Supporting change impact analysis for intelligent agent systems

Hoa Khanh Dam[a], Aditya Ghose[a]

[a]*School of Computer Science and Software Engineering*
University of Wollongong, Australia.

Abstract

Software maintenance and evolution is an important and lengthy phase in the software life-cycle which can account for as much as two-thirds of the total software development costs. Intelligent agent technology has evolved rapidly over the past few years as evidenced by the increasing number of agent systems in many different domains. Intelligent agent systems with their distinct characteristics and behaviours introduce new problems in software maintenance. However, in contrast to a substantial amount of work in providing methodologies for analysing, designing and implementing agent-based systems, there has been very little work on maintenance and evolution of agent systems. A critical issue in software maintenance and evolution is change impact analysis: estimating the potential effects of changes before they are made as an agent system evolves. In this paper, we propose two distinct approaches to change impact analysis for the well-known and widely-developed Belief-Desire-Intention agent systems. On the one hand, our static technique computes the impact of a change by analysing the source code and identifying various dependencies within the agent system. On the other hand, our dynamic technique builds a representation of an agent's behaviour by analyzing its execution traces which consist of goals and plans, and uses this representation to estimate impacts. We have implemented both techniques and in this paper we also report on the experimental results that compare their effectiveness in practice.

Email addresses: hoa@uow.edu.au (Hoa Khanh Dam), aditya@uow.edu.au (Aditya Ghose)

URL: http://www.uow.edu.au/∼hoa/ (Hoa Khanh Dam), http://www.uow.edu.au/∼aditya/ (Aditya Ghose)

Keywords: change impact analysis, multi-agent systems

1. Introduction

Software maintenance and evolution is an important and lengthy phase in the software life-cycle which can account for as much as two-thirds of the total software development costs [1, page 449]. In fact, a substantial proportion of the resources expended within the Information Technology industry goes towards the maintenance of software systems. Annual software maintenance cost in the United States had been estimated to be more than $70 billion [2]. A recent prediction [3] has indicated that by the year 2020 more than 60% of software developers will be working on software maintenance and evolution. This is mainly due to the fact that the ever-changing business environment demands constant and rapid evolution of software, and consequently change is inevitable if software systems are to remain useful.

A software agent [4] is an *autonomous* computational entity being *situated* in an environment and being able to operate independently in terms of making its own decisions about which activities to pursue. An agent has goals which it is able to pursue over time, and at the same time it can respond in a timely fashion to changes that occur in the environment it operates. An agent is also able to interact with other agents in order to accomplish its goals. Such useful notions of intelligent agents have made them a popular choice for developing software in a number of areas. In fact, the practical utility of agents has been demonstrated in a wide range of domains such as air traffic control, space exploration, weather alerting [5], business process management [6], holonic manufacturing [7], e-commerce and information management [8, 9]. This number continues to increase since there are compelling reasons to use intelligent agent technology such as its proven ability to significantly improve the development of complex systems in a broad range of areas (over 350% productivity increase in one substantial study [10]).

Agent systems, like conventional software systems, will evolve and will need to be maintained throughout their life to meet ever-changing user requirements and environment changes. Agent systems are, however, different from classical systems since they have distinct concepts (e.g. plans, beliefs, goals, events, etc.) and architectures (e.g. the Belief Desire Intention architecture [11]). *However, there has been very little work on providing support for the maintenance and evolution of agent systems.* Since intelligent agents

are a relatively new technology, maintenance of agent-based systems has not been so far a critical issue. However, if we are to be successful in the long-term adoption of agent-oriented development of software systems that remain useful after delivery, it is now crucial for the research community to provide solutions and insights that will improve the practice of maintaining and evolving agent systems.

Software maintenance and evolution activities are usually classified as adaptive maintenance (changing the system in response to changes in its environment so it continues to function), corrective maintenance (changing the system to fix errors), and perfective maintenance (changing the system's functionality to meet changing needs). Therefore, dealing with changes is central in software maintenance and evolution and includes two key aspects [12]: *change impact analysis* – predicting the potential consequences of a proposed change; and *change propagation* – implementing a change by propagating changes to maintain consistency within the software. Previous work [13] has proposed a framework that supports change propagation in the evolution of agent-oriented design models (i.e. Prometheus [14]). That framework has also been extended to deal with other model types (e.g. UML models in [15], enterprise architecture models [16], or service-oriented architecture models [17]).

The focus of this paper is on change impact analysis of agent systems. Change impact analysis [18] usually starts with the software maintainer examining the change request and determining the entities initially affected by the change (i.e. the *primary changes*). The software maintainer then determines other entities in the system that have potential dependency relationships with the initial ones, and forms a set of impacts. Those impacted components also relate to other entities and thus the impact analysis continues this process until a complete impact set is obtained. Change impact analysis plays a major part in planning and establishing the feasibility of a change in terms of predicting the cost and complexity of the change (before implementing it). This help reduces the risks associated with making changes that have unintended, expensive, or even disastrous effects on an existing system. Furthermore, change impact analysis can be used to predict or identify parts of a system that will need to be retested (i.e. regression testing) as a result of changes.

The importance of the change impact analysis problem has led to substantial work on proposing specific impact analysis techniques. Although notions and ideas from a large body of work addressing change impact anal-

ysis for classical software systems (e.g. [18, 19, 20]) can be adapted, agent systems with their distinct characteristics and architectures introduce new problems in software maintenance. For instance, while object-oriented software deals with classes, methods and fields, a typical agent-based software, e.g. the Belief-Desire-Intention (BDI) [11] agents, consists of agents, plans, events/goals and beliefs. In addition, existing dynamic impact techniques (e.g. the well-known PathImpact [20]) tend to consider a linear, successful execution of a program, whereas an agent's execution may contain parallelisation (e.g. achieving two goals concurrently), interruption (e.g. suspending an executing plan to deal with higher priority events), and failures handling (e.g. trying alternative plans in pursuing a goal). As a result, there is a strong need to have impact analysis techniques (and tools) that specifically support agent systems and their distinct characteristics. Since most of agent programs (e.g. AgentSpeak[1] [21]) are logic programs, other closely related work is in the area of software maintenance for logic programs. Most of the work in this area (e.g. [22]) however, focuses on program slicing. In fact, a recent proposal by Bordini et. al. [23] also adopts the approach in [22] to slice AgentSpeak programs for model checking purposes. To the best of our knowledge, there is however no work in the area of change impact analysis for agent systems at the source code level.

In this paper[2], we present two different approaches to change impact analysis of an agent program: one based on analysis of agent source code (i.e. static impact analysis), and the other on analysis of execution logs of an agent program (i.e. dynamic impact analysis). Both approaches specifically on agent systems, in particular the well-known and widely-used BDI agents written in the AgentSpeak programming language. Our static approach involves the development of a classification of various dependencies existing in an agent system (including within an agent and between agents) and a taxonomy of changes to an agent system. Our static approach has access to source code to build up a dependency graph representing the agent system and use this to compute impacts. In contrast, our dynamic approach calculates the impact of a change using dynamic information which is collected from execution data for a specific set of agent executions (e.g. executions based on an

[1]The language was originally called AgentSpeak(L), but is commonly referred to as AgentSpeak.

[2]Earlier versions of this paper only focus on static impact analysis [24] and outline some preliminary ideas of an approach to dynamic change impact analysis [25].

operational profile or executions of test suites). Such dynamic information contains two key aspects determining the behaviour of a BDI agent system: the *goals* an agent pursued and the *plans* it deployed to achieve those goals. The dynamic technique analyse that information to determine when a plan or goal is changed, what other plans and goals are potentially impacted by the change. Dynamic analysis results tend to be more practically useful (compared to those of static analysis) since they better reflect how the system is actually being used, and consequently do not have computed impacts derived from impossible system behaviour (which is the case for some static analysis approaches). We have performed an empirical validation using two real agent systems to compare the effectiveness of both approaches. It should be emphasized that although Jason [26] and AgentSpeak is our setting, as will be seen later ideas from our approach can be adapted to other agent programming languages and extended to address change impact analysis in agent design models.

The paper is organised as follows. We begin with background on the AgentSpeak programming language and the Belief-Desire-Intention architecture that it is built upon (section 2). Our static and dynamic impact analysis techniques are described in section 3 and 4 respectively. An empirical validation of both techniques is described in section 5. Finally, we discuss related work in section 6, and then conclude and outline some directions for future work in section 7.

2. Background

2.1. Belief-Desire-Intention (BDI) agents

Since the 1980s, the field of intelligent agent technology has attracted a substantial amount of interest from both academia and industry. The *Belief-Desire-Intention* architecture is one of the most well-established and widely-used agent models. Many agent systems, especially those situated in complex and dynamic environments with real-time reasoning and control requirements, have been developed based on the most well-established and widely-used *Belief-Desire-Intention* (BDI) agent architecture [11]. The BDI family of agent theories, languages and systems are inspired by the philosophical work of Bratman [27] about how humans do resource bounded practical reasoning, i.e. figure out what to do and how to act under limited resource capacity. An agent's beliefs represent information about the environment,

the agent itself, or other agents, from the agent's perspective. Desires represent the objectives to be accomplished in terms of states of the world that the agent wants to reach. Intentions represent the currently chosen courses of action to pursue a certain desire that the agent has committed.

Those theoretical ideas of the BDI model have been modified to suit a practical computational environment. In fact, while BDI theories focus on desires and goals, BDI implementations (e.g. [11]) deal with *events*. Events are significant occurrences that the agent should respond to in some way. Some BDI agent implementation platforms (e.g. Jason) model the change associated with the adoption of new *(sub)goals* as events. Furthermore, in BDI implementations intentions are viewed as the *plans* which are currently being executed by the agent.

A practical goal-oriented BDI-style agent is basically a reactive planning system which selects and executes plans to achieve its *goals* or *events* (e.g. landing an unmanned aircraft) in a systematic manner. Such plans are selected from the agent's plan library, which is a collection of pre-defined plans representing the agent's procedural knowledge of the domain in which it operates (e.g. alternative plans to landing an aircraft). Each plan has a *context condition* which defines the situation in which the plan is *applicable*, i.e. it is sensible to use the plan in a particular situation (e.g. a certain weather condition). For example, an unmanned aerial vehicle (UAV) agent controller may have some plans for landing an aircraft which may only be applicable under normal weather conditions and some other plans to be deployed only under emergency situations. The determination of a plan's applicability involves checking whether the plan's context condition holds in the current moment in time (i.e. making choices as late as possible). The agent selects one of the applicable plans to execute. This would involve executing its body which may contain a sequence of primitive actions (e.g. retracting the flaps) and subgoals (e.g. obtaining landing permission from air control tower), which can trigger further plans.

We describe here a typical execution cycle that implements the decision-making of an agent following an implementation of the BDI architecture. The cycle can be viewed as consisting of the following steps, shown in figure 1:

1. An event is received from the environment, or is generated internally by belief changes or plan execution. The agent responds to this event by selecting from its plan library a set of plans (P_1 - P_k) that are relevant

(i.e. match the invocation condition) for handling the event (by looking at the plans' definition).

2. The agent then determines the subset of the relevant plans (P_m - P_n) that is applicable in terms of handling the particular event. The determination of a plan's applicability involves checking whether the plan's context condition holds in the current situation.

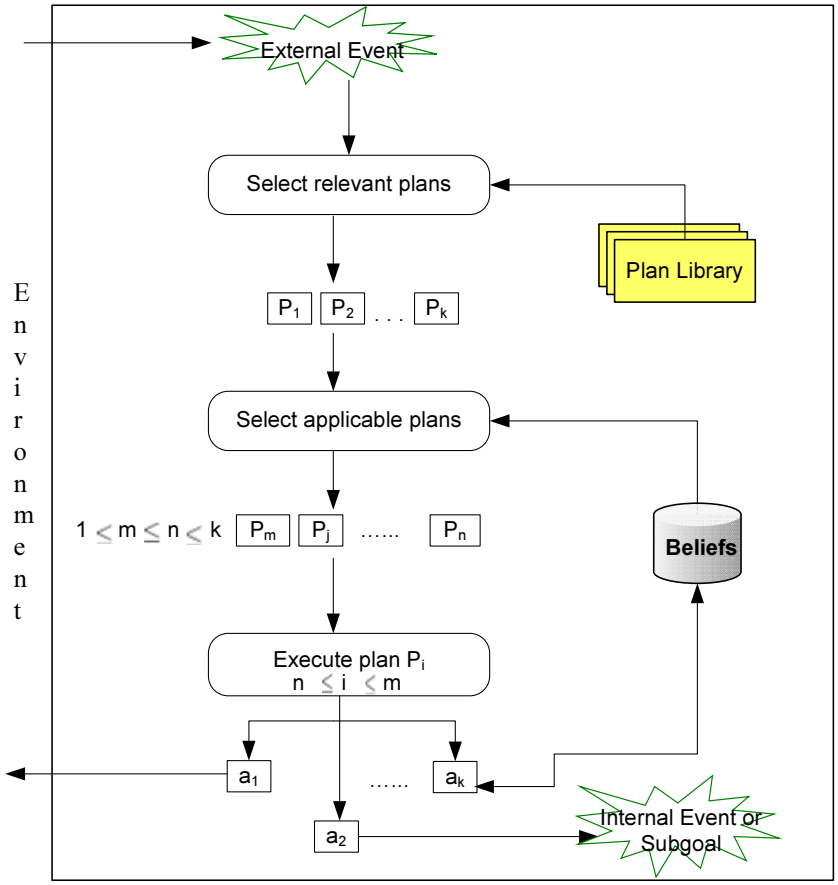

Figure 1: A typical BDI execution cycle

3. The agent selects one of the applicable plans (e.g. P_i). This may be based on certain pre-determined priority (e.g. the first applicable plan is selected in JACK's [28] default mechanism) although more sophisticated mechanisms can also be used, depending on the implementation.
4. The agent then executes the selected plan (e.g. P_i) by performing its actions and sub-goals (a_1 - a_k). The actions can be modifying or

querying the agent's beliefs, raising new events, or interacting with the environment.

