Improving the reactivity of BDI agent programs

Hoa Khanh Dam, Tiancheng Zhang, and Aditya Ghose

School of Computer Science and Software Engineering
University of Wollongong
New South Wales 2522, Australia
{hoa,tz746,aditya}@uow.edu.au

Abstract. Intelligent agent technology has evolved rapidly over the past few years along with the growing number of agent applications in various domains. However, very little work has been dedicated to define quality metrics for the design of an agent-based system. Previous efforts mostly focus on adopting classical metrics such as coupling and cohesion to measure quality of an agent design. We argue that the time has come to work towards a set of software quality metrics that are specific to the distinct characteristics of agent-based systems. In this paper, we propose a method to measure the reactivity of an agent design which provides indications of how the agent system responds to changes in the environment in a timely fashion. The proposed metric is part of the framework which facilitates the restructuring of an Belief-Desire-Intention agent program to improve its reactivity. Our framework was developed into a prototype tool which is integrated with Jason, a well-known agent-oriented programming platform.

1 Introduction

Proactiveness and reactivity are arguably two important characteristics of an intelligent agent system which operates in a dynamic environment [30]. Agents should pursue their goals over time and they should be able to perceive their environment and respond in a timely fashion to changes that occur in it. A critical aspect in an agent's decision-making is balancing proactive and reactive aspects: we want the agent to achieve its goals by default while also taking changes in the environment into account. Practical Belief-Desire-Intention (BDI) [24] agent systems attempt to achieve an effective balance between goal-directed and reactive behaviour by constantly perceiving the environment and reasoning about how to act so as to achieve their goals in terms of selecting appropriate plans from their plan library. A crucial point in BDI systems is that execution occurs at each step and the particular choice of the specific plan to achieve a goal should be left for *as late as possible* so as to consider the *latest information in the environment* the agents might have.

In other words, the choice of plan for a goal which a BDI agent has should only be made when the agent is about to start acting upon it. The context of a plan (or plan's preconditions) is used for checking the current situation in order to evaluate a particular plan among various alternative plans for the goal. For example, consider a goal to *be at the university without getting wet* with two plans: WalkPlan and BusPlan. WalkPlan has the context condition *weather is not raining* and involves walking to a friend's house to collect a book and then walking to the university. BusPlan has the context condition *weather is raining* and involves also walking to the friend's house to collect the book and then catching a bus to the university. According to the plans' definition, the context of each plan is evaluated (with respect to the latest information that the agent has about its environment) just before the agent is about to leave home and go to the university. If the weather is not raining when the agent is about to leave home, the agent would commit to its WalkPlan.

However, changes constantly occur in a dynamic environment and what have been true of the environment when the context of the chosen plan was evaluated may not still be true during the course of the agent executing the plan. In our earlier example, the weather was not raining when the agent was about to leave home but it may start raining while the agent is at the friend's house. If the agent continues executing the WalkPlan, it would get wet and fails to achieve its goal. The problem essentially resides in the way in which the two plans are written, resulting in the agent being not reactive to changes in the environment while pursuing its goal. The issue is that the agent is programmed to evaluate the situation quite early and thus makes an early commitment, which may eventually result in failures. Although there has been a range of work on addressing failure recovery in BDI (e.g. [26]), the issue illustrated in the above example can be dealt with by providing effective support for the software engineers in developing BDI plans that are more suitably reactive to changes in the environment. Unfortunately, this kind of support is rarely provided in the current proliferation of agent-oriented software engineering methodologies [12].

In this paper, we attempt to fill that gap by providing a framework which supports the software engineers in writting BDI agent plans. Specifically, we propose a design metric to measure how reactive a BDI agent is during the course of achieving its goals, i.e. measuring the reactivity of plans available to handle a given goal/event. We then present a methodology to increase the reactivity of an agent based on the effective use of subgoals as a device to postpone execution of actions until possibly more information is available about the context. More specifically, our framework facilitates the identification of actions that are dependent on both the plan's context and changes in the environment, and the grouping of those actions into subgoals. A distinct feature of our methodology is that it focusses only on restructuring the agent program without adding further capabilities to the agents. Our framework is implemented into a toolkit that works with Jason on the Eclipse platform.

The paper is structured as follows. In section 2, we discuss different factors that contribute to the reactivity of an agent program. An abstract programming environment is described in section 3. We then describe an reactivity measure in section 4 and a methodology to restructure an agent program to improve its reactivity in section 5. The prototype implementation of our framework is described in section 6. Finally, we discuss related work (section 7), and conclude and outline some future work (section 8).

2 Factors contributing to reactivity

One of the most well-established and widely-used agent models is the *Belief-Desire-Intention* (BDI) model [16]. BDI agents' behaviour is mostly determined in terms of their plans to handle events or achieve goals[1]. Each plan P is typically of the form $G : [C] \leftarrow B$, meaning that plan P is an applicable plan for achieving goal G when context condition C is believed true. A BDI agent also has a belief set which encodes what the agent perceives its environment. Mechanisms exist to check whether a plan's context condition hold with respect to the agent's beliefs, and to add and delete a ground basic belief to and from a belief base. The plan body[2] B (following abstract notations such as AgentSpeak(L) [24] or CAN [29]) typically contains a sequence of formulæs, each of which can be a domain action that is meant to be directly executed in the world (e.g. lifting an aircraft's flaps) or a subgoal (written as $!G$) (e.g. obtaining landing permission) to be resolved by further plans.

