Automated change impact analysis for agent systems

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

Abstract—**Intelligent agent technology has evolved rapidly over the past few years along with the growing number of agent systems in various domains. Although a substantial amount of work in agent-oriented software engineering has provided methodologies for analysing, designing and implementing agent-based systems, recent studies have highlighted that there has been very little work on maintenance and evolution of agent-based systems. A critical issue in software maintenance and evolution is change impact analysis: determining the potential consequences of a proposed change. There has been a proliferation of techniques proposed to support change impact analysis of procedural or object-oriented systems, but to the best of our knowledge, no such an effort has been made for agent-based software. In this paper, we fill this gap by proposing a framework to support change impact analysis for agent systems. At the core of our framework is the taxonomy of atomic changes which can precisely capture semantic differences between versions of an agent system. We also present a change impact model in the form of an intra-agent dependency graph that represents various dependencies within an agent system. An algorithm to compute the set of entities impacted by a change is also presented. The proposed techniques have been implemented in AgentCIA, a change impact analysis plugin for Jason, one of the most well-known agent programming platforms.**

I. Introduction

A software agent [1] 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, business process management, holonic manufacturing, e-commerce and information management [2, 3]. The number of agent-based applications continues to increase since there are compelling reasons to use intelligent agent technology. In fact, a recent study by a large company that used agent technology to develop complex software applications has shown an average productivity gain of over 350% [4].

Agent systems, like conventional software systems, undergo rapid changes during their evolution 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 [5]). 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.

Previous work [6] has proposed a framework that supports change propagation in the evolution of agent-oriented design models. That framework has also been extended to deal with other design types (e.g. UML models in [7]). Change propagation techniques, however, focus on implementing a change by propagating changes to maintain consistency within the software. Another important aspect of dealing with changes is change impact analysis, which aims to assess the extent of a change, i.e. the entities/components that are potentially impacted by the change, and consequently predicts the cost and complexity of the change (before implementing it).

The process of change impact analysis consists of two major steps [8]. Firstly, the software engineer examines the change request and identifies the entities initially affected by the change. Next, the software engineer identifies other entities in the program 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 establishing the feasibility of a change in terms of determining whether the change is to be undertaken. If the change is rejected, then the request is returned to its origin for revision or more discussion. Otherwise, the change process enters the next phase which involves the implementation of the change (i.e. change propagation).

Although notions and ideas from a large body of work addressing change impact analysis for classical software systems (e.g. [8–11]) can be adapted, agent systems with their distinct characteristics and architectures introduce new problems in software maintenance. For instance, object-oriented software deals with classes, methods and fields whereas a typical agent-based software consists of agents, plans and beliefs. Since most of agent programs (e.g. AgentSpeak[1] [12]) 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. [14]) however, focuses on program slicing. In fact, a recent proposal by Bordini et. al. [15] also adopts the approach in [14] 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, we fill that gap by proposing a framework to support change impact analysis in agent systems, specifically those written in the widely-used AgentSpeak programming language. More specifically, we propose a taxonomy of changes for an agent system so that any change request to an agent system can be decomposed into a set of atomic changes which are useful to the impact analysis. Furthermore, our change impact analysis framework has a classification of various dependencies existing in an agent system, including within an agent and between agents. This taxonomy of dependencies is the basis for building a dependency graph for an agent system. We also propose an algorithm to traverse such a graph to trace relationships and consequently compute the impact of a change. Our framework has been implemented in AgentCIA, a change impact analysis plugin for Jason[2] [16], the most well-known and widely-used development platform for AgentSpeak. Although Jason 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 II), and give an example agent program (section III). In section IV, we describe our change propagation framework, firstly presenting a taxonomy of changes and a classification of dependencies in an agent system, and then discussing how they are used to compute the impact of a change. We describe an prototype implementation of our framework and discuss the experiments we have conducted to evaluate it in section V. Finally, we discuss related work in section VI, and then conclude and outline some directions for future work in section VII.

II. Background

Among many agent programming languages (refer to [17] 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 [12] – one of the most popular BDI agent-oriented languages. Jason is also a well-known platform for the development of multi-agent systems. In this section, we will briefly describe the BDI architecture and the syntax of AgentSpeak.