A plan can be successfully executed, in which case the (sub-)goal is regarded to have been accomplished. Execution of a plan, however, can fail in some situations, e.g. a sub-goal may have no applicable plans, or an action can fail, or a test can be false. In these cases, if the agent is attempting to achieve a goal, a mechanism that handles failure is used. Typically, the agent tries an alternative applicable plan for responding to the triggering event of the failed plan. It is also noted that failures propagate upwards through the event-plan tree: if a plan fails its parent event is re-posted; if this fails then the parent of the event fails and so on.

2.1.1. AgentSpeak and Jason

Among many agent programming languages (refer to [29] for an overview), some of the most widely used rely on the well-known Belief-Desire-Intention (BDI) agent architecture. For this reason, we have chosen Jason, an extension of AgentSpeak [21] – one of the most popular BDI agent-oriented languages. Jason is also a well-known and widely-used platform for the development of multi-agent systems. In addition, several open-source agent systems (with different *versions*) developed using Jason are publicly available, which is critical for our evaluation purpose. In this section, we will briefly describe the BDI architecture and the syntax of AgentSpeak.

AgentSpeak is an agent-oriented programming language developed based on the BDI architecture. Figure 2 describes the grammar of an agent specification in AgentSpeak. An agent ag is specified by a set of beliefs bs and a set of plans ps (the agent's plan library). The belief set consists of a number of *belief literals*, each of which is in the form of a predicate P over the first order terms (i.e. $t_1, \ldots, t_n$).

A plan p in AgentSpeak is specified by a triggering event te, a context condition ct, and a plan body h. A triggering event can be the addition (i.e. $+at$) or the deletion (i.e. $-at$) of a belief from an agent's belief base, or the addition (i.e. $+g$) or the deletion of a goal (i.e. $-g$). A context condition is a Boolean formula of an agent's belief literals. The plan body contains a sequence of actions (i.e. action symbol $A(t_1, \ldots, t_n)$), goals (i.e. g) and belief updates ($+at$ for adding and $-at$ for removing). Goals can be either *achievement goals* (i.e. $!at$, indicating the agent wants to achieve a state where at is a true belief) or *test goals* (i.e. $?at$, indicating the agent wants

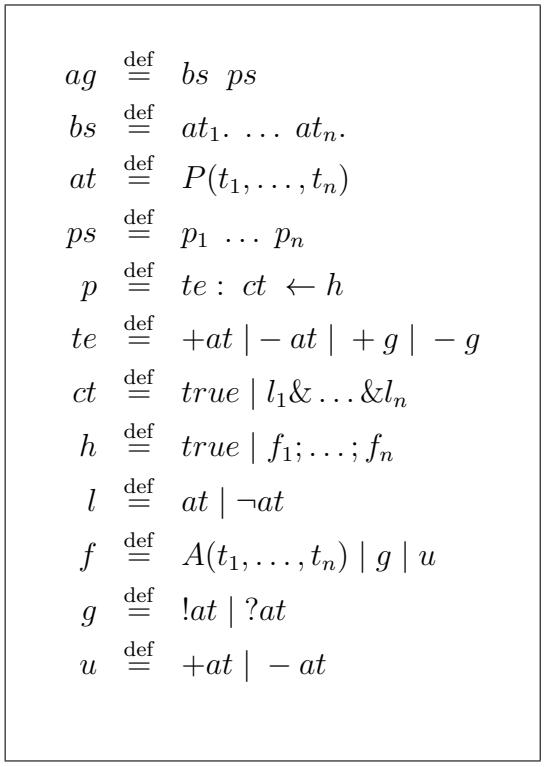

Figure 2: The concrete syntax of AgentSpeak(L) (adapted from [23])

to test whether *at* is a true belief or not).

Jason [26] is one of the most well-known platform for the development of multi-agent systems using AgentSpeak. Jason also slightly extends AgentSpeak in a number of ways such as having annotations for atomic formulae, Prolog-like rules in the belief base, labels for plans, and so on. A particular extension of AgentSpeak in Jason that we are interested in and is supported by our change impact analysis framework is agent communication since it demonstrates the dependencies between agents in a multi-agent system.

2.2. Example

The running example that we use in this paper is adapted from a simple agent system[3] that consists of two robots collecting garbage on planet Mars,

[3]The source code of the original agent program is available on the Jason project website `http://jason.sourceforge.net`

which is represented as a territory grid. The first robot (i.e. r1) is responsible for looking for garbage and delivering them to the second robot (i.e. r2) where they are burnt. If robot $r1$ finds a piece of garbage, the robot picks it up, delivers it to the location of $r2$ and drops the garbage there. Robot $r1$ then returns to the location where the last garbage was found and continues the search from that location. We modify the original example slightly: instead of having robot $r2$ placed at a fixed location, we allow it to move around. Therefore, when robot $r1$ finds a piece of garbage, it sends a message to robot $r2$ to ask for its current location. Robot $r2$ then tells $r1$ its location and stays there to wait for $r1$ to deliver a piece of garbage. This modification is to demonstrate the communication taking place between the two agents. The AgentSpeak/Jason code for the two agents are presented as follows.

`Agent r1`
Beliefs:
checking(slots).
pos(r2,2,2).
Plans:
+pos(r1,X1,Y1) : checking(slots) ∧ ¬garbage(r1) (P1)
← next(X1,Y1).
+garbage(r1) : checking(slots) (P2)
← !stop(check);
.send(r2, askOne, pos(r2,X2,Y2), pos(r2,X2,Y2));
.abolish(pos(r2,_,_));
+pos(r2,X2,Y2);
pick(garb);
!go(r2);
drop(garb);
!continue(check).
+!stop(check) : true (P3)
← ?pos(r1,X1,Y1);
+pos(back,X1,Y1); // remember where to go back
–checking(slots).
+!continue(check) : true (P4)
← !go(back); // goes back and continue to check
?pos(back,X1,Y1);
–pos(back,X1,Y1);
+checking(slots);

```
        next(X1,Y1).
+!go(R) : pos(R,X1,Y1) & pos(r1,X1,Y1)                    (P5)
    ← true.
+!go(R) : true                                            (P6)
    ← ?pos(R,X1,Y1);
        moveTowards(X1,Y1);
        !go(R)
```

The first agent $r1$ has one initial belief that it is checking all the slots in the grid for garbage (i.e. $checking(slots)$) and one initial belief about the location of robot $r2$ (i.e. $pos(r2, 2, 2)$). The agent however has 6 different plans in its plan library. The first plan $P1$ is triggered when the agent perceives that it is in a new position $(X1, Y1)$[4]. If it is currently in the mode of checking for garbage and no garbage is perceived in that location, it then moves the robot to the next slot in the grid by performing the primitive action $next(X1, Y1)$ where $(X1, Y1)$ is its current position. The second plan $P2$ is triggered when robot $r1$ perceives garbage in its location. If the robot is currently in the mode of checking for garbage, this plan would be executed as follow. First, the robot stops checking for garbage by posting a subgoal $stop(check)$. Second, the robot $r1$ sends a message to $r2$ and asks for its current location. Robot $r1$ then picks the garbage (i.e. primitive action $pick(garb)$), goes to the location of robot $r2$ (i.e. subgoal $go(r2)$) and drops the garbage at the slot where $r2$ is currently located (i.e. primitive action $drop(garb)$).

Plan $P3$ is the only relevant plan to achieve subgoal $stop(check)$ and it is always applicable since its context condition is always true. Following this plan, robot $r1$ first retrieves its current location from the agent's belief base by posting a test goal $?pos(R, X1, Y1)$. It then records this position its belief (so that it can go back to this location later) by adding a belief $pos(back, X1, Y1)$ to its belief base. The agent also indicates that it is not in the mode of searching for garbage by deleting belief $checking(slots)$ from its belief base. Plan $P4$, on the other hand, is used for the agent to go back to its previous location (after delivering the garbage to agent $r2$) and continue searching for garbage.

[4]Note that lower-case terms denote constants while upper-case terms are regarded as variables.

Plans $P5$ and $P6$ are used for accomplishing the goal of going to a specific location on the grid where R is located. According to plan $P6$, the agent first gets the position $(X1, Y1)$ of R from its belief base, then moves itself towards that position (by performing a primitive action $moveTowards(X1, Y1)$), and continues going towards R (by posting subgoal $go(R)$ recursively). According to plan $P5$, if agent $r1$ is already at the position of R, it would do nothing in order to achieve the goal of going towards R. This plan is to terminate the recursion as in plan $P6$.

`Agent r2`
Beliefs:
moving(slots).
Plans:
+pos(r2,X2,Y2) : moving(slots) (P7)
← next(X2,Y2).
+?pos(r2,X2,Y2) : moving(slots) (P8)
← –moving(slots).
+garbage(r2) : true (P9)
← burn(garb);
?pos(r2,X2,Y2);
+moving(slots);
next(X2,Y2).

Above is the code for the second robot. Its initial belief base has a single predicate indicating that it is in the mode of moving around the grid. The agent has three plans in its plan library. When the agent perceives that it is in a new position and it is currently in the moving mode, plan $P7$ is executed which moves the robot to the next slot. When robot $r2$ receives a message from $r1$ asking for $r2$'s current position (refer to plan $P2$ of agent $r1$), a test goal $+?pos(r2, X2, Y2)$ is generated within $r2$. Plan $P8$ is used to achieve that test goal[5] by removing $moving(slots)$ from agent $r2$'s belief base, indicating that the robot stays at the current location (to wait for the delivery of garbage from robot $r1$). Finally, when robot $r2$ perceives a garbage on its location, it executes plan $P9$ which burns the garbage, changes to the moving mode, and moves to the next slot.

[5]Note that when receiving the message from robot $r1$, $r2$ replies with its current location. This is a default action as implemented in Jason and we do not need to explicitly specify in the code.

3. Static impact analysis

In this section, we describe our approach to static change impact analysis for an agent program (see figure 3). The process of impact analysis starts with the software engineer examining the change request, identifying the entities (e.g. agents, plans, goals, beliefs, etc.) in the existing agent program (i.e. AP in figure 3) initially affected by the change, and making changes to those entities. Such primary changes result in a new version AP' of the agent program. Our impact analysis technique then compares the two versions and automatically detects all the primary changes previously performed. This would eliminate the overhead of specifying each and every change by the software engineer. Our impact analysis technique also automatically classifies those changes using a taxonomy of changes on an agent system. The automatic classification of the changes into atomic changes enables a precise impact analysis – any change in an agent program is a set of instances of change types in the taxonomy.

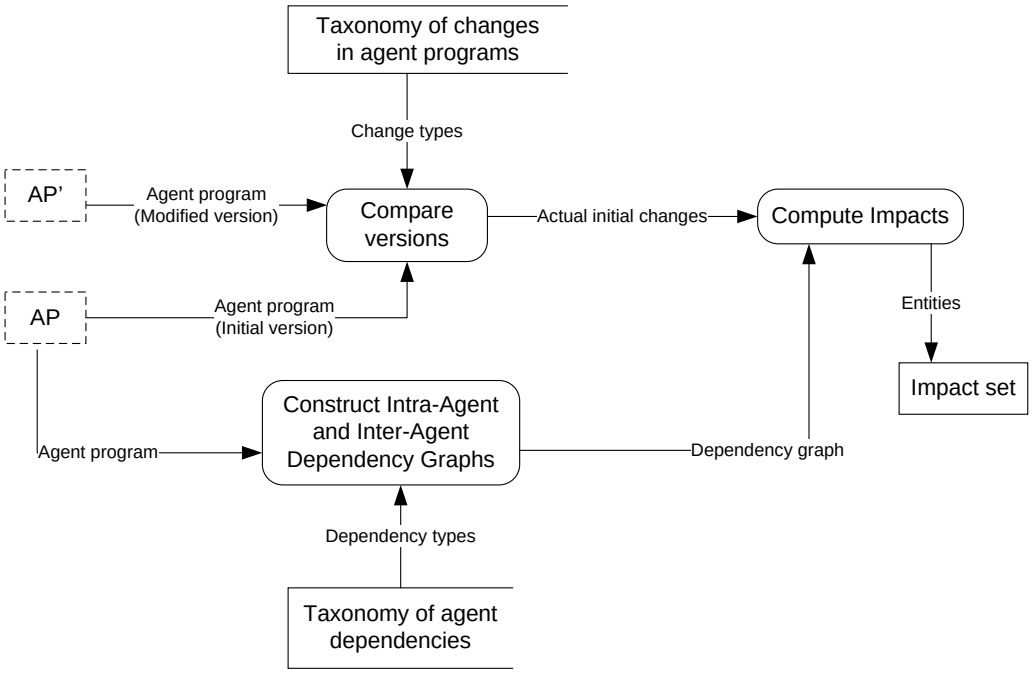

Figure 3: A static change impact analysis framework for agent systems

We then calculate the impact of the primary changes that have initially been made to the agent system. To do so, we first construct graphs (i.e.

