ascertain whether the desired percept is already calculated. For example, `arriveAtDest` is already defined for use as a BDI percept. If the percept collection function already exists, the developer must ensure that the appropriate agent type is registered for this percept within the relevant function. For example, in `arriveAtDest()` we have:

```
if agent.type = taxi
   AND agent.loc = dest(agent) \* obtained from infrastructure data *\
       // collect and package this percept
```

If we now want this percept provided to agents of type commuter, we must make the first line:

```
if ((agent.type = taxi) OR (agent.type = commuter))
   AND agent.loc = dest(agent) \* obtained from infrastructure data *\
       // collect and package this percept
```

The arriveAtDest function is triggered by the MATSim LinkEnterEvent event using MATSim provided callbacks. Thus, we have defined handleEvent(LinkEnterEvent) to call all percept collection functions triggered by this event – in this case arriveAtDest.

The ClosetoDest percept will be triggered by the same MATSim event LinkEnterEvent, so to add this, we must add the call to ClosetoDest in the handleEvent(LinkEnterEvent) and then define our ClosetoDest function within the AgentActivityEventHandler class. We only want to send the ClosetoDest percept when we first come within the defined distance of our destination, not at every step. Therefore, the ClosetoDest function must first check whether this percept has already been sent to this agent, for the current destination. If so, nothing more is done. If not, it is ascertained whether the link entered is within the desired "close-to" distance and, if so, the percept is registered. For efficiency, the first link "close-to" the dest can be calculated and recorded when the DriveTo action is initiated; in which case, one must only check whether the entered link-ID is the same as the recorded "close-to" link-ID.

In principle, percepts could also be calculated in a function executed after all agents had been stepped. The important thing is that when a percept occurs, it is recorded in the percept data package for that agent. Further work is required to ascertain which percept collection methods will be most efficient with very large numbers of agents.

31.4.2 The RequestLocation *Query*

Queries are defined in, and managed through, the MATSimPerceptQueryHandler class. A function queryPercept(agent,query,response) responds to a query by extracting the specific query and calling the relevant defined function. So, for example, to respond to the queryPercept(ownID, ''RequestLocation agentID'',loc) query from an agent, queryPercept will contain the code:

```
if percept_type = "RequestLocation"
   loc = getLocation("agentID")
```

The getLocation function will then ascertain the location of agentID, storing the value in loc. If the query is already defined in MATSim, nothing further is required to use it in an application.

31.4.3 The DriveTo *BDI-Action*

The DriveTo(loc) BDI action is, of course, the most basic and commonly used BDI action in MATSim and is already implemented in our infrastructure. As long as the appropriate BDI action and parameters are passed in the information package from the BDI system, nothing further is required within MATSim. However, for the purpose of illustration, we will assume it has not yet been implemented and we will go through the steps of defining a new BDI action with this as an example.

The MATSimActionList class defines mappings for all BDI actions in the system and the MATSim function calls that realize those BDI actions. Any new BDI action must first be added to this list.

The MATSimActionHandler defines all functions that realize BDI actions, as well as a processAction function which handles all BDI action strings from the BDI system, calling the appropriate helper functions. Thus any new BDI action must have its implementation defined within this class and must have the appropriate call to the function added within processAction. Let us call the relevant function that we will add processDriveTo. This function will always need the agentID as a parameter, as well as whatever parameters are provided in the action package.

So, in our example, we will have the function `processDriveTo(agentID, loc)` which needs to be defined. The function for the new action must perform two key tasks:

1. Obtain the MATSim plan of the relevant agent and modify it so that regular MATSim execution of the plan will have the desired effect.
   Generally, when the plan is accessed, it will have a single dummy activity with end-time infinity. The end time of this activity must be set to now and a leg must be instantiated with the link corresponding to the destination `loc` as the end point and the links to be followed, as calculated by the router. This leg must then be inserted into the plan, followed by a new dummy activity instance with end time infinity.
2. Place the action instance into the list of actions being monitored.

It is also necessary to set up recognition of when the action has finished, so that this information can be sent back to the BDI system and the agent can continue to reason about its next actions. This is done via the MATSim callbacks provided, in the same way as detecting percepts. However, the corresponding function, instead of placing information in the percept package for the agent, will modify the `status` of the relevant BDI action instance in the information package to PASSED and remove the instance from the list of actions being monitored. It is also possible to define a condition where the action should be considered FAILED and to detect this in a similar way. Alternatively, failure can be managed by sending a percept, and having the BDI agent abort the action as a result[5].

The current structure assumes that multiple actions of a single agent cannot be executed in parallel (a reasonable assumption for MATSim). It is the responsibility of the BDI system to allow only one active BDI action per agent.

Further instructions, as well as examples, can be found in our BDI-MATSim integration repository.

31.4.4 Discussion

An important aspect of a simulation design using BDI agents within MATSim is deciding on which abstraction level BDI actions should be described. So far, we have tended to have BDI actions map to a single `leg` or `activity` within a MATSim plan. However, it is certainly easy to think of BDI actions that combine several such components. Straightforward examples would be grocery shopping or taking kids to school - both involving a leg to a destination, an activity at that destination and a return leg. There are no immediately obvious advantages associated with BDI actions at higher abstraction levels (requiring coding of these actions in MATSim) vs using lower level BDI actions with the higher level coded as BDI plans/goals. Future experience and experimentation may provide insights to guide decisions.