A practical goal-oriented BDI-style agent is basically a reactive planning system which selects and executes plans to achieve its goals or to handle events in a systematic manner. Events are significant occurrences that the agent should respond to in some way. Most BDI agent implementation platforms model the change associated with the adoption of new (sub)goals as events. BDI agents have a collection of pre-defined plan recipes (or types), usually referred to as a *plan library*. Each plan consists of: (a) an invocation condition which defines the event that triggers this plan (i.e. the event that the plan is *relevant* for); (b) a context condition (usually referring to the agent's beliefs) which defines the situation in which the plan is *applicable*, i.e. it is sensible to use the plan in a particular situation; and (c) a plan body containing a sequence of primitive actions and subgoals (which can trigger further plans) that are performed for plan execution to be successful.

We describe here a typical execution cycle that implements the decision-making of an agent following an implementation of the BDI architecture (e.g. AgentSpeak). The cycle can be viewed as consisting of the following steps. 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 that are relevant (i.e. match the invocation condition) for handling the event (by looking at the plans' definition). The agent then determines the subset of the relevant plans 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. The agent selects one of the applicable plans. The agent then executes the selected plan by performing its actions and subgoals. 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 subgoal 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.

AgentSpeak was developed based on the BDI architecture. Figure 1 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

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

[2]http://jason.sourceforge.net

$$\begin{array}{lcl}
ag & ::= & bs\ \ ps \\
bs & ::= & at_1.\ \ldots\ at_n. \\
at & ::= & P(t_1,\ldots,t_n) \\
ps & ::= & p_1\ \ldots\ p_n \\
p & ::= & te:\ ct\ \leftarrow h \\
te & ::= & +at \mid -at \mid +g \mid -g \\
ct & ::= & true \mid l_1 \& \ldots \& l_n \\
h & ::= & true \mid f_1;\ \ldots;\ f_n \\
l & ::= & at \mid \neg\ at \\
f & ::= & A(t_1,\ldots,t_n) \mid g \mid u \\
g & ::= & !at \mid ?at \\
u & ::= & +at \mid -at
\end{array}$$

Fig. 1. The concrete syntax of AgentSpeak(L) (adapted from [15])

$A(t_1,\ldots,t_n)$), 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 to test whether at is a true belief or not).

Jason [16] 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.

III. 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, 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), Reply);
        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

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

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

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.

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 L 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 L, 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 lot.

IV. Change impact analysis framework

In this section, we describe a change impact analysis framework for an agent program (see figure 2). The process of change impact analysis consists of two major steps and involves two actors: the software engineer and the tool which is an implementation of our framework. Firstly, the software engineer examines the change request, identifies the entities

[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.

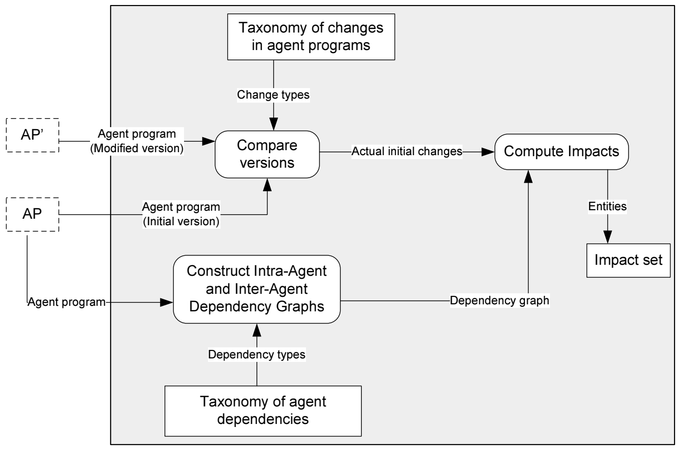
Taxonomy of changes in agent programs
Change types
AP'
Agent program (Modified version)
Compare versions
Actual initial changes
Compute Impacts
AP
Agent program (Initial version)
Entities
Impact set
Construct Intra-Agent and Inter-Agent Dependency Graphs
Agent program
Dependency graph
Dependency types
Taxonomy of agent dependencies

Fig. 2. A change impact analysis framework for agent systems

(e.g. agents, plans, goals, beliefs, etc.) in the existing agent program (i.e. AP in figure 2) initially affected by the change, and makes changes to those entities. Such primary changes result in a new version AP' of the agent program. Our framework then compares the two versions and automatically detects all the primary changes previously performed. This feature of our framework would also eliminate the overhead of specifying each and every change by the software engineer. Our framework 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 an instance of exactly one change type in the taxonomy.

In the next step, the software engineer relies on our framework to determine the impact of the primary changes that they have initially made to the agent system. Our framework then automatically constructs 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). Our framework then uses 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 key components of our change impact analysis framework