Intra-Agent and Inter-Agent Dependency Graphs) which capture dependencies between different entities in the original version of the agent program (i.e. AP) using a pre-defined taxonomy of dependencies. The classification captures various types of dependencies including intra-agent dependencies (e.g. between entities in a plan, between plans and beliefs, and between plans in an agent) and inter-agent dependencies (in the form of inter-agent communication). We then use the dependency graphs to calculates the set of impacted entities. We consider an impacted entity as being the one may require modification due to the change of another entity. Therefore, we traverse the dependency graphs to identify other entities that have potential dependency relationships with the initial ones, and form a set of impacts. Those impacted entities also relate to other entities and thus the impact analysis continues this process until a complete transitive closure graph is obtained. We now describe the two key data sources (i.e. a change taxonomy and a classification of dependencies of agent systems) in our impact analysis technique and how our technique calculates the impact set in details.

3.1. Change taxonomy of agent systems

We now describe different types of changes in an AgentSpeak program and their relationships (refer to figure 4). The change taxonomy was developed by examining all entities comprising an AgentSpeak program. At the system level, an agent program has a number of agents, each of which has a set of plans and a set of beliefs. Therefore, changes made to an AgentSpeak program include changing an existing agent, adding a new agent or deleting an existing agent. Changes to an agent involve adding a new plan (to the plan library), deleting or changing an existing plan, adding a new belief (to the belief set), and removing or changing an existing belief. Note that since a belief is a literal (e.g. the belief literal $pos(r2, 2, 2)$ of agent $r1$ in our example in section 2.2), changing a belief is considered as changing the literal.

A plan consists of a triggering event, a context condition (which is a conjunction of literals) and a plan body. Therefore, changes made to a plan include changing the triggering event, changing the context condition and changing the plan body. A triggering event consists of a literal, e.g. the triggering event $+garbage(r1)$ has the literal $garbage(r1)$ whereas $+!go(R)$ has the literal $go(R)$. Therefore, changing the triggering event is actually making a change to the corresponding literal. Note that the concept of changing a literal is similar for both context conditions and triggering events since they both consist of literals. Changing a literal is in turn classified

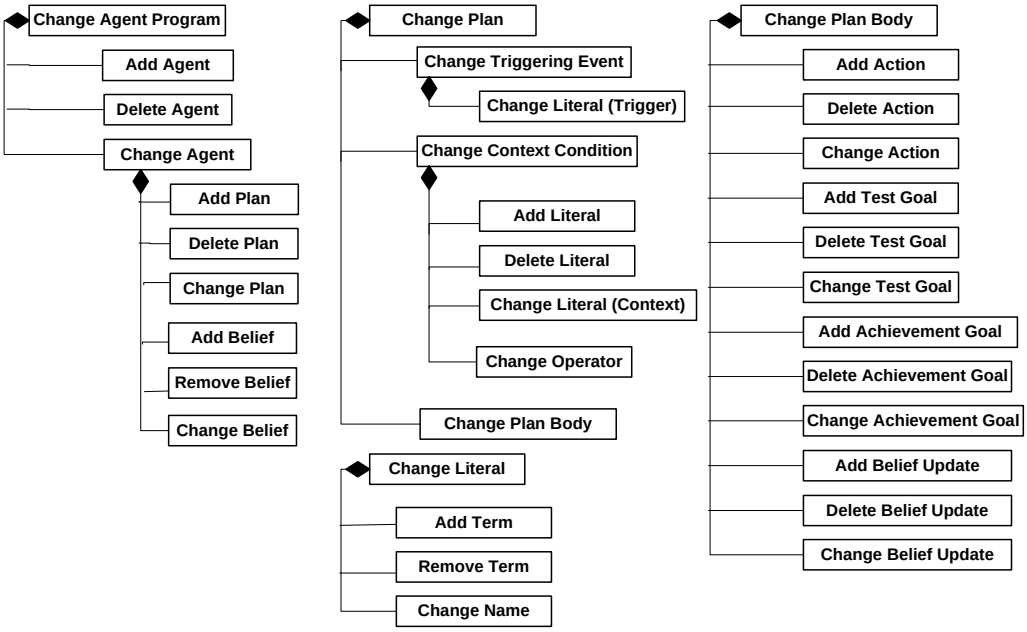

Figure 4: A taxonomy of changes for AgentSpeak programs

into three types: changing literal name, adding a new term, and removing an existing term.

Furthermore, since a context condition is a conjunction of literals (e.g. $checking(slots) \wedge \neg garbage(r1)$), we consider changing a context condition as involving either adding a literal, deleting a literal or changing an existing literal. Although a context condition in AgentSpeak contains only a conjunction of literals, Jason (as an extension of AgentSpeak) does allow disjunctions in a context condition. Therefore, we also consider changing operator (e.g. to a disjunction) as a change type. Finally, a plan body may consist of a number of entities including actions, test goals, achievement goals and belief updates. Therefore, changing a plan body involves either adding, deleting or changing such entities. For instance, one can change plan $P2$ of agent $r1$ by deleting the achievement goal $!stop(check)$ from its body or adding a new action into its body. In addition, similarly to changing the name of a triggering event's literal, changing the name of the literal associated with an action, a test goal, an achievement goal or a belief update is considered as a deletion (e.g. of an existing action) and addition (e.g. of a new action).

Figure 4 describes a taxonomy of changes in AgentSpeak programs. It

also shows dependencies between different change types: the non-leaf nodes representing changes that take place only when the leaf node change occurs. For instance, a change in an agent program could be induced by changing an existing agent in the program, which in turn could be induced by adding a new plan to that agent's plan library. Such dependencies allow us to consider the impact of a change at multiple levels of granularity.

3.2. Classification of dependencies in agent systems

We present here a taxonomy of dependencies that exist in an agent system. These include intra-agent dependencies: between plans and beliefs, between plans, and between entities (e.g. triggering event, context condition and plan body) within a plan; and inter-agent dependencies (via inter-agent messages). We use the example in section 2.2 to illustrate different types of dependencies.

3.2.1. Dependencies between plans within an agent

A plan body may consist of subgoals (which can be either test goals or achievement goals) or belief updates. There may be other plans (in the agent's plan library) which achieve those subgoals or handle the belief update events. Therefore a subgoal or a belief update in a plan depends on the triggering event of another plan if the two associated literals are unifiable[6]. For instance, subgoal $!go(r2)$ in the body of plan $P2$ depends on the triggering event $+!go(R)$ of plans $P5$ and $P6$ since $go(r2)$ is unifiable with $go(R)$. On the other hand, the belief update $+pos(back, X1, Y1)$ of plan $P3$ does not depend on the triggering event $+pos(r1, X1, Y1)$ of plan $P1$ since $pos(back, X1, Y1)$ is not unifiable with $pos(r1, X1, Y1)$ ($r1$ is not $back$).

3.2.2. Dependencies within a plan

Dependencies within a plan revolve around the triggering event, context condition and entities (i.e. actions, belief updates, test goals and achievement goals) in the plan body and are mediated by shared variables. It is noted that since plans resemble logic programming clauses, the scope of a variable is limited to a plan.

[6]Two literals α and β are unifiable iff there are substitition δ_A and δ_B such that $\alpha\delta_A = \beta\delta_B$ where (e.g.) $\alpha\delta_A$ is the application of the substitution δ_A to α, i.e. the literal that is obtained when each occurrence in α of a variable in δ_A is replaced by the associated term.

- A predicate in the context condition depends on the triggering event if their associated literals shares a common term. For instance, the predicate $garbage(r1)$ in the context condition of plan $P1$ depends on its triggering event $+pos(r1, X1, Y1)$.

- An entity (i.e. action, test goal, achievement goal or a belief update) in the plan body depends on a triggering event or a predicate in the context condition if their associated literals share a common term. For instance, the action $next(X1, Y1)$ in plan $P1$ depends on the plan's triggering event $+pos(r1, X1, Y1)$ since they share variables $X1$ and $Y1$. Similarly, subgoal $!go(R)$ in plan $P6$ depends on the plan's triggering event $+!go(R)$.

- An entity (i.e. action, test goal, achievement goal or a belief update) in the plan body depends on another entity appearing earlier in the same plan body if their associated literals share a common term. For instance, subgoal $!go(r2)$ of plan $P2$ depends on action $.send(r2, askOne, pos(r2, X2, Y2), Reply)$ since they share $r2$. Similarly, the belief update $-pos(back, X1, Y1)$ depends on the test goal $?pos(back, X1, Y1)$ in plan $P4$, and subgoal $!go(R)$ depends on test goal $?pos(R, X1, Y1)$ in plan $P6$.

3.2.3. Dependencies between a plan and a belief

There are a number of dependencies between a plan and a belief:

- A predicate in the plan's context condition depends on a belief if their associated literals are unifiable. For instance, the predicate $pos(R, X1, Y1)$ in the context condition of plan $P5$ depends on the belief $pos(r2, 2, 2)$ in the initial belief base of agent $r1$.

- A test goal in the plan body depends on a belief if their associated literals are unifiable.

- A triggering event in the plan depends on a belief if their associated literals are unifiable.

3.2.4. Dependencies between two agents

Agents (in Jason) communicate with each other in terms of exchanging messages: agent s sends a message to agent r by executing $.send(r, ilf, msg)$,

where ilf is illocutionary forces including: information exchange: *tell* and *untell*; goal delegation: *achieve* and *unachieve*; know-how related: $tellHow$, $untellHow$, and $askHow$; information seeking: $askOne$, and $askAll$. For instance, plan $P2$ of agent $r1$ (refer to the example in section 2.2) has an action of sending a message to $r2$ and asking for its location (i.e. $.send(r2, askOne, pos(r2, X2, Y2), Reply)$). For more details regarding the inter-agent communication messages supported in Jason, we refer the readers to [26].

Assume that $.send(r, ilf, msg)$ is an action in the plan P of agent s, which sends msg to agent r. Dependencies between agents are reduced to dependencies between the msg (e.g. plans, events, and beliefs) with entities in agent r. Therefore, we can apply the above classification of dependencies between entities within an agent to dependencies between entities across different agents.

- tell/untell: msg is a belief which is added to agent r when it receives the message. Therefore, the dependencies between plans in r and this belief can be applied here. For example, assuming agent $r2$ sends a message $.send(r1, tell, pos(r2, 4, 3))$ to agent $r1$, then we can establish a dependency between the message $pos(r2, 4, 3)$ and plan $P6$ (of agent $r1$) through the plan's context condition $pos(R, X1, Y1)$.

- achieve/unachieve: msg is an event/goal. There are dependencies between the triggering event of plans in r and msg if their associated literals are unifiable. For example, assuming agent $r2$ sends a message $.send(r1, achieve, go(r2))$ to agent $r1$, then we can establish a dependency between the message $go(r2)$ with plans P5 and P6 (of agent $r1$) through their triggering event $+!go(R)$.

- tellHow: msg is a plan and thus dependencies between this plan and other plans, and this plan and beliefs in the receiving agent are established. For example, assuming that agent $r2$ sends a message $.send(r1, tellHow, +!go(R) : checking(X) \leftarrow moveForwards(2, 2))$, then we can establish, for example, a dependency between the triggering event $+!go(R)$ with the subgoal $!go(r2)$ in plan P2, or a dependency between context condition $checking(X)$ with belief $checking(slots)$.

- askHow: msg is a triggering event. There are dependencies between the triggering event of plan in r with this msg if their associated literals are unifiable. For example, assuming agent $r2$ sends a message

$.send(r1, achieve, "+!go(r2))$ to agent $r1$, then we can establish a dependency between the message $+!go(r2)$ with plans P5 and P6 (of agent $r1$) through their triggering event $+!go(R)$.

- askOne[7]: msg is a test goal. Therefore, there are dependencies between triggering event (of a test goal) of plans in r and this msg, and also dependencies between other beliefs in r and msg. For instance, the message $pos(r2, X2, Y2)$ in the action $.send(r2, askOne, pos(r2, X2, Y2), Reply)$ of plan $P2$ in agent $r1$ depends on the triggering event $+?pos(r2, X2, Y2)$ of plan $P8$ in agent $r2$.

3.3. Calculate impact set

In order to calculate the impacts, we construct two graphs that represent various dependencies in an agent system that have been discussed in the previous section. Firstly, an Intra-Agent Dependency Graph is used to describe the dependency among entities within an agent. Note that edges in this graph are directional to reflect the dependency relationship. This graph can be used to calculate the impacted entities inside an agent when certain entities in the agent are changed.