For example, the following two plans (design option 1) are to achieve goal g, one (P11) when condition c holds and the other (P12) when c does not hold. Both plans involves the execution of a sequence of basic actions (e.g. a_1, a_2, and a_3 in plan P11, and a_1, a'_2, and a'_3 in plan P12).

P11 $g : c \leftarrow a_1;\ a_2;\ a_3$ **Design Option 1**
P12 $g : \neg\ c \leftarrow a_1;\ a'_2;\ a'_3$

Since agents are situated in dynamic environments, it is crucial that they are able to react to changes (that are relevant to the agents' interest) in the environment in a timely fashion. When designing an agent system, reactivity can be maximized in several ways. Firstly, the developers need to make sure that the agent's plan library has plans to cover all environmental possibilities that are significant to the agent's deliberation. For example, if changes in the weather are important, then a reactive agent should have a plan when the weather is fine and another plan when it is not fine. This is related to the completeness of the conjunction of all possible combinations of contextual data. For example, in the first design option above, the two plans completely cover two scenarios relating to the truth value of context condition c. This issue is known in agent-oriented software engineering as the concept of *coverage*, i.e. whether, for a given goal, there will always be some applicable plan to achieve it. Recent work (e.g.

[1] Goals can be declarative or procedural but this has no impact on the results of our approach.

[2] Note that different BDI languages provide different constructs for crafting plans. For example, Jason [5] differentiates between achievement goals and test goals, whereas 3APL [14] distinguishes actions into mental actions, communication actions, external actions, test actions and abstract plans.

[27]) has also refined this coverage concept and defined a numerical measure of the extent to which the set of plans for a goal cover the state space of the environment. In order to increase coverage, we may need to add further plans to an agent's capabilities. There are also other perspectives on agent reactivity, for example the ability of an agent to detect if the currently selected plan is not valid and possibly switch plans. These are however *not* the focus of our current work which we will illustrate in the following scenarios.

An agent's plans should be constructed in such a way that the agent it commits to a certain courses of action as late as possible in achieving its goals, i.e. to wait until the agent has the most updated information about the environment. Therefore, an alternative design for plans to achieve the same goal g is shown below. Note here the use of subgoals as a way to delay the evaluation for the current context.

P21 $g \leftarrow a_1;\ !sg$ **Design Option 2**
P22 $sg : c \leftarrow a_2;\ a_3$
P23 $sg : \neg\ c \leftarrow a_2';\ a_3'$

Both design options would lead to the same ways of achieving goal g, i.e. by performing either $\langle a_1;\ a_2;\ a_3\rangle$ or $\langle a_1;\ a_2';\ a_3'\rangle$. However, in the second design option (i.e. plans P21, P22 and P23 with subgoal sg) the agent does not commit to do either $\langle a_2;\ a_3\rangle$ or $\langle a_2';\ a_3'\rangle$ until a_1 is completed. By contrast, with the first design option the agent makes this commitment earlier. This means in the case if there are any changes in the environment at the time after a_1 is completed (e.g. condition c no longer holds or vice versa), the agent fails to respond to this change (e.g. continues either doing $\langle a_2;\ a_3\rangle$ or $\langle a_2';\ a_3'\rangle$). Therefore, the second design option makes the agent more reactive than the first one does. This example indicates that the number of subgoals in the agent's plans has an impact on how reactive it is at run time.

The use of subgoals is also encouraged in agent design since it decouples a goal from its plan and makes it easy to add other plan choices later. However, just only turning basic actions into subgoals does not merely improve the reactivity of an agents. Let us consider the following design.

P31 $g : c \leftarrow a_1;\ !sg_2;\ a_3$ **Design Option 3**
P32 $g : \neg\ c \leftarrow a_1;\ !sg_2';\ a_3'$
P33 $sg \leftarrow a_2$
P34 $sg' \leftarrow a_2'$

In this third design option, although there are two subgoals in the agent's plans, the actual behaviour of the agent in this example is identical to the one in the initial design. Therefore, only subgoals that have plans with non-empty context conditions affect the reactivity of an agent. However, simply counting the number of subgoals may not reveal the degree of reactivity of an agent programs. Let us consider the following design option.

P41 $g \leftarrow a_1;\ a_2;\ !sg$ **Design Option 4**
P42 $g \leftarrow a_1;\ a_2';\ !sg$
P43 $sg : c \leftarrow a_3$
P44 $sg' : \neg\ c \leftarrow a_3'$

Both design options 2 and 4 are decomposed to the same sequence of actions and have the same number of subgoals with context conditions. The only difference here is how basic actions are grouped into subgoals. In order to differentiate those cases, we need to understand the semantic of the basic actions in terms of how the environment affects an action's executability and how it changes the current environmental context. In addition, we also need to assess whether subgoals occur at the appropriate places in the plan definitions. In the next section, we will describe an abstract BDI programming environment which supports the software engineers in reasoning about the reactivity of an agent program.