31.5 Examples

Here, we describe two different examples of BDI agents within MATSim: a bushfire evacuation simulation, where MATSim is being used because traffic flow is a crucial component in this type of evacuation and a taxi application developed as a demonstrator for integration of a BDI system with MATSim (Padgham et al., 2014). We compare this approach to incorporating taxis with that described in Chapter 23 for incorporating dynamically scheduled vehicles and with the approaches to "within-day replanning" described in Chapter 30.

[5] The simplest way in JACK is to use a maintenance condition relying on a belief that is modified as the result of a percept.

Both our example applications use only the `Mobsim` engine (`QSim`) of MATSim and do no repeated daily cycles with plan scoring and modification. There are undoubtedly applications which could benefit from a combination of BDI agents and agents which evolve using MATSim's scoring and replanning, but we have not yet investigated them.

31.5.1 Bushfire Example

The bushfire example (currently) involves modeling of residents and their decision-making behavior about what to do regarding a nearby bushfire. Potential driving activities include picking up children from a school or other facility, checking on neighbors or friends and driving to a local or more distant destination, possibly via a specified route. Decision making may involve various factors, such as time of day, ideas about what other family members are doing, warnings and notifications from emergency services, observations of neighbors, etc. In one approach, we focused on incorporating well-developed and validated actual human decision making models in a bushfire situation, developed by a collaborator. Our contribution has been to integrate this with MATSim, using our integration framework, to provide data about any traffic-related issues, thus providing a more valuable simulation to planners. In our other approach, we model both residents and an incident controller. Here, our focus has been on technical issues that involve providing an interactive simulation suitable for use by emergency services personnel and/or communities for exploration of potential strategies.

In the interactive version, the incident controller assigns specified evacuation centers and routes to residents in certain sections of the town being evacuated. Evacuation of different areas may be started at different times. Residents follow the incident controller's instructions with some probability based on their individual situations (currently modeled very superficially). Following the suggested route is achieved by driving via suggested way points (using the `DriveTo` BDI action), with the BDI agent (potentially) re-assessing as each waypoint is reached. An alternative would be to define a new BDI action `DriveToViaWaypoints`. One issue that arose during the development of this simulation involved road congestion; MATSim routing algorithms began developing very circuitous routes, sometimes going back towards the fire threat. There were two issues illustrated here about developing a realistic simulation: one was that, realistically, people would not choose their routes based on global knowledge of current congestion; the other was that, regardless of congestion, people would not head back into the fire zone. The current solution is to use a routing algorithm not accounting for current road speeds, using only static speed limits. Going forward, one may want to assume some knowledge of congestion (based on radio broadcasts or other social media). An interesting future research question is how to best achieve responsibility sharing for realistic behavior between MATSim and the BDI decision-making program, on route selection.

31.5.2 Taxi Example

The taxi prototype application was developed purely as a 'proof of concept', allowing decisions to be made dynamically by the BDI brain on an ongoing basis, then carried out by the MATSim execution engine. There is a simple taxi administrator in the BDI system, which generates jobs, posts them to a notice board and confirms requests from taxis to take specific jobs. Taxis have plans allowing them to take jobs from the board, go to a taxi rank, or take a break. After taking a job from the board, the taxi drives to the pick-up address, picks up the passenger, then drives to the destination and drops them off. When the taxi approaches the destination, it looks on the job board for nearby jobs; if something suitable is found, it requests it from the administrator. The only BDI action implemented in this application is a simple `DriveTo`. The `ClosetoDest` percept was used as described in Section 31.4. This application was tested with the Berlin road network and the 15 963 agents in the MATSim sample files, with all agents operating as BDI taxi agents. Profiling

showed that, by far, the majority of the execution time was spent in route planning, with very little in the BDI reasoning, or communication with the BDI system.

31.5.3 Discussion

Both evacuation and taxis are discussed in Chapters 30 and 23, as applications requiring a reactive approach to planning, rather than iteration over many days to find the preferred plan. Chapter 23 discusses two implementation options: one which replaces the MATSim agent with an agent that considers what to do at each relevant decision point (particularly intersections); the other leaves the agent code as is, but modifies the agent's plans when certain events occur. The BDI approach has the computational advantages of the latter, in that only a small subset of agents require changes to their plans at any simulation step and many existing MATSim routines can be used to modify the plans. However, it also has many of the advantages of the former approach; agents are still fully autonomous, with all decision making occurring within the BDI system. By registering for any percepts which could potentially cause the agent to change its mind, the agent remains fully in control at all times. However, it only needs to decide its next action when it completes the current high level action—which will almost certainly be orders of magnitude less often than at each intersection—or when a percept arrives indicating a need to reconsider. The provision of the ability to drop current BDI actions (legs or activities) provides the same level of reactive autonomy as the fully reactive within day replanning agent, but probably at a lower computational cost. Perhaps more important than the computational cost savings: agent decision making can be programmed in a framework that is at a high level of abstraction, using goals, plans and beliefs, within existing highly efficient platforms such as JACK (Winikoff, 2005), Jadex (Braubach et al., 2005) or Jason (Bordini et al., 2007). Design tools for developing such agents also already exist (Padgham and Winikoff, 2004). One study has shown that using a BDI language makes program development hugely more efficient than programming in Java (Benfield et al., 2006). The close mapping between intuitively understandable design diagrams and the program code implementing this in a BDI system is also highly advantageous for validating design of realistic agents with domain experts. We have discussed design of resident agents in a sandbagging flood scenario, with emergency services personnel extremely experienced in that domain and found the representation to be effective. We consider that this representational aspect can be a significant advantage when compared to programming the agent using the `DynAgentLogic` facility described in Chapter 23.