Definition 1. *The Intra-Agent Dependency Graph (Intra-Agent DG) of an agent is a directed graph G = (N, E). N is the set of nodes in which each entity (including a belief in the agent's initial belief base, or a triggering event, a predicate in the context condition, an action, a belief update, or a subgoal in a plan) maps to a node.* $E \subseteq (N \times N)$ *is the set of edges in which each dependency between two entities in the agent maps to an edge, and the target node of the edge depends on the source node.*

Secondly, an Inter-Agent Dependency Graph is used to describe dependency relationships among different agents.

Definition 2. *An Inter-Agent Dependency Graph (Inter-Agent DG) is a set of tuples* $\psi = (G_i, E_i)$, $i = 1 \ldots n$, *where n is the number of intra-agent DGs (i.e. the sub-graphs) in* ψ. G_i *is an intra-agent DG and* N_i *is the set of nodes in* G_i. *U represents all the nodes in* ψ *and* $U = \bigcup_{i=1}^{n} N_i$. *Relation* $E_i \subseteq N_i \times (U - N_i)$ *represents a set of edges among the sub-graphs that maps a node in* N_i *to another node in U (but not in* N_i*).*

[7] *askAll* can be treated similarly.

The input to our framework is a set of atomic changes, each of which is in the form of a tuple $< E_{id}, CT >$ where E_{id} is the ID of an entity being changed and CT is the type of change. We generate an internal unique identifier for each entity (e.g. test goals, plans, etc.) in an agent program. For example, $< g1, ChangeTestGoal >$ indicates changing test goal $g1$. Note that since $g1$ is the ID of the test goal, we are able to retrieve the plan it belongs to, and the agent the plan belongs to.

Algorithm 1: ComputeTotalImpacts(): Compute total impacts of a change

Input:
CS, the set of atomic changes initially made
G, the Inter-Agent DG of the original agent system
Output: IES, the set of impacted entities in the agent system

```
1  begin
2      Unmark all nodes in G
3      foreach change C in CS do
4          Let E be the entity changed by C
5          Let CT be the change type specified in C
6          IES ≝ IES ∪ {E}
7          if CT is not a type of Addition then
8              Let N be the node of the graph that represents E
9              IES ≝ IES ∪ ComputeImpact(N, G)
10         end
11     end
12 end
```

We compute impacts by simply traversing the inter-agent dependency graph using a depth first search as presented in algorithms 1 and 2. Algorithm 1 describes the function $ComputeTotalImpacts()$ that takes a set of atomic changes initially made to the original agent system and the inter-agent DG of the original agent system as input, and returns a set of entities impacted by the changes. For each change, we identify the entity being changed and add it to the impact set. We then obtain the change type as specified in this change (e.g. Add Test Goal or Delete Test Goal) and check whether it is an *addition* change type. If so, we move to the next change since the addition of

a new entity would not affect any other entities in the system (because they do not use the new entity yet). Otherwise, we get the node representing the entity being changed in the the inter-agent DG, and perform a depth first search. Algorithm 2 traverse the inter-agent DG starting from a given node to collect nodes that are reachable from that node and add the entities they represent to the impact set.

Algorithm 2: ComputeImpact(): Compute impacts of a change using depth first search

Input:
G, the Inter-Agent Dependency Graph
N, a node in this graph
Output: IES, the set of impacted entities in the agent system

```
1  begin
2      Mark N
3      foreach Node M in Target(N) do
4          if M is not marked yet then
5              Let E_M be the entity that M represents
6              IES ≝ IES ∪ {E_M} ∪ ComputeImpact(M, G)
7              Mark node M
8          end
9      end
10 end
```

Our algorithm for computing the impact set is based on calculating transitive closure which has been widely used in change impact analysis for traditional software. This offers a conservative approach to estimate the system-wide impacts of proposed changes. We can improve it slightly by having a depth, i.e. impact distance, and stop searching if it reaches a certain depth. This technique used in [30] based on a (weak) assumption that if direct impacts have high potential for being true, then those further away will be less likely. Another different approach is establishing some barriers to the transitive propagation by assessing (using some heuristics) whether an entity is likely to be changed and consequently deciding whether a change should be propagated from this entity to other dependent entities. We would also need to investigate whether there are propagating entities in agent systems – those

entities do not change but they propagate changes to their neighbours. Future work would involve investigating other techniques that can be employed to improve our current algorithms.

Let us briefly illustrate how this method works using the running example Mars robots in section 2.2. Assume that there is a new requirement that the robots are required to explore what lies underneath Mars' surface. As a result, the robots are now able to perceive the depth where they are located. Assume that the software engineer firstly modifies agent $r2$ to meet this new requirement by adding another argument which represents the depth dimension (i.e. $Z2$) to triggering event $+?pos(r2, X2, Y2)$ of plan $P8$, and then uses our framework to estimate the impact of this change.

Within the agent $r2$, since test goal $?pos(r2, X2, Y2)$ in plan $P9$ depends on the modified triggering event of $P8$ (refer to section 3.2 for the taxonomy of dependencies), this goal is included in the impact set by our framework. In addition, action $next(X2, Y2)$ in $P9$ depends on the test goal $?pos(r2, X2, Y2)$ and thus the action is also added to the impact set. In terms of inter-agent dependencies, action $.send(r2, askOne, pos(r2, X2, Y2), Reply)$ in $P2$ of agent $r1$ depends on triggering event $?pos(r2, X2, Y2)$ in agent $r2$, therefore this action is also included in the impact set. Furthermore, subgoal $!go(r2)$ in the same plan depends on this action and consequently is added to the impact set. This process continues until we collect all the entities forming the complete impact set of the initial change.

4. Dynamic impact analysis

The above static impact analysis technique examines an agent's source code and determines the impact of a change. In this section, we describe a dynamic approach to impact analysis of agent systems which relies on its execution log (rather than its source code). We first describe a typical execution trace of an agent program and then show how an impact set can be computed from execution traces generated from different behaviours of an agent system.

4.1. Execution traces

The hierarchical structure of BDI plans which determine the run-time behaviour of a BDI agent can be viewed as a goal-plan tree where each goal has children representing the relevant plans for achieving it, and each plan has children representing the subgoals (including primitive actions) that it

has. This goal-plan tree can be seen as an "and/or" tree: each goal is achieved by a successful execution of one of its plan ("or"), and the success of each plan relies on all of its sub-goals being resolved ("and"). Figure 5 shows an example of such a goal-plan tree for landing an unmanned aircraft (goal G). This can be achieved by either plan P1 (normal landing plan) or P2 (emergency landing plan), depending on the current condition. The normal landing plan has two subgoals: obtaining landing permission (G1) and flying at lower speeds (G2). Obtaining landing permission from the air traffic controller can be achieved using either traditional analogue voice radio (P3), or using digital Voice-over-IP system (P4) or using data link communication (P5). Flying at lower speeds can be achieved by lifting the aircraft's flaps (P6). The emerging landing plan (P6) involves landing on a suitable site (subgoal G3), which can be achieved by either landing on a ground (P7) or landing on water (P8).

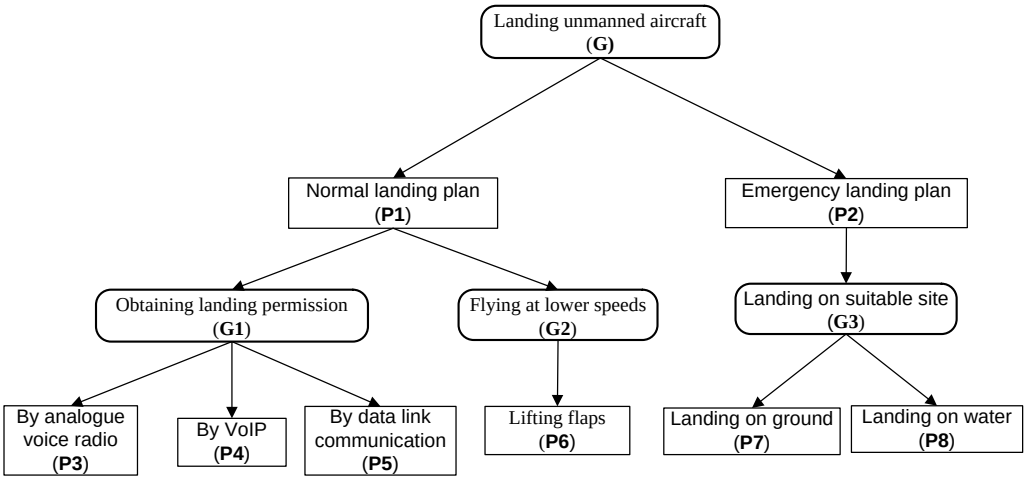

Figure 5: A goal-plan tree for agent A

As an example, suppose we have a single execution trace t, shown by a string of letters in figure 6, for an agent A whose a goal-plan tree appears in figure 5. Note that G_p denotes goal G being posted, whereas G_s indicates goal G being successfully achieved. Similarly, P_e denotes plan P beginning execution and P_s indicates a successful completion of plan P. As can be seen, the execution trace in figure 6 demonstrates that landing the aircraft (goal G is posted) in a normal situation leads to the execution of the normal landing plan (P1 executes), which involves obtaining landing permission (goal

G1 is posted) using analogue voice radio (plan P3 executes and successfully completes, and consequently goal G1 is successfully resolved). The execution of the normal landing plan then involves flying the aircraft at lower speeds (goal G3 is posted) by lifting the flaps (plan P6 executes and successfully completes, and consequently goal G2 is successfully resolved). The normal landing plan P1 therefore successfully completes, and thus the landing aircraft goal G is successfully achieved.

$$\boxed{\mathrm{G}_p\ \mathrm{P1}_e\ \mathrm{G1}_p\ \mathrm{P3}_e\ \mathrm{P3}_s\ \mathrm{G1}_s\ \mathrm{G2}_p\ \mathrm{P6}_e\ \mathrm{P6}_s\ \mathrm{G2}_s\ \mathrm{P1}_s\ \mathrm{G}_s}$$

Figure 6: A typical execution trace t for an agent A

4.2. Calculate the impact set

Assume that we propose to change plan $P6$ (lifting the flaps) in the above example, an impact analysis technique needs to determine the other plans and/or goals that are potentially affected by the change (i.e. the impact set). The static analysis technique proposed in [24] computes the impact set by considering static (direct and indirect) dependencies between $P6$ and other goals or plans in the agent system. It works under the assumption that a change in $P6$ has potential impact on any nodes that are reachable from $P6$ or can reach $P6$ in the goal-plan tree for agent A. Therefore, an impact set of plan $P6$ returned by the static technique in [24] contains all entities in the goal-plan tree in figure 5. This would result in highly inaccurate impact set, as evidenced by the experimental result (i.e. precision of approximately 0.3–0.4). We will now show that our dynamic analysis technique which relies on information from execution traces can predict impact sets that are more accurate than those computed by static analysis.

The intuitive reason for our dynamic analysis technique can be summarized as follows: a change made to plan or goal E would only propagate down any (and only) dynamic paths that have been observed to pass through E. As a result, any plan or goal that is executed after E, and any goal or plan which is on the execution stack after E finishes its execution, is included in the set of potentially impacted goals or plans. Thus, calculating the impact set for plan or goal E involves searching forward in the execution trace to find plans/goals that are called directly or indirectly by E and goals/plans that are executed after E finishes, and searching backward to discover the goals/plans which E returns into.

Our dynamic analysis technique relies on execution traces such as the one in figure 6 rather static goal-plan trees. Given a set of changes, we adapt the PathImpact technique [20] to perform forward and backward walks of a trace to identify the impact set of the changes. The forward walk determines all plans executed and all goals posted after the changed goal/plan, whereas the backward walk identifies plans/goals into which the execution can return. More specifically, for each changed entity E (which can be either a plan or goal) and each occurrence of E_e (if E is a plan) or E_p (if E is a goal), we will do the following. Note that we will illustrate our technique using an example of trace t in figure 6 and a change set {P6} (i.e. only plan P6 is modified).

- In the forward walk, we start from the entity immediately following E_e (if E is a plan) or E_p (if E is a goal), add every plan executed or goal posted into the impact set (i.e. every entity F such that the trace contains an entry F_e or F_p after the occurrence of E_e or E_p), and count the number of unmatched successes. "Unmatched" (successful) goals/plans (G_s or P_s) are those that we do not encounter their execution (G_p or P_e) in our forward walk. For example, if our forward walk encounters both $G1_p$ and $G1_s$, then $G1_s$ is considered as "matched" success. Otherwise (i.e. $G1_p$ is not encountered in this specific walk), $G1_s$ is a "unmatched" success. In our example, in the forward walk we start at $P6_s$ and add nothing to the impact set since there is no plan executed or goal posted after P6. We however count 3 unmatched successes (i.e. $G2_s$, $P1_s$, and G_s)

- In the backward walk, we begin from the entity immediately preceding E_e (if E is a plan) or E_p (if E is a goal), and add into the impact set as many unmatched plans or goals as the number of unmatched successes counted in the forward walk. In our example, we add G2, P1, and G to the impact set.

- Add E to the impact set if it is not already there. Therefore, the impact set in our example would be {P6, G2, P1, G}. It means that changing the lifting flaps plan might impact the goal of flying at lower speeds, and consequently the normal landing plan and the goal of landing the aircraft.

The above trace is an example of a typical, successful execution. An agent however may often exhibit some distinct behaviours including concurrency,

failures, and interruption. We now explain how such behaviours can be observed from analysing execution traces and how our technique deals with them.