3 Abstract BDI programming environment

State-of-the-art BDI agent programming environment (e.g. Jason [5], JACK [6] or Jadex [23]) have only a partial and syntactical understanding of the BDI software system (while the meaning of the software is implicitly interpretable by human developers), and thus they provide limited support to understand and improve the reactivity of an agent program. We believe that in order to provide further value to the software engineers, a BDI programming environment should get a deeper, more semantic understanding of what each agent in the software does and how they operates in the environment. In order to achieve this, the semantics of agent actions and the environment states need to be explicitly specified in such a formal way that they can be automatically interpreted. Note that this information are already, albeit implicitly, available to the software engineers when they develop the agent system. We now define how such information can be formally defined and used for the purpose of improving the reactivity of an agent program.

Action description library

Agents are situated in an environment and thus must be able to act within that environment. As a result, each agent has a set of basic actions which are the basic means of the agent to change its environment. Basic actions define the capabilities that an agent can use to achieve its goals and should be made available to the software engineers when they develop the agent program. The software engineer refers to those basic actions in the program through their symbolic representation (e.g. predicates in AgentSpeak(L) [24]). For example, if the software engineer is programming a robot agent to collect garbages, they need to know the actions the agent is capable of doing (those its hardware allow it to do).

An agent's basic actions are defined and provided to the software engineer in the form of an *action description library.* Each action has an optional precondition[3] which is restricted to a conjunction of belief literals specifying the situation in which the action is executable. An action is executable if the belief literals in its precondition are in the agent's current belief. If the precondition of an action is not satisfied, the action is not executable and thus the plan to which it belongs fails. We have added this clarification to the paper. The description of an action in the action library also specifies the action's effects in terms of adding (i.e. $+b$) or deleting (i.e. $-b$) belief atoms. Some existing BDI programming languages such as the CAN (Conceptual Agent Notation) family of BDI languages [26], 3APL [14], or GOAL [18] also offer a similar STRIPS-like description of agents' actions. In this paper, we assume that all actions are deterministic and our future work will explore to include non-deterministic actions.

Table 1. An example of the action description library

Action	**Precondition**	**Effect**
collect(garbage)	at(robot, Place) $\wedge$ at(garbage, Place)	$-$at(garbage, Place)
moveTowards(NewPlace)	$\neg$ weather(raining) $\wedge$ at(robot, OldPlace)	+at(robot, NewPlace) $-$at(robot, OldPlace)
notify(headquarter)	none	$-$sent(notification)
rainproof(self)	weather(raining)	+resistant(robot, water)

Table 1 shows an example of the action description library for a robot agent. As can be seen, there are four actions the robot can perform: collect a garbage, move to a new location, notify the headquarter agent and rainproof itself. Collecting a garbage, i.e. $collect(garbage)$, has a precondition that both the robot and the garbage are at the same $Place$ and has the effect that the garbage is no longer at the $Place$, i.e. deleting $at(garbage, Place)$ from the belief set. Moving towards a new place, i.e. $moveTowards(NewPlace)$, has a precondition that the weather is not raining and the robot is at the $OldPlace$ (i.e. not at the $NewPlace$), and has the effect that the robot is at the $NewPlace$, i.e. deleting $at(robot, OldPlace)$ and adding $at(robot, NewPlace)$. On the other hand, both $notify(headquarter)$ and $rainproof(self)$ are neutral actions since they do not change the environment, i.e. one sending a message to another agent while the other making itself rainproof. While the action $notify(headquarter)$ has no precondition, the action $rainproof(self)$ is executable only when the weather is raining.

[3] An omitted precondition is equivalent to *true*, i.e. the action can be executed in any situation.

Environmental state model

Since an agent-based system is situated in an environment which it interacts with, an explicit modelling of the environment is very important to the development of an agent system [31]. Existing agent-oriented methodologies do not address this aspect very well [12] and thus the software engineers may find it difficult when it comes to implementation. For example, Gaia [31] models the environment simply, in terms of variables (or tuples) that the agents can read and write (and consume). Prometheus' design [21] has an environment model capturing actors, percepts (inputs from the actor to the agent system) and actions (outputs from the system to actors), whereas INGENIAS' [22] environment viewpoint defines the entities (i.e. resources, other agents, and applications) with which an agent system interacts.

For the purpose of our current work in this paper, we are particularly interested in one important aspect of the environment: how the environment changes in responding to actions performed by agents in the system and other external events (caused by the actors of the system). Changes in the state of the environment may result in updating the agent's beliefs through perception of the environment. Thus, in an (abstract) environmental state model, we use the addition and deletion of belief atoms to describe changes in the sate of the environment. We represent the environment dynamics using a set of rules. In the left-hand side of a particular rule is an agent action (internal to the multi-agent system) or an external event, each of which is annotated with a source (i.e. the agent performing the action or the actor generating the event). In the right-hand side of the rule is a sequence of possible belief updates, reflecting perceived changes in the state of the environment due to the stimuli in the left-hand side. Rules that have external stimuli source (i.e. external events) denote that those environment-state changes can happen at any time regardless of whether the agent is executing any actions or not.