Concurrency

An agent can interleave multiple activities concurrently, each of which attempts to achieve one of the agent's goals. Such goals are active simultaneously and might interact both negatively and positively [31]. Negative interactions may involve such things as competition for resources, i.e. an aircraft having 10 units of fuel but pursuing two goals, each of which requires 6 units. Positive interactions may include situations where the two goals potentially share a common subgoal. For instance, an autonomous aircraft, which has two (sub)goals of obtaining landing permission (G1) and flying at lower speeds (G2), could achieve the goals sequentially by contacting the air traffic controller using (e.g.) analogue voice radio (i.e. plan P3) to obtain landing permission, and then if approval, lifting the flaps (plan P6) to lowering the aircraft's speed. Alternatively, it could pursue those two goals in parallel, i.e. getting the landing permission and lowering the speed at the same time.

$$\boxed{\mathrm{G}_p\ \mathrm{P1}_e\ \mathrm{G1}_p\ \mathrm{G2}_p\ \mathrm{P3}_e\ \mathrm{P6}_e\ \mathrm{P3}_s\ \mathrm{G1}_s\ \mathrm{P6}_s\ \mathrm{G2}_s\ \mathrm{P1}_s\ \mathrm{G}_s}$$

Figure 7: Trace t_1 for agent A (concurrency)

The above example indicate that a goal that is pursued concurrently with a changed goal is also potentially affected by the change because of possible interactions. By analysing an execution trace, we are able to detect goals that are pursued concurrently. For instance, in our goal-plan tree example in figure 5, the agent may choose to achieve goals G1 and G2 concurrently, which is demonstrated as trace t_1 in figure 7. As can be seen, the agent does not wait for goal G1 to be resolved before posting goal G2 and the execution of plans P3 and P6 interleaves with one another. We can apply the same technique described earlier to determine the impact set of the changed plan P6. In the forward walk starting from $\mathrm{P3}_s$ (immediately followed $\mathrm{P6}_e$), we count 5 unmatched successes. As a result, in the backward walk we add P3, G2, G1, P1 and G into the impact set. It means that changing the lifting flaps plan (P6) might also impact the obtaining landing permission goal (G1) using analogue voice radio (plan P3) as they are pursued and executed concurrently with the plan P6.

Failures

Execution of a plan, however, can fail in some situations, e.g. a subgoal may have no applicable plans or an action can fail. For example, assume that due to a technical problem one of the flaps cannot be lifted. If the agent is pursuing to achieve a goal (e.g. landing the aircraft), a mechanism that handles failure is used. Typically, the agent tries an alternative applicable plan to achieve the goal. Figure 8 shows an execution trace t_2 illustrating an example in which plan P3 (obtaining landing permission using analogue voice radio) is tried and fails (denoting as $P3_f$), and P4 (using VoIP instead) is tried and succeeds. For a given changed goal (e.g. goal G1, obtaining landing permission), static analysis (e.g. [24]) would determine all plans that are relevant to the goal (e.g. plans P3, P4 and P5 for goal G1) as being potentially impacted by the change. Details from the execution traces that has plan failures may however reveal which of those plans are actually executed and which are not. For instance, trace t_2 indicates that only plans P3 and P4 are executed and consequently implies that only these two plans are potentially affected by the change in goal G1 (obtaining landing permission). Therefore, plan failures further indicate that dynamic impact analysis tends to give a more accurate impact set than static analysis.

$$\boxed{G_p\ P1_e\ G1_p\ P3_e\ P3_f\ P4_e\ P4_s\ G1_s\ G2_p\ P6_e\ P6_s\ G2_s\ P1_s\ G_s}$$

Figure 8: Trace t_2 for agent A (plan failure)

In the case when all applicable plans are tried and failed, a goal is considered to have failed. Note that failures propagate upwards through the goal-plan tree: if a plan fails its parent goal is re-posted; if this fails then the parent of the goal (i.e. a plan) fails and so on. Note that in case of plan failure, reposting the goal and/or trying alternative plans is not the default behaviour in Jason. A rather simple plan failure mechanism needs to be implemented to achieve this behaviour (e.g. defining plans to handle the goal deletion event generated by Jason interpreter). Execution trace t_3 (figure 9 shows an example of such a failure propagation in which all plans P3, P4 and P5 are tried and fail, resulting in goal G1 having failed, and consequently plan P1 having failed. Therefore, plan P2 is executed to resolve goal G. In contrast to trace t_2, trace t_3 reveals that all the three plans P3, P4, and P5 are executed. In order to accommodate the failure cases, our technique needs a slight change: in the forward walk we also collect unmatched failures

(i.e. E_f), and in the backward walk we include as many unmatched plans or goals as the total number of unmatched successes and failures counted in the forward walk.

$$G_p\ P1_e\ G1_p\ P3_e\ P3_f\ P4_e\ P4_f\ P5_e\ P5_f\ G1_f\ P1_f\ P2_e\ G3_p\ P7_e\ P7_s\ G3_s\ G_s$$

Figure 9: Trace t_3 for agent A (goal failure)

Interruption

An executing plan might be suspended (and put into the set of suspended plans) due to either waiting for feedback on action execution (e.g. confirmation of the flaps retracted) or waiting for message replies from other agents (e.g. instructions from the air control tower). Before another execution cycle begins, the agent checks whether any such feedback are now available, and if so the relevant plans are updated and pushed back in the set of current plans so that their execution can be resumed in the next execution cycle. In some other cases, an executing plan may be suspended since a higher-priority event has just occurred and the agent needs to deal with it urgently. For instance, an unmanned aircraft is unloading goods at location X (as part of executing a plan to transport goods to X) and unfortunately an earthquake occurs at X. The agent needs to suspend the current plan (transporting goods) and executes another plan (e.g. evacuation plan) to deal with this (assumed) higher priority event/goal. This behaviour is relatively common in agent systems since one of the key agent's properties is the ability to respond quickly to changes in the environment.

$$G_p\ P1_e\ G1_p\ P3_e\ G2_p\ P6_e\ P6_s\ P3_s\ G1_s\ G2_s\ P1_s\ G_s$$

Figure 10: Trace t_4 for agent A

In some situations, there might be some relationships between the suspended plan and the newly emerging goal. For instance, the plan to transport goods to location X is suspended since the earthquake and the evacuation plan is also at location X. Therefore, a change to one of them might have an impact on the other. Plan suspension can be observed from an execution trace of an agent. For instance, execution trace t_4 (figure 10) indicates that P3 (obtaining landing permission using analogue voice radio) is executed to

achieve goal G1 but is then suspended since the agent needs to resolve goal G2, i.e. flying at lower speeds (due to for example safety reasons) by executing plan P6 (lifting the flaps). After plan P6 completes, the agent continues completing the execution of plan P3. Similarly to traces produced by concurrency, we can apply the same technique described earlier to determine impact sets from traces derived from plan suspension.

In practice, there are usually multiple execution traces of an agent system. Each trace corresponds to an execution of the program, i.e. if the program stops and starts again, it gives another execution trace. Multiple execution traces would potentially represent different behaviour of the program. In this case, we process each single trace and compute the union of the impact sets returned by each execution traces. For example, assume that execution traces t (figure 6) and t_2 (figure 8) are collected from an operation profile of the agent in the above example. Given the changed goal $G1$, processing trace t gives us an impact set of {G1, P3, G2, P6, P1, G} whereas trace t_2 gives us {G1, P3, P4, G2, P6, P1, G}. Therefore, considering both execution traces would return an impact set of {G1, P3, P4, G2, P6, P1, G} in this example.

Our impact analysis approach can also cover inter-communication between agents by observing execution traces of the whole multi-agent system (rather than each individual agent in the system). For example, when agent $r2$ (in our example) receives the askOne message from agent $r1$ asking for its current location, an test goal (i.e. $+?pos(r2, X2, Y2)$ is generated/posted within agent $r2$. By observing the execution trace of the whole system, we would observe that the execution of plan $P2$ in agent $r1$ (which has the askOne message) is followed by the test goal in agent $r2$, and possibly the plan (in $r2$) handling this goal and so on. The same impact analysis technique (described earlier) can be applied in this case.

4.3. Algorithm

Our dynamic impact analysis technique is formalized in terms of two algorithms. Algorithm 3 describes the function $ComputeTotalImpacts()$ that takes a set of initially changed plans or goals in an agent system (i.e. the primary changes) and a set of execution traces as input, and returns a set of entities (either goals/plans) potentially impacted by the changes. It simply processes each change against each execution trace and union the resulted impact sets.

The main processing is in algorithm 4 which describes the function *ComputeImpact()*. This function takes as input a changed plan or goal (i.e. E)

Algorithm 3: ComputeTotalImpacts(): Compute total impacts of a set of changes given a set of execution traces

Input:
ES, the set of initially changed plans/goals
EXS, the set of execution traces
Output: IES, the set of impacted goals/plans in an agent system, initially empty

```
1 begin
2   foreach changed E in ES do
3     foreach execution trace EX in EXS do
4       IES =def IES ∪ ComputeImpact(E, EX)
5     end
6   end
7 end
```

and an execution trace (i.e. EX), and returns an impact set (i.e. IES) by following the technique we presented in the previous section. Firstly, it looks for the occurrence of E_p (in case E is a goal) or E_e (E is a plan) in the execution trace (line 3), and if found, it starts the forward and backward walks (lines 8 – 26) on the execution trace. The forward walk (lines 10 – 18) traverses from the entity immediately following E to the end of the trace. It adds every plan executed or goal posted into the impact set (lines 11 – 13), and counts the number of unmatched successes or failures (lines 14 – 16). The backward walk (lines 20 – 26) traverses from the entity immediately preceding E . It adds into the impact set as many unmatched plans or goals as the number of unmatched successes or failures counted in the forward walk (lines 21 – 24).

Our impact analysis algorithm requires a time that depends on the size of the execution trace analysed. In terms of space, the algorithm's space cost is proportional to the size of the traces (which can be very large). Therefore, our future work would involve exploring how to compress the execution traces to reduce the overhead in both time and space.

Algorithm 4: ComputeImpact(): Compute impact of a change given an execution trace

Input:
E, the changed plan/goal
EX, an execution trace (an array)
Output: IES, the set of impacted goals/plans in an agent system, initially empty

```
1  begin
2      length ≝ length of EX
3      i ≝ position of E_p (goal) or E_e (plan) in EX
4      if i == –1 then
5          return IES ;
6      end
7      IES ≝ IES ∪ {E}
       // Go forward
8      k ≝ i+1
9      unmatched ≝ 0
10     while k < length do
11         if type(EX[k]) == e or p then
12             IES ≝ IES ∪ {EX[k]}
13         end
14         else if type(EX[k]) == s or f and no execution or posting of
           EX[k] from EX[i] to EX[length–1] then
15             unmatched++
16         end
17         k++
18     end
       // Go backward
19     j ≝ i–1
20     while j > –1 do
21         if type(EX[j]) == e or p and no failure or success of EX[j]
           from EX[0] to EX[i] and unmatched > 0 then
22             IES ≝ IES ∪ {EX[k]}
23             unmatched– –
24         end
25         j– –
26     end
27     return IES
28 end
```

5. Empirical Validation

Both impact analysis techniques have been implemented. A prototype of our static impact analysis has been implemented in AgentCIA[8], a change impact analysis plugin for the Jason IDE[9]. The plugin is a full implementation of our static impact analysis technique including: a *parser* for extracting the AgentSpeak code, building an Inter-Agent DG, and comparing a given AgentSpeak program and its modified version to detect the changes and classifying them into atomic change elements; an analyzer is mainly responsible for calculating the change impact; and a view to display the impact results. We have also implemented our dynamic impact analysis algorithms which uses information from execution logs of an agent program.

We have performed an experiment to investigate whether our impact analysis technique computes an appropriate impact set relative to a set of dynamic execution traces, and how that impact set compares to those calculated by the static approach. We now discuss how we have designed such an experiment, the measures that we used, the outcomes and some major threats to the validity of our experiment.

5.1. Design and measures

The key question that we would like to address is: how well our static impact analysis technique works in practice compared with our dynamic impact analysis technique. In addition, we would like to compare the effectiveness of both techniques with a random impact analysis involving a set of entities being randomly selected as an impact set. Given a set of primary changes PC, the effectiveness measurement we used involves two sets: the set of *potentially* impacted entities (e.g. plans/goals) predicted by an impact analysis technique (i.e. the estimated set E) and the set of entities *actually* affected (i.e. the actual set A), for a given set of primary changes PC.

Given a set of primary changes PC, the aim of an impact analysis technique is determining the portion of the software truly affected by the change (set A). However, an impact analysis technique may not be fully accurate when identifying the impact set. Figure 11 illustrates this issue. Overestimating impact generates *false-positives* (i.e. entities that are in E but not

[8]AgentCIA is available at http://agentspeakcia.sourceforge.net

[9]The Jason IDE on the Eclipse platform provides an environment for developing AgentSpeak agent systems, and also supports plugin development

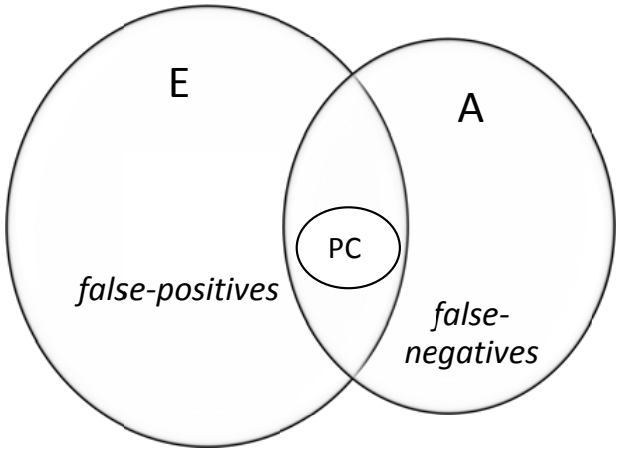

Figure 11: The impact of a change PC (E is the estimated set and A is the actual set)

in A), which forces the software maintainers to spend additional, unneeded time investigating the impact set that contains unnecessary information. On the other hand, underestimating impact produces *false-negatives* (i.e. entities that are in A but not in E), which leads the software maintainer to omit important impacts of a change. If such impacts continue being omitted when implementing the change, they may cause inconsistencies in the software which result in bugs.

It is important to note that our approach to static impact analysis is effectively an abstraction of what could be a far more detailed and complex approach that might compute the precise impact of a given change (we avoid this approach given our emphasis on supporting agent developer decisions in near-real time). Generally, analysis at a finer level of granularity would give a more accurate result but attract more computational overhead. For example, analysis done at the level of variables (instead of literals as in our approach) would be more precise in terms of telling exactly what other variables or terms are affected by a change to a given variable. Such an analysis would however be very computationally expensive. Our experimental evaluation of this approach thus takes a similar form to the evaluation of our dynamic impact analysis approach, i.e., we compute the effectiveness of the retrieval of impacted artifacts as if were an information retrieval problem, even though the exact set of impacted artifacts could in principle be computed (but with a likely unacceptable time complexity).

We use two relative measures: *precision* and *recall*, which are associated

with false-positives and false negatives. Precision and recall are the most widely-used metrics in the information retrieval literature and have been recently adapted to the impact analysis setting (e.g. [30]). Precision is defined as the ratio between the correctly predicted (i.e. the intersection of E and A) and the total of predicted entities (i.e. E): $Precision = \frac{|E \cap A|}{|E|}$. Recall is the ratio between correctly predicted entities (i.e. the intersection of E and A) and the total of actually affected entities (i.e. A): $Recall = \frac{|E \cap A|}{|A|}$. On the one hand, a perfect precision score of 1.0 means that every entity predicted as being impacted by an impact analysis technique was actually changed. On the other hand, a perfect recall score of 1.0 means that all changed entities were predicted as being impacted by the technique.

Execution traces (used by our dynamic impact analysis) may be collected from the execution of a subset of the code (other parts may never be executed). We therefore define a metric, namely T_{rep}, to measure the extent of an agent system that is represented in the traces. T_{rep} is the percentage of the number of (distinct) goals and plans appearing in the traces collected against the number of (distinct) goals and plans in the agent system under analysis.

Two AgentSpeak/Jason agent systems (the Gold Miner, and the Cows and Herders) were also selected for our experiments. Those systems were developed by the Jason team to compete at the Multi-Agent Programming Contest[10] in 2007 and 2009. The Gold Miner has a team of agents which explore a dynamically changing environment to avoid obstacles and collect golds. Participating agent teams compete with another agent team for the gold in the environment. The Cows and Herders is also a team of agents which need to compete with other teams to control the behavior of animals and lead them to their own corral (the winning agent team is the one that has a higher number of cows collected in its corrals). The source code of both systems are available on the Jason project website[11]

The source code of both systems are instrumented to record program event traces which are used by our dynamic analysis technique. We have

[10] `http://www.multiagentcontest.org`

[11] We have found that not many AgentSpeak programs have source code available and only the two agent programs that were selected have different versions of their code available. Versions of those programs were retrieved from the SourceForge versioning system that hosts the Jason project (`jason.sourceforge.net`).

also selected two versions of each system and used the actual changes made between them as the benchmark. Table 12 summarizes details of the initial version of each system. For example, the Gold Miner system has 9 different agents, 4 initial beliefs, 86 plans, and 809 entities (including beliefs, triggering events, context conditions and other entities in the plans' body) and was written in 597 lines of code. The execution trace representative value T_{rep} is 89.5% for the Gold Miner and 95.3% for the Cows and Herders, indicating most of the plans and goals appear in the execution traces we collected.

#	**Gold Miner**	**Cows and Herders**
Agents	9	6
Plans	86	107
Lines of code	597	992

Figure 12: Details of the Gold Miner and the Cows and Herders agent systems

Our experiments have been conducted for each of the two systems as follows. First, we extracted all the changes (i.e. PC) that were actually made to the first version of the agent system (compared to the second version). For our static analysis technique, these changes were classified according to our change taxonomy: PC consists of a set of atomic changes $< C_1, C_2, \ldots, C_n >$ that were actually changed between the two versions (i.e. the actual set A). For our dynamic analysis technique, since we consider only goals and plans, and changes made to a goal (either as a triggering event of a plan or as a subgoal part of the plan body) implies that changes have been made to that plan, we consider only plans that have been changed between the two versions. Therefore, PC consists of a set of plans $< Pl_1, Pl_2, \ldots, Pl_n >$ that were actually changed between the two versions (i.e. the actual set A).

For both techniques, we used this set to simulate the changes made to the first version. Specifically, we applied the change regarding plan Pl_1 to the first version, used our framework to compute the impact set E_1, and calculated the precision and recall for E_1 against A. Note that $E1$ is the *estimated/predicted* impact set, i.e. the set of entities that are *predicted* (by our technique) to be changed due to changing Pl_1. Since it is a prediction, some of those entities are in A (i.e. the set of entities that are *actually* changed), but some are not. A is set of entities that are actually changed between versions and is used to measure the correctness of our impact analysis techniques. Next, we applied the changes regarding $< Pl_1, Pl_2 >$ to the first version, used our framework to compute the impact set E_2, and calculated the precision and recall for E_2

against A. We continued this process for Pl_3, Pl_4 and so on to Pl_n. Each sequence of initial changes (i.e. $< Pl_1, Pl_2, \ldots, Pl_k >$, $k \leq n$) represents a change scenario in practice in which the software maintainer makes a certain initial changes to an existing version of the system and would like to know the impact of those changes. It can be seen that when the set of primary changes is PC is the actual set A (i.e. all the actual changes $< Pl_1, Pl_2, \ldots, Pl_n >$ are input our our impact analysis), the impact set estimated (i.e. E) by our analysis would contain all entities in A (consequently the recall in this case is 1) but may also contain entities that are not A (the precision is however not 1).

For the static technique, the same process was also applied to the set of atomic changes $< C_1, C_2, \ldots, C_n >$. Note that the order of selected changes was randomly established, and might affect the final outcome, which we have not explored yet. It is emphasized that changes considered in both static and dynamic techniques are the same but at different levels of granularity, e.g. for our static technique, changes to a context condition and triggering event of a plan are considered as 2 changes, whereas our dynamic technique consider them just as a change to the plan (since the dynamic technique only focuses on goals and plans).

As a sanity check for our impact analysis techniques, we have also implemented a random impact analysis involving a set of entities being randomly selected as an impact set, and compare both of our dynamic and static analysis techniques with the random analysis in terms of precision and recall.

5.2. Results

Figures 13 and 14 shows precision and recall for different scenarios of changes made to the Gold Miner and Cows and Herders agent systems using our static analysis technique. The size of the actual set A for the Gold Minder is 54 and for the Cows and Herders is 167. As can be seen in both cases, recall rises when the number of initial changes increases. For example, recall is 0.006 when there is only 1 initial change and is 1 when there are 167 initial changes made to the Cows and Herders (0.02 and 1 respectively for the Gold Miner). As more changes are made, our framework can return more entities that are potentially impacted by those changes, which consequently leads to the increase in recall. This increasing pattern is however not observed for precision. In fact, precision for the Gold Miner and the Cows and Herders fluctuates within the range 0.25 – 0.26 and 0.3 – 0.5 respectively. Recall increases due to the increase in the number of primary changes

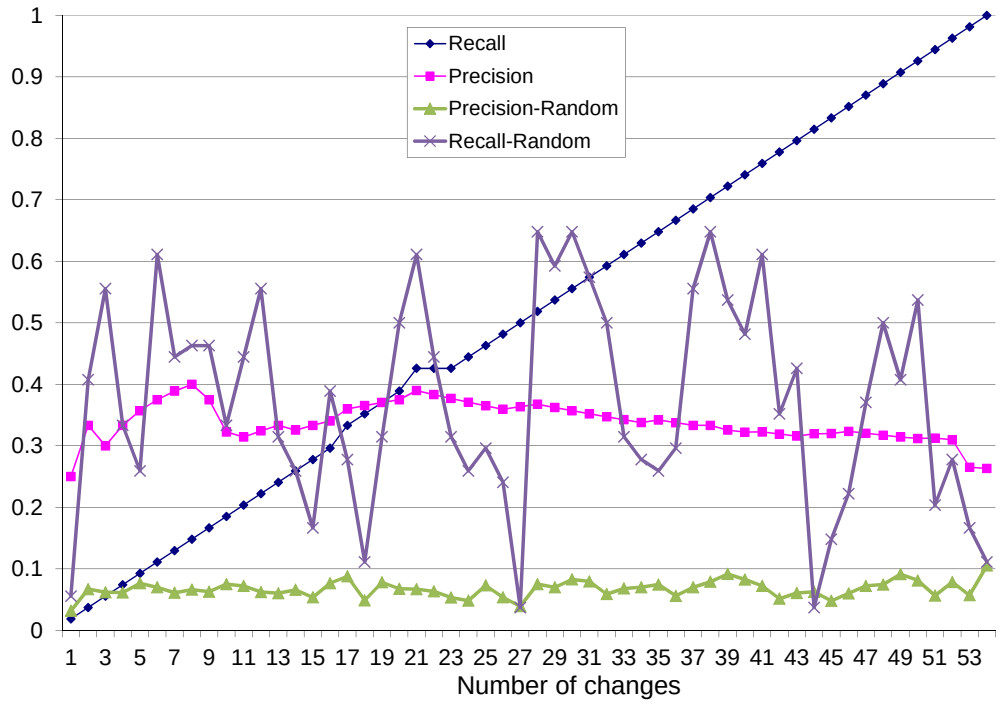

Figure 13: Precision and Recall for the Gold Miner agent system (static impact analysis)

(i.e. the input) and thus more entities are identified as being impacted by our technique (including the primary changes), but less of them are actually changed/impacted (and thus precision slightly decreases in some cases).

For our dynamic impact technique, precision and recall for different scenarios of changes made to the Gold Miner and Cows and Herders agent systems are reported in Figures 15 and 16. We ran each system (which was instrumented to record program event traces) 10 times[12], collect execution traces from all those runs and use them as input to our change impact analysis. For both cases, the execution traces we obtained contain only the execution of two agents in those systems. Therefore, we focused our analysis on the changes involving those agents between the two versions of each system. As can be seen in Figures 15 and 16, there are 9 changed plans between two versions of the Gold Miner (the size of the actual set A is 9)

[12]The program was run multiple times due to the non-deterministic (randomly generation) of the environment.

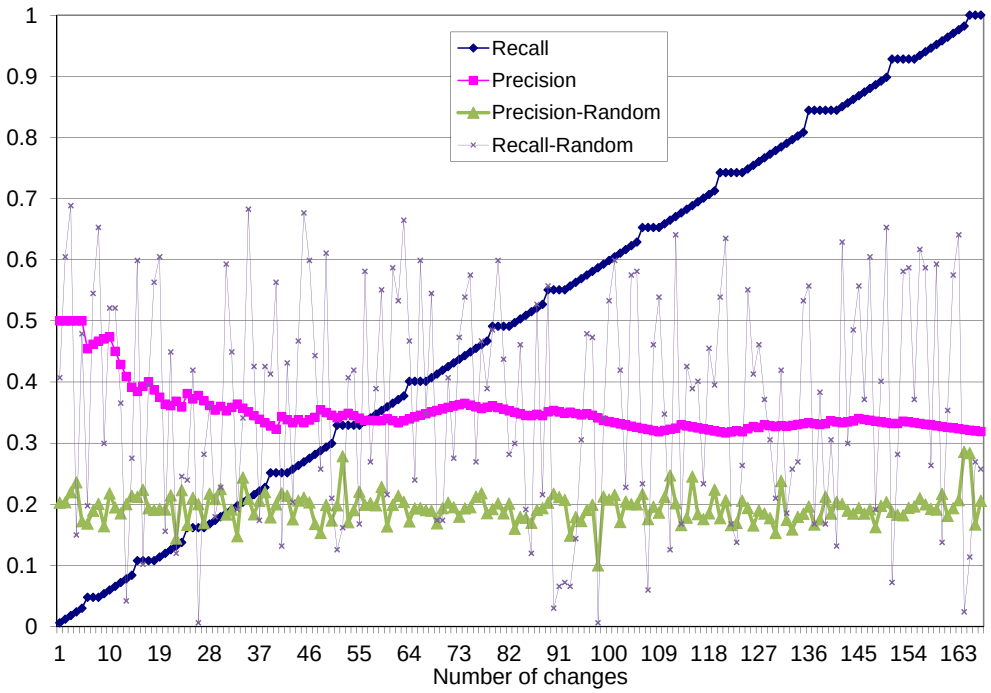

Figure 14: Precision and Recall for the Cows and Herders agent system (static impact analysis)

and 15 changed plans between two versions of Cows and Herders (the size of the actual set A is 15). In the case of the Gold Miner, precision and recall are very low after the first primary change was made to the system, but they significantly rise to 0.44 (precision) and 0.89 (recall) after the second primary change was made and remain unchanged after the 3rd, 4th, 5th, 6th, 7th, and 8th change were made. A similar pattern can also be observed in the Cows and Herders where precision and recall jump from 0.07 and 0.2 respectively after the 3rd change were made to 0.5 and 0.7 respectively after the 4th change were made.