Table 2. An example of the environmental state model

rain[weather] → +weather(raining)
stop-rain[weather] → −weather(raining)
collect(garbage)[robot-agent] → −at(garbage,house)
moveTowards(NewPlace)[robot-agent] → +at(robot, NewPlace) ; −at(robot, OldPlace)

Table 2 shows an example of the environmental state model. The first two rules describe the environment changes from state *raining* to *not raining* or vice versa due to the external stimuli caused by an actor (i.e. the weather in this case). Those changes are non-deterministic which can happen at any time during the agent's execution. By contrast, the last two rules in Table 2 describe changes in the environment due to agent actions. They are extracted directly from the action description library. Note that only actions (e.g. $\mathit{collect}(\mathit{garbage})$ and $\mathit{moveTowards}(\mathit{Place})$) which change the state of the environment are recorded here. The other two actions of the robot agent (i.e. $\mathit{notify}(\mathit{headquarter})$ and $\mathit{rainproof}(\mathit{self})$) do not change the environment. The environment may also have static aspects, e.g. the location of the house, which are never changed and are not captured in the environmental state model.

4 Reactivity measure

In this section, we define a measure of reactivity for a single plan or a set of plans (achieving the same goal) to capture the intuition that in the course of selecting and executing the plan(s), the agent takes into account the latest information from its environment to avoid potential plan failures. Our reactivity measure can be regarded as a static measure since it involves analysing source code, as opposed to dynamic measures which assess the characteristics of the software during execution [3].

Given goal G, let $\mathcal{PL}(G)$ be the set $\{P_1, P_2, ..., P_n\}$ of alternative plans for achieving G. The reactivity of goal G, denoting as $\mathcal{R}(G)$, is the product of the reactivity of each plan in $\mathcal{PL}(G)$. We take the product since any of the applicable plans may be selected to achieve the goal and each plan is independent of each other.

$$\mathcal{R}(G) = \prod_{P \in \mathcal{PL}(G)} \mathcal{R}(P)$$

The reactivity of a plan P, denoting as $\mathcal{R}(P)$, depends on two important factors: the number of *context dependent* actions in the plan and the reactivity of each subgoal in the plan (except the subgoals which are the same as the goal of the plan). An action A in plan P is said to be context dependent if the following conditions hold: (i) A is not the first formulæ appearing in the body of plan P; (ii) the precondition of A, denoting as $\mathit{prec}(A)$, intersects with the context condition of P; and (iii) the truth value of $\mathit{prec}(A)$ may change due to either an external stimuli or actions performed by another agent, or the actions preceding A in plan P. Note that this information can be gathered from the environmental state model and the action description library. For example, action $\mathit{notify}(\mathit{headquarter})$ in plan $\mathit{Pl}1$ below is not a context dependent action since it does not satisfy condition (i). Action $\mathit{moveTowards}(\mathit{house})$ is context dependent since it satisfied all the three conditions – one of its preconditions, $\mathit{weather}(\mathit{raining})$, is part of the context condition of P, and the truth value of $\mathit{weather}(\mathit{raining})$ changes over time as specified in the environment state model. By contrast, the truth value of the precondition of action $\mathit{collect}(\mathit{garbage})$, which is $\mathit{at}(\mathit{garbage}, \mathit{house})$, also changes over time but is not affected by either action $\mathit{notify}(\mathit{headquarter})$ or action $\mathit{moveTowards}(\mathit{house})$ or any external stimuli. Therefore, action $\mathit{collect}(\mathit{garbage})$ is not a context dependent action. Overall, there is only one context dependent action in plan $\mathit{Pl}1$.

Formally, given a plan P, let $\mathcal{AC}(P)$ be the number of context dependent actions in plan P and $\mathcal{SG}(P)$ be the set of subgoals in P, the reactivity of plan P is defined as follows. Note that we again take the product since for a plan to succeed all the sub-goals must be accomplished, and we assume that the reactivity of each sub-goal is independent of each other's reactivity.

$$\mathcal{R}(P) = \frac{1}{1 + \mathcal{AC}(P)} \times \prod_{G \in \mathcal{SG}(P)} \mathcal{R}(G)$$

It can be easily seen that the reactivity of a goal or plan is the range of $(0, 1]$ where 1 represents the maximum reactivity. Let us illustrate how the reactivity measure is computed using the following example. The robot agent has a goal of possessing the garbage which is located at the house. To achieve this goal, it has two plans which are applicable when the weather is raining (plan $Pl1$) or not raining (plan $Pl2$) respectively. The robot is required to notify the headquarter before executing any plan. Plan $Pl1$ involves moving towards the house and collect the garbage, whereas plan $Pl2$ involves making the robot itself rainproof and continuing to try to achieve the goal.