As can be seen from the results, a similar pattern has been observed for both static and dynamic techniques: when a certain number of primary changes have been made, the tool is able to produce a stable and more accurate prediction of impacts. In addition, the average precision obtained using the static technique for the Gold Miner and the Cows and Herders are 0.25 and 0.4 respectively. Compared to the static technique, the dynamic

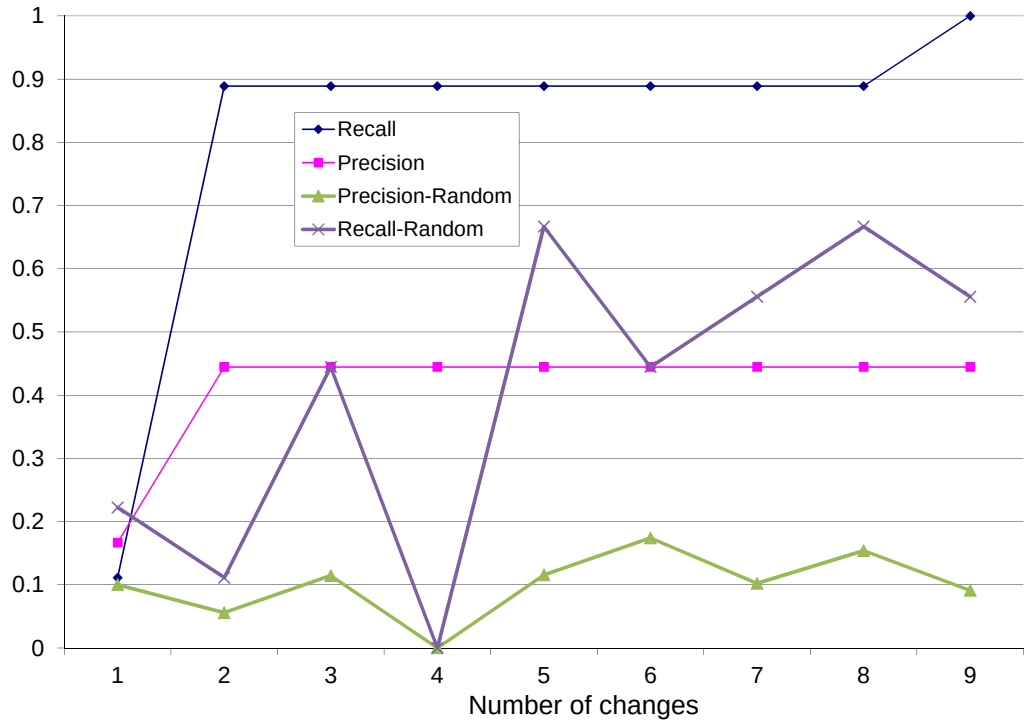

Figure 15: Precision and Recall for the Gold Miner agent system (dynamic impact analysis)

technique increased precision between 25% and 76%. The dynamic technique also produced significantly better average recall than the static technique. In fact, the static technique gives an average recall of 0.4 (Gold Miner) and 0.3 (Cows and Herders). Using our dynamic technique, we obtained 123% and 133% improvement of recall.

As can be seen from figures 13, 14, 15, and 16, the random impact analysis is really imprecise in all instances with the average precision being well below 0.2 for both source code analysis or execution trace analysis. While the precisions of the random technique remains relative constant (e.g. around 0.2 for the Cows and Herders source code analysis), its recalls fluctuate highly (e.g. from 0 to 0.7 for the Cows and Herders source code analysis) but, on average, are also below the recalls of our dynamic and static techniques. We also infer from the results that both of our techniques are effective in terms of recall if a reasonable amount of primary changes is provided. For example, while precision of our static analysis technique is always better than

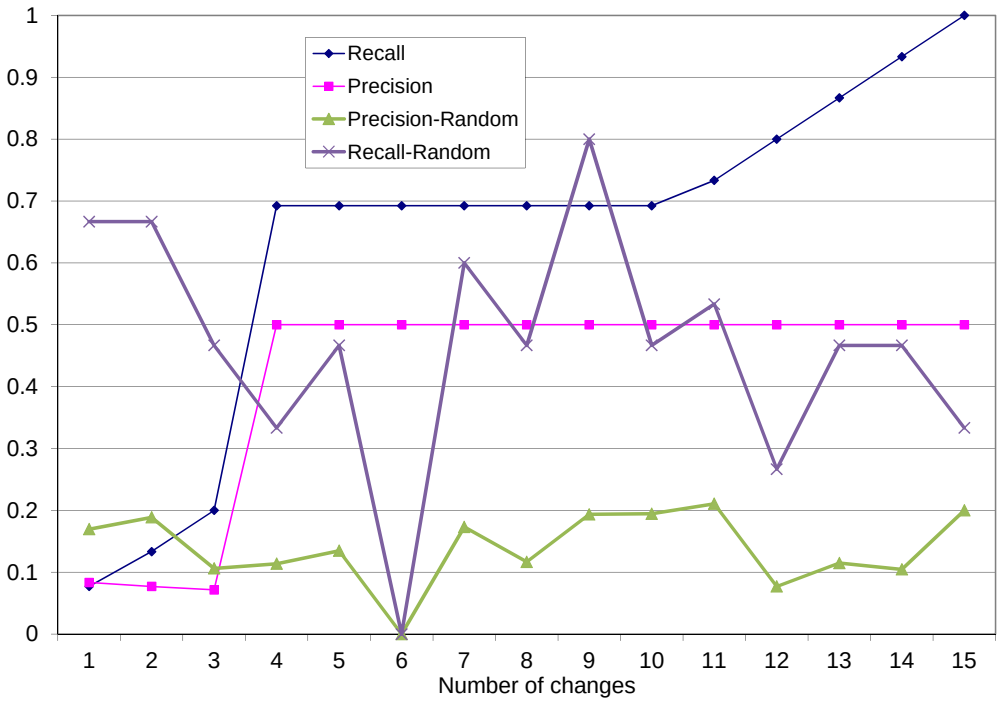

Figure 16: Precision and Recall for the Cows and Herders agent system (dynamic impact analysis)

the random analysis, recall improves over the random analysis only when the number of initial changes is more than 20 (see figure 13). The same pattern can be observed in other figures 14, 15 and 16 but with different thresholds (respectively, 50, 2 and 9 changes). This confirms with our earlier observation that as more changes are made, our techniques can return more entities that are potentially impacted by those changes, which thus leads to the increase in recall. It suggests that our impact analysis techniques would be more useful when more initial changes have been made to the system.

In general, the initial atomic change (i.e. the initial starting point or the initially changed entity) would affect the impact set that we compute due to a number of reasons. First, in the case of static impact analysis, our computation relies on the dependencies between the initially changed entity with other entities in the system. Specifically, if the initially changed entity is located in an isolated part of the system, a fewer number of entities would be impacted, comparing to the case where the initially changed entity is highly

coupled with other parts of the system. Second, in the case of dynamic impact analysis, our computation relies on the existence of the initially changed entity (plans or goals) in the execution trace(s) and also its relationships with other plans and goals (in terms of sequence of execution, interruption, concurrency and failures as discussed in the paper). In order to confirm this observation, We have performed additional experiments for each atomic change in the dynamic impact analysis of the Gold Miner agent system, i.e. we have used as each of the nine initially changed plans in the actual set as an input to our dynamic impact analysis. We have found that one of the plans does not appear in the execution traces that we collected, and consequently gives an empty impact set and one. There is also one plan which gives very low precision and recall (i.e. 0.16 and 0.11 respectively) while the remaining seven plans give the same performance in terms of precision and recall (i.e. 0.44 and 0.89 respectively). A random analysis has also performed for this case, which also indicates its imprecision (precision value being below 0.2) compared to our techniques.

5.3. Threats to validity

There are however a number of issues relating to the validity of our study. In terms of the internal validity, perhaps the biggest issue is that we have not conducted an evaluation with human software engineers, but instead have used a "simulated user" that follows a maintenance process. This maintenance process however does represent real changes in real agent systems. Regarding the external validity, we have studied only two agent systems and a set of changes, which may not be representative of agent systems and changes generally. In addition, the execution traces we collected may not represent different types of operational profiles of agent systems in practice. Therefore, further experiments on larger-scale agent systems, change distributions, and types of operational profiles are needed for generalisation of our results and are in fact an important part of our future work. It should however be noted that since most of BDI-style agent programs consist of plans and goals, we expect that our results would generalise to other BDI agent-oriented programming languages. Although the change taxonomy and classification of dependencies presented in this paper is specific to AgentSpeak programs, our static impact analysis approach can be generalized to other BDI agent programming languages. For example, in order to apply our approach to GOAL [32] (or 2APL [33]) programs, one would need to define a taxonomy of atomic changes that can be made to GOAL programs and a classification

of dependencies specific to GOAL programs. Our dynamic impact analysis approach relies on execution traces of plans and goals which are quite generic and are commonly found in different BDI-based agent programs (including GOAL and 2APL).

Our approach can also be extended to agent-based simulation platforms, which is based on BDI for the implementation of individual agents (e.g. the MAS-SOC [34] platform for agent-based social simulation uses AgentSpeak). Similarly, our approach can be extended to cover impact of changes in the environment to the agents' internals. To do so, the taxonomy of changes would need to be extended to cover changes in the environment (e.g. percepts and effects of agent actions on perceptible properties of the environment in Jason, or artifacts and their operations as in the CArtAgO[13] environment). In addition, dependencies between the environment and an agent's internal should also be established (e.g. dependencies between environmental percepts and triggering events – both are literals in Jason). Dependencies between plans and other types of belief should also be established, should the framework be extended to cover multiple belief sources.

Although the dynamic technique has shown better results in our experiments, we acknowledge that BDI systems potentially have large behaviour space (as shown in a recent study [35]) and thus dynamic analysis may underestimate the impact(i.e. low recall). This is because each execution may give different execution trace depending on context dependent selection of goals and plans and the results produced by dynamic analysis will not include impacts for parts of the program that have not been executed. Therefore, dynamic impact analysis is particularly useful in cases where safety is not required because they provide more precise information about impact with regard to a specific program behaviour than static analyses. If the environment in which the agent will operate in the future is unknown, static analysis, by its nature, considers all the possibilities, which seems more appropriate. Further experiments of such cases are needed to confirm this.

6. Related work

As mentioned earlier, there has been very limited work on change impact analysis for agent systems. In contrast, there have been a proliferation

[13] `http://cartago.sourceforge.net/`

of techniques proposing to support change impact analysis of procedural or object-oriented systems (seminal work presented in [18] or more recent work such as [19, 20, 36, 30]), which can be classified into two groups: static and dynamic analysis. Static impact analysis techniques (e.g. [37] for object-oriented systems or a number of techniques reported in [18]) usually perform either program slicing or graph traversals (by analysing the source code) to compute impact sets. These techniques are considered to be conservative in that they consider all possible program inputs and behaviours. Results produced by static analysis may have enormous impact sets, which are sometimes unnecessary or even too large to be of practical use. Recently, the work in [36] proposed a variable granularity approach to improve the precision of an impact analysis by allowing the software engineer to choose a level of granularity (e.g. classes, or class members or code fragments) during an iterative process, i.e. the software engineer interactively works with the tool in each step to correct the mistakes (i.e. false positives) made by the tool. Our static analysis technique can be extended in this direction to variably cover a range of granularity levels (e.g. agents or plans or literals) in agent systems. Recent techniques (e.g. [38, 39, 40]) leverage the emerging mining software repositories technology to make use of the historical co-changes to compute the impact (i.e. entities co-changing frequently in the past are very likely to co-change in the future).

There has been a range of work on slicing logic programs. In particular, Zhao et. al. [22] proposed a Literal Dependency Net (LDN) representing different dependencies in concurrent logic programs that are in the form of Guarded Horn Clauses (GHC) [41]. An LDN diagraph represents four types of dependencies between literals in a logic program: control, unification, data, synchronization, and communication dependencies. Since an AgentSpeak plan has the exact same structure of a GHC, Bordini et. al. [23] adopted the LDN proposed in [22] to slice agent programs written in AgentSpeak, i.e. identifying a set of plans that is equivalent to the original agent system with respect to a given slicing property. Their technique is used to reduce the state-space in verifying agent behaviours.