Pl1 $+!has(robot, garbage) : \neg\ weather(raining) \wedge at(garbage, house)$
$\leftarrow notify(headquarter);$
$moveTowards(house);$
$collect(garbage).$

Pl2 $+!has(robot, garbage) : weather(raining) \wedge at(garbage, house)$
$\leftarrow notify(headquarter);$
$rainproof(self);$
$!has(robot, garbage).$

The above plans have some problems in responding to changes in the environment. If the weather is not raining, only plan $Pl1$ is applicable and the agent commits to execute this plan. However, after the first action of this plan is executed (i.e. $notifiy(headquarter)$), it starts raining and thus the agent cannot execute $moveTowards(house)$ since its precondition no longer holds. A similar problem is also found in plan $Pl2$ where the context is evaluated quite early (i.e. well before action *rainproof* is executed), resulting in the agent being less reactive to changes in the environment. Those issues are correctly reflected in the reactivity measure that we have defined. More specifically, the reactivity of goal $+!has(robot, garbage)$ is the product of the reactivity measures of plans $Pl1$ and $Pl2$. As we have explained earlier, there is only one context dependent action, i.e. $moveTowards(house)$, and no subgoal in plan $Pl1$, and thus the reactivity of plan $Pl1$ is 1/2 (i.e. 0.5). Plan $Pl2$ also has only one context dependent action, i.e. $rainproof(self)$, and its only subgoal, i.e. $has(robot, garbage)$, is the same as the plan's goal. Hence, the reactivity of plan $Pl2$ is also 1/2. Therefore, the reactivity of goal $+!has(robot, garbage)$ is $\mathcal{R}(Pl1) \times \mathcal{R}(Pl2) = 0.25$. This value indicates that the set of plans to achieve goal $+!has(robot, garbage)$ are not strongly reactive. In the next section, we will describe a methodology of how they are restructured for reactivity improvement.

5 Restructuring for reactivity improvement

Note that the reactivity measure above does not include the coverage measure which has already specifically addressed in previous work [27]. To maximize the coverage, the software engineer may need to add additional plans (i.e. new capabilities) to cover as many situations in the environment as possible. By contrast, improving the reactivity measure (as defined in the previous section) only involves restructuring an existing body of an agent code (without adding any new capabilities) in order to improve the reactivity of a program. Although this is somewhat similar to the concept of refactoring in the mainstream software engineering, we avoid the use of that term here since refactoring would not change the software's behaviour, whereas restructuring for reactivity improvement may alter an agent's external behaviour (in terms of making it more reactive to environmental changes).

The definition of the reactivity measure indicates how an agent program can be restructured to improve its reactivity. More specifically, to improve reactivity of a goal, we need to improve the reactivity of each of its applicable plans. To improve the reactivity of a plan, we need to reduce the number of context dependent actions in the plan and increase the reactivity of its subgoals. One extreme method is making every action become a subgoal, which results in the agent unnecessarily making evaluation of the environment at every step of execution.

We can however restructure the program in a more systematic and elegant manner. Note that this restructuring process is done at design time and thus does not affect the run-time performance of an agent. The key is to find the context dependent actions in a plan and turn them into subgoals. The following steps are described as follows. Assume that plan P is written as $g : c \leftarrow b_1;\ b_2;\ ...;\ b_n$

1. Starting from b_2, we go through each formulae b_i in the body of plan P and check the following.
2. If b_i is a subgoal and its reactivity measure is less than 1, then we identify the relevant plans for handling b_i and apply the same technique (steps 1 – 6) to restructure those plans.
3. If b_i is an action, we check whether b_i is a context dependent action using information in the action description library and the environmental state model. If b_i is a context dependent action, we do the following.
4. Create a subgoal sg and replace the remaining sequence of actions/subgoals $< b_i, b_{i+1}, ..., b_n >$ with subgoal sg in the plan body. Subgoal sg can be named after what it can achieve, and it has the same arguments as goal g. Plan P is now written as $g : c \leftarrow b_1;\ b_2;\ ...;\ b_{i-1};\ !sg$.
5. Create a plan for handling the newly created subgoal sg with context condition c and the plan body containing the extracted sequence of actions/subgoals. The new plan P' is written as $sg : c \leftarrow b_i;\ b_{i+1};\ ...;\ b_n$.

6. Continue restructuring plan P' following steps $1-5$.

Let us illustrate how the above steps are applied to restructure the set of plans $\{Pl1, Pl2\}$ for achieving goal $has(robot, garbage)$ described in the previous section. For restructuring plan $Pl1$, we first identify that action $moveTowards(house)$ is context dependent. Therefore, we create a new subgoal $!sg1(rotbot, garbage)$ and extract the sequence $\langle moveTowards(house);\ collect(garbage)\rangle$ into a new plan $Pl12$ for handling this subgoal. We then continue trying to restructure plan $Pl12$ but $collect(garbage)$ is not an domain dependent action, and thus the restructuring process for plan $Pl1$ stops here, resulting into two new plans $Pl11$ and $Pl12$. We note that the capabilities of the agent still remain although its behaviour may change towards being more reactive during the course of achieving goal $has(robot, garbage)$. The agent now makes an evaluation of the context when it is more appropriate to do so, i.e. before it performs the $moveTowards(house)$ action (see plan $Pl12$).

Pl11 $\quad +!has(robot, garbage) : \neg\ weather(raining) \wedge at(garbage, house)$
$\qquad\qquad \leftarrow notify(headquarter);$
$\qquad\qquad\quad !sg1(rotbot, garbage).$

Pl12 $\quad +!sg1(rotbot, garbage) : \neg\ weather(raining) \wedge at(garbage, house)$
$\qquad\qquad \leftarrow moveTowards(house);$
$\qquad\qquad\quad collect(garbage).$

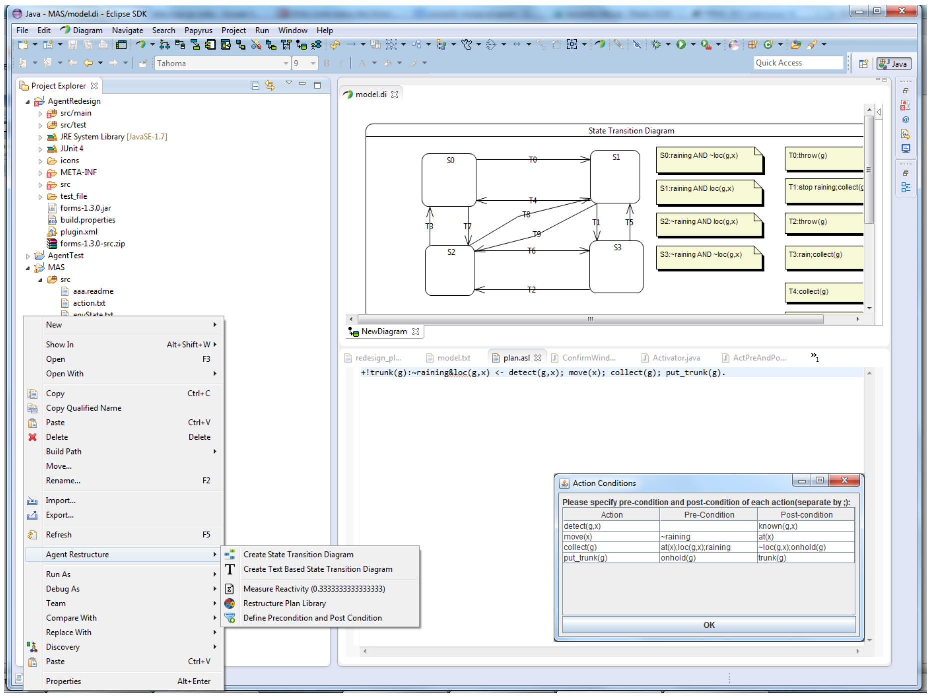
Since plan $Pl11$ no longer has any context-dependent action, its reactivity is equal to the reactivity of subgoal $sg1$, which is equal to the reactivity of plan $Pl12$. Plan $Pl12$ also does not have any context dependent action and thus its reactivity is 1. Similarly, plan $Pl2$ are restructured into two plans $Pl21$ and $Pl22$ as follows. The reactivity of both plans $Pl21$ and $Pl22$ are also 1 since they no longer have any context dependent action. Hence, the restructuring has improved the reactivity of achieving goal $has(robot, garbage)$ from 0.25 to the maximum 1. In the restructured program, the agent make a choice of moving to collect the garbage or rainproofing itself later (i.e. after notifying the headquarter agent) to consider the latest weather information.

Pl21 $\quad +!has(robot, garbage) : weather(raining) \wedge at(garbage, house)$
$\qquad\qquad \leftarrow notify(headquarter);$
$\qquad\qquad\quad !sg2(rotbot, garbage).$

Pl22 $\quad +!sg2(rotbot, garbage) : weather(raining) \wedge at(garbage, house)$
$\qquad\qquad \leftarrow rainproof(self);$
$\qquad\qquad\quad !has(robot, garbage).$

Finally, since the two pairs of plans $Pl11$ and $Pl21$, and plans $Pl12$ and $Pl22$ are substantially similar, we can further restructure $Pl11$ and $Pl21$, into one plan by simply renaming goal $sg2$ to $sg1$.

6 Implementation



```
+!trunk(g):~raining&loc(g,x) <- detect(g,x); move(x); collect(g); put_trunk(g).
```

Please specify pre-condition and post-condition of each action(separate by ;):

Action	Pre-Condition	Post-condition
detect(g,x)		known(g,x)
move(x)	~raining	at(x)
collect(g)	at(x);loc(g,x);raining	~loc(g,x);onhold(g)
put_trunk(g)	onhold(g)	trunk(g)

Fig. 1. A screenshot of our prototype tool

We have developed a prototype implementation of our framework. The tool was implemented as a plugin[4] on the Eclipse platform and thus can be run together with the Eclipse-based code editor of Jason[5], one of the most well-known platforms for developing agent applications in AgentSpeak. Figure 1 shows a screenshot of the tool which is being used. The tool has all the features of the framework that we have proposed in this paper. These features are grouped under the "Agent Restructure" menu item when the user right-clicks on an agent project (see the right-hand-side part of Figure 1).