Our static impact analysis technique is also inspired by notions and ideas proposed by Zhao et. al. [22]. In particular, several types of dependency within a AgentSpeak multi-agent system are adapted from their LDN, e.g. the dependency between a subgoal and a triggering event is a form of unification dependency. However, the execution model of GHC and AgentSpeak are fundamentally different. For instance, the behaviour of an agent to a

particular external event is captured in a single intention, as a stack of committed sub-behaviours, which provides a global coherence that is absent in GHC. Our work is also different from the work in [23] in several aspects. Firstly, their work aims at selecting plans based on their impact on the truth predicate that appears in a slicing property specification. We, on the other hand, target at collecting all parts of plans as well as beliefs that are impacted by a change. Secondly, their work does not explicitly deals with the inter-communication between agents as in our work.

In contrast to static analysis, dynamic analysis techniques (e.g. [19, 20]) compute impacts using data obtained from executing a program. Our work in this paper belongs to this category and is inspired by the well-known PathImpact technique [20] for procedural and object-oriented systems (in particular the forward and backward walks through an execution trace to determine impact sets). Dynamic analysis results are more practically useful since they better reflect how the system is actually being used, and consequently do not have computed impacts derived from impossible system behaviour. However, due to their dependency on the inputs used to execute the program, the results produced by dynamic analysis will not include impacts for parts (e.g. functions) of the program that are not executed. This might be an issue for agent systems since a recent quantitative study [35] has shown that the behaviour space of BDI agents is large. Recently, a hybrid impact analysis technique proposed in [30] for object-oriented systems based on both static and dynamic analysis produces positive results in terms of reducing false-negatives.

7. Conclusions and future work

Although there have been an increasing number of intelligent agent applications, very little work has been done for supporting the maintenance and evolution of agent systems. In this paper, we aimed towards filling this gap by proposing two different approaches to change impact analysis for agent systems, particularly the well-known and widely-used BDI agents. In the static impact analysis approach, we have defined a taxonomy of dependencies that capture various dependent relationships within an agent system, including both inter-agent dependencies and intra-agent dependencies. We have also proposed a taxonomy of atomic changes and an algorithm which computes the impact of a change in an agent system. Our static technique

has been implemented in AgentCIA, a plugin for the Jason IDE, a well-known platform for agent development.

In contrast to the static technique which requires an analysis of source code, our dynamic analysis technique analyses past execution traces of a BDI agent system to determine a set of plans and goals that are impacted by a change made to a plan or goal, for a given programs behaviour. We have examined different types of traces that exhibit typical behaviours of an agent system such as pursuing multiple goals simultaneously, handling failures, and interrupting plan execution when higher priority events occur. An empirical validation on two real agent systems has been performed, indicating the effectiveness of our approach given a reasonable number of primary changes performed. The experimental results also showed that our dynamic technique improved both recall (123% and 133%) and precision (between 25% and 76%) compared to the static technique.

An important part of our future work involves investigating techniques to improve the effectiveness of our approach, i.e. increasing precision and recall. This includes utilising the categories of changes and analysing the characteristics of these categories: what kinds of changes will or will not affect other parts of the system. We would also like to explore if there are propagating entities in agent systems – those entities do not change but they propagate the changes to their neighbours. In addition, we plan to adapt the guidelines proposed in [42] including the use of impact semantics and structural constraints to structure our impact results. We also plan to explore a set of metrics which quantitatively represents the degree of the impact. Our future work would involve performing further large-scale empirical studies on other (large) types of agent systems to have more understanding of the efficiency and effectiveness of our technique. An implementation of the program slicing technique proposed in [23] to slice AgentSpeak programs (for model checking purposes) was not readily available and an appropriate implementation/adaption for impact analysis will require considerable effort to create. Therefore, we would also like to investigate the accuracy of their technique in computing impacts with our approach. Approaches like model checking (e.g. [23]) have the advantage of exhautive coverage which means all possible executions will be explored. Therefore, exploring a hybrid technique which combines such approaches with our dynamic impact analysis approach is also part of our future work. In addition, since it is important to explore how changes from the organization and/or the environment affect the agent system, we would like to investigate how our approach can also consider other

programming constructs (e.g. the organization, environment, etc. in JaCaMo [43]). We also plan to develop an algorithm to efficiently compress execution traces (similarly to the technique proposed with PathImpact [20]) to save spaces. Finally, an important part of our future work involves investigating a hybrid technique which combines dynamic and static impact analysis for agent systems.

Acknowledgements

We would like to thank Michael Winikoff for his comments on this work, and Vinh San To for helping with the implementation of our impact analysis techniques.

References

[1] H. Van Vliet, Software engineering: principles and practice, 2nd Edition, John Wiley & Sons, Inc., 2001, iSBN 0471975087.

[2] J. Sutherland, Business objects in corporate information systems, ACM Computing Surveys 27 (2) (1995) 274–276. doi:http://doi.acm.org/10.1145/210376.210394.

[3] C. Jones, Applied Software Measurement: Global Analysis of Productivity and Quality, McGraw-Hill, Inc., New York, NY, USA, 2008.

[4] M. Wooldridge, An Introduction to MultiAgent Systems, John Wiley & Sons (Chichester, England), 2002, iSBN 0 47149691X.

[5] I. Mathieson, S. Dance, L. Padgham, M. Gorman, M. Winikoff, An open meteorological alerting system: Issues and solutions, in: V. Estivill-Castro (Ed.), Proceedings of the 27th Australasian Computer Science Conference, Dunedin, New Zealand, 2004, pp. 351–358.

[6] B. Burmeister, M. Arnold, F. Copaciu, G. Rimassa, BDI-Agents for agile goal-oriented business processes, in: Padgham, Parkes, Müller, Parsons (Eds.), Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, 2008, pp. 37–44.

[7] L. Monostori, J. Váncza, S. Kumara, Agent based systems for manufacturing, CIRP Annals-Manufacturing Technology 55 (2) (2006) 697–720.

[8] M. Pěchouček, V. Mařík, Industrial deployment of multi-agent technologies: review and selected case studies, Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) 17 (2008) 397–431. doi:10.1007/s10458-008-9050-0.

[9] S. Munroe, T. Miller, R. A. Belecheanu, M. Pěchouček, P. McBurney, M. Luck, Crossing the agent technology chasm: Lessons, experiences and challenges in commercial applications of agents, Knowledge Engineering Review 21 (4) (2006) 345–392. doi:10.1017/S0269888906001020.

[10] S. S. Benfield, J. Hendrickson, D. Galanti, Making a strong business case for multiagent technology, in: AAMAS '06: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, ACM, New York, NY, USA, 2006, pp. 10–15. doi:http://doi.acm.org/10.1145/1160633.1160938.

[11] A. S. Rao, M. P. Georgeff, BDI agents: From theory to practice, in: V. R. Lesser, L. Gasser (Eds.), Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA, The MIT Press, 1995, pp. 312–319.

[12] T. Mens, Introduction and roadmap: History and challenges of software evolution, in: T. Mens, S. Demeyer (Eds.), Software Evolution, Springer Berlin Heidelberg, 2008.

[13] H. K. Dam, M. Winikoff, An agent-oriented approach to change propagation in software maintenance, Journal of Autonomous Agents and Multi-Agent Systems 23 (3) (2011) 384–452. doi:10.1007/s10458-010-9163-0.

[14] L. Padgham, M. Winikoff, Developing intelligent agent systems: A practical guide, John Wiley & Sons, Chichester, 2004, ISBN 0-470-86120-7.

[15] H. K. Dam, M. Winikoff, Supporting change propagation in UML models, in: Proceedings of the 26th IEEE International Conference on Software Maintenance, ICSM '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–10.

[16] H. K. Dam, L.-S. Le, A. Ghose, Supporting change propagation in the evolution of enterprise architectures, in: Proceedings of the 2010 14th IEEE International Enterprise Distributed Object Computing Conference, EDOC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 24–33. doi:http://dx.doi.org/10.1109/EDOC.2010.23.
URL http://dx.doi.org/10.1109/EDOC.2010.23

[17] H. K. Dam, A. Ghose, Supporting change propagation in the maintenance and evolution of service-oriented architectures, in: Proceedings of the 2010 Asia Pacific Software Engineering Conference, APSEC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 156–165.

[18] R. Arnold, S. Bohner, Software Change Impact Analysis, IEEE Computer Society Press, 1996.

[19] T. Apiwattanapong, A. Orso, M. J. Harrold, Efficient and precise dynamic impact analysis using execute-after sequences, in: ICSE '05: Proceedings of the 27th International Conference on Software Engineering, ACM, New York, NY, USA, 2005, pp. 432–441. doi:http://doi.acm.org/10.1145/1062455.1062534.

[20] J. Law, G. Rothermel, Whole program path-based dynamic impact analysis, in: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2003, pp. 308–318.

[21] A. S. Rao, AgentSpeak(L): BDI agents speak out in a logical computable language, in: W. V. de Velde, J. Perrame (Eds.), Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Springer Verlag, 1996, pp. 42–55, lNAI, Volume 1038.

[22] J. Zhao, J. Cheng, K. Ushijima, Program dependence analysis of concurrent logic programs and its applications, in: Proceedings of the 1996 International Conference on Parallel and Distributed Systems, ICPADS '96, IEEE Computer Society, Washington, DC, USA, 1996, pp. 282–291.

[23] R. H. Bordini, M. Fisher, M. Wooldridge, W. Visser, Property-based slicing for agent verification, Journal of Logic and Computation 19 (2009) 1385–1425.

[24] H. K. Dam, A. Ghose, Automated change impact analysis for agent systems, in: Proceedings of the 27th IEEE International Conference on Software Maintenance, ICSM '11, IEEE, Washington, DC, USA, 2011, pp. 33–42.

[25] H. K. Dam, A. Ghose, Dynamic change impact analysis for maintaining and evolving agent systems (extended abstract), in: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems, 2012, pp. 1433–1434.

[26] R. H. Bordini, J. F. Hübner, M. Wooldridge, Programming multi-agent systems in AgentSpeak using Jason, Wiley, 2007, ISBN 0470029005.

[27] M. E. Bratman, Intentions, Plans, and Practical Reason, Harvard University Press, Cambridge, MA, 1987.

[28] P. Busetta, N. Howden, R. Rönnquist, A. Hodgson, Structuring BDI agents in functional clusters, in: Agent Theories, Architectures, and Languages (ATAL-99), Springer-Verlag, 2000, pp. 277–289, lNCS 1757.

[29] R. H. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (Eds.), Multi-Agent Programming: Languages, Platforms and Applications, Springer, 2005.

[30] M. C. O. Maia, R. A. Bittencourt, J. C. A. de Figueiredo, D. D. S. Guerrero, The hybrid technique for object-oriented software change impact analysis, in: Proceedings of the 14th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2010, pp. 252–255.

[31] J. Thangarajah, L. Padgham, M. Winikoff, Detecting and avoiding interference between goals in intelligent agents, in: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI), 2003, pp. 721–726.

[32] K. Hindriks, F. de Boer, W. van der Hoek, 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.

[33] M. Dastani, 2APL: a practical agent programming language, Autonomous Agents and Multi-Agent Systems 16 (3) (2008) 214–248.

[34] R. H. Bordini, A. C. d. R. Costa, J. F. Hübner, F. Y. Okuyama, A. F. Moreira, R. Vieira, MAS-SOC: a social simulation platform based on agent-oriented programming, Journal of Artificial Societies and Social Simulation 8 (3) (2005) 7.

[35] M. Winikoff, S. Cranefield, On the testability of BDI agents, in: Proceedings of the 8th European Workshop on Multi-Agent Systems (EUMAS2010), 2010.

[36] M. Petrenko, V. Rajlich, Variable granularity for improving precision of impact analysis, in: The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009, IEEE Computer Society, 2009, pp. 10–19.

[37] L. Li, J. Offutt, Algorithmic analysis of the impacts of changes to object-oriented software, in: Proceedings of the International Conference on Software Maintenance (ICSM' 96), IEEE, 1996, pp. 171–184.

[38] A. E. Hassan, R. C. Holt, Predicting change propagation in software systems, in: ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 2004, pp. 284–293.

[39] H. Malik, A. E. Hassan, Supporting software evolution using adaptive change propagation, in: ICSM '08: Proceedings of the 24th IEEE International Conference on Software Maintenance, Beijing, China, 2008.

[40] H. Kagdi, J. I. Maletic, B. Sharif, Mining software repositories for traceability links, in: Proceedings of the 15th IEEE International Conference on Program Comprehension, ICPC '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 145–154. doi:10.1109/ICPC.2007.28.
URL http://dx.doi.org/10.1109/ICPC.2007.28

[41] K. Ueda, Guarded Horn Clauses, in: Proceedings of the 4th Conference on Logic Programming '85, Springer-Verlag, London, UK, 1986, pp. 168–179.

[42] S. A. Bohner, Software change impacts - an evolving perspective, in: ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02), IEEE Computer Society, Washington, DC, USA, 2002, pp. 263–272.

[43] A. Ricci, M. Viroli, A. Omicini, Give agents their artifacts: the a&a approach for engineering working environments in mas, in: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems, ACM, New York, NY, USA, 2007, pp. 613–615. doi:10.1145/1329125.1329308.