The tool provides a user interface which allows the user to establish an action description library in terms of defining actions and their preconditions and effects. In addition, the tool is able to parse an existing agent program and populate the library with actions. This feature is important since it allows for our tool to be used with existing agent programs. The tool allows the user to define an environmental state model as either text-based or graphical-based (UML

[4] The tool is available at `https://code.google.com/p/agent-redesign`.
[5] `http://jason.sourceforge.net/Jason/`

state machine diagram). In addition, the tool is able to computes the reactivity measure of an agent program. Finally, the tool supports the user to interactively restructure the current program for reactivity improvement based on our methodology described in the previous section.

7 Related Work

During the past decade, there has been a proliferation of agent-oriented design methodologies [17] and programming languages [4] proposed in the literature. Some of them have become mature and been extensively used in both academic and industrial settings. Unfortunately, there have been however not many efforts in developing software metrics specifically for agent systems. Some early work by Barber and Martin [19] proposes an approach to measure an agent's autonomy in terms of the goals they pursue. Padgham and Winikoff [21] proposed the use of data coupling metric for the identification of agents, as part of Prometheus, an agent-oriented methodology. The work in [7] proposes an approach to evaluate how an agent-oriented methodology supports the pro-activeness of agents in the context of comparing a number of agent-oriented methodologies. Their measure is however coarse-grained and is not suitable for specific agent programs.

To the best our knowledge, there has been no work in measuring the reactivity of an agent program. Recent work by Alonso et. al. [1, 2] which proposes a set of metrics to measure the pro-activeness, autonomy and sociability of agent programs is perhaps the most closely related to ours. In particularly, one of the component in their pro-activeness metric is reaction, which is measured in terms of the number of requests from the environment that the agent can respond and the complexity of the agent operations. The former is similar to the ideas of the agent's ability to handle external events as in our reactivity metric. The latter is however not clearly defined in their work. One clear difference between their work and ours is that their metrics are applied to general-purpose agents while our work is specific to BDI agents (as we deal specifically with plans, events, and context conditions), the most well-known and popular agent types. More recently, Thangarajah et. al. [27] have proposed to measure plan coverage and overlap for BDI agents. Coverage indicates whether there will always be some plan with a matching context condition, whereas overlap refers to whether there is, in some situations, more than one plan that is applicable. Plan coverage are counted using model counting to measure the portion of the number of models in which a set of plans are applicable using their context conditions against the total number of models in the domain of concern. This is related to an agent's reactivity in terms of having plans to respond to different scenarios of the environment's changes. However, in order to increase the plan coverage, we may need to alter the existing context condition and/or adding further plans to the agent's capabilities. This is different from the methodology we have proposed in this paper. Some existing techniques that use planning to either generate new plans on the fly (e.g. [20]) or to perform lookahead of plan execution (e.g. [25]) are loosely related to our work. Unlike our approach, such techniques however require substantial changes and/or extensions made to the existing BDI programming languages and platforms.

Due to its maturity, the object-oriented software engineering literature has a substantial amount of work in software metrics. Such work (e.g. [8], [3]) target at defining metrics for classes and objects by examining their functions, attributes and the relationships between them. Although these concepts are specific to objects and cannot be applied directly to agents, our work is inspired and built up the notion and ideas from object-oriented software metrics. Our methodology of restructuring agent programs is also inspired by code reactoring [15] in mainstream software engineering and is part of the body of work that supports the maintenance and evolution of agent systems [9–11, 13]. Code refactoring refers to a set of techniques for restructuring an existing code to modify its internal structure without altering its external behaviour. Code refactoring is specifically for improving some of the nonfunctional attributes of the software including code readability, maintainability of the source code, and extensibility of the internal architecture. However, there has been very little work on refactoring for agent systems. The work in [28] identifies some common "bad smells" problems (e.g. duplicated behaviour structure and big plans) in multi-agent system design and proposes some refactoring patterns to eliminate those bad smells. Their work however only focuses on design models rather source code as in our work.

8 Conclusions

In this paper, we have argued that there are a number of factors that affect the reactivity of BDI agent systems, one of which is the ability to delay their commitment to a certain courses of action as late as possible (i.e. wait until the agent has the most updated information about the environment) by preferring subgoals over primitive actions. Based on this notion, we have developed a framework which facilitates the restructuring of a body of code of an BDI agent program in order to improve its reactivity. A novel aspect of the framework is a reactivity measure which reflects whether a particular choice of actions is left as late as possible so as to consider the latest information in the environment. The framework is built upon a programming environment where the semantic of agent actions and changes in the environment are explicitly defined. We also propose a methodology of how to improve the reactivity measure of an agent program. Although our approach is generally applicable to any BDI programming languages, we implemented a restructuring plug-in integrated with the Eclipse-based development environment of the well-known Jason platform.

In terms of future work, we plan to conduct an evaluation of our reactivity metric and methodology. In addition, we plan to investigate some dynamic measures for agents' reactivity, i.e. measure the reactivity of the agents during execution, and compare the reliability of both static and dynamic measures as well as explore how they can complement each other. We also plan to explore how to provide a more complex model of the environment. Our future work also involves implementing our approach and methodology to support other BDI programming languages, especially those that allow STRIPS-like specification of actions such as 3APL [14]. Finally, a long term plan of our work would involves proposing a metric suite for agent systems including not only reactivity but also other agent properties such as pro-activeness, autonomy and sociability, and developing a comprehensive set of discipline techniques to improve those quality aspects of an agent system.

References

1. F. Alonso, J. L. Fuertes, L. Martinez, and H. Soza. Towards a set of measures for evaluating software agent autonomy. In *Proceedings of the 2009 Eighth Mexican International Conference on Artificial Intelligence*, MICAI '09, pages 73–78, Washington, DC, USA, 2009. IEEE Computer Society.
2. F. Alonso, J. L. Fuertes, L. Martinez, and H. Soza. Measuring the pro-activity of software agents. *International Conference on Software Engineering Advances*, 0:319–324, 2010.
3. G. Barnes and B. Swim. Inheriting software metrics. *Journal of Object-Oriented Programming*, 6(7):27–34, November - December 1993.
4. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, 2005.
5. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley, 2007. ISBN 0470029005.
6. P. Busetta, N. Howden, R. Rönnquist, and A. Hodgson. Structuring BDI agents in functional clusters. In *Agent Theories, Architectures, and Languages (ATAL-99)*, pages 277–289. Springer-Verlag, 2000. LNCS 1757.
7. L. Cernuzzi and G. Rossi. On the evaluation of agent oriented modeling methods. In *Proceedings of Agent Oriented Methodology Workshop*, Seattle, November 2002.
8. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20:476–493, June 1994.
9. H. K. Dam and A. Ghose. Automated change impact analysis for agent systems. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 33–42, Washington, DC, USA, 2011. IEEE.
10. H. K. Dam and A. Ghose. Supporting change impact analysis for intelligent agent systems. *Science of Computer Programming*, 78(9):1728 – 1750, 2013.
11. H. K. Dam and M. Winikoff. An agent-oriented approach to change propagation in software maintenance. *Journal of Autonomous Agents and Multi-Agent Systems*, 23(3):384–452, 2011.
12. H. K. Dam and M. Winikoff. Towards a next-generation AOSE methodology. *Science of Computer Programming*, 78(6):684–694, June 2013.
13. K. H. Dam and M. Winikoff. Cost-based BDI plan selection for change propagation. In Padgham, Parkes, Müller, and Parsons, editors, *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 217–224, Estoril, Portugal, May 2008.
14. M. Dastani, M. Birna Riemsdijk, and J.-J. Meyer. Programming multi-agent systems in 3APL. In R. Bordini, M. Dastani, J. Dix, and A. Fallah Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 39–67. Springer US, 2005.
15. M. Fowler and K. Beck. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

16. M. Georgeff and A. Rao. Rational software agents: From theory to practice. In *Agent Technology: Foundations, Applications, and Markets*, chapter 8, pages 139–160. 1998.
17. B. Henderson-Sellers and P. Giorgini, editors. *Agent-Oriented Methodologies*. Idea Group Publishing, 2005.
18. K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Meyer. Agent programming with declarative goals. In *Intelligent Agents VI - Proceedings of the 7th International Workshop on Agent Theories, Architectures, and Languages (ATAL'2000)*. Springer Verlag, 2001.
19. C. E. Martin, K. S. Barber, and K. S. Barber. Agent autonomy: Specification, measurement, and dynamic adjustment. In *In Proceedings of the Autonomy Control Software Workshop, Agents '99*, pages 8–15, 1999.
20. F. Meneguzzi and M. Luck. Composing high-level plans for declarative agent programming. In *Proceedings of the 5th international conference on Declarative agent languages and technologies V*, DALT'07, pages 69–85, 2008.
21. L. Padgham and M. Winikoff. *Developing intelligent agent systems: A practical guide*. John Wiley & Sons, Chichester, 2004. ISBN 0-470-86120-7.
22. J. Pavon, J. J. Gomez-Sanz, and R. Fuentes. The INGENIAS methodology and tools. In B. Henderson-Sellers and P. Giorgini, editors, *Agent-Oriented Methodologies*, chapter IX, pages 236–276. Idea Group Publishing, 2005.
23. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming*, pages 149–174. Springer Science+Business Media Inc., USA, 9 2005. Book chapter.
24. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. Perrame, editors, *Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, pages 42–55. Springer Verlag, 1996. LNAI, Volume 1038.
25. S. Sardina, L. de Silva, and L. Padgham. Hierarchical planning in bdi agent programming languages: a formal approach. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 1001–1008, 2006.
26. S. Sardina and L. Padgham. A BDI agent programming language with failure recovery, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70, 2011.
27. J. Thangarajah, S. Sardina, and L. Padgham. Measuring plan coverage and overlap for agent reasoning. In *Proceedings of the 11th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2012)*, pages 1049–1056, Valencia, Spain, June 2012.
28. A. M. Tiryaki, E. E. Ekinci, and O. Dikenelli. Refactoring in multi agent system development. In *Proceedings of the 6th German conference on Multiagent System Technologies*, MATES '08, pages 183–194, 2008.
29. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, pages 470–481, Toulouse, France, 2002.
30. M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons (Chichester, England), 2002. ISBN 0 47149691X.
31. F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, 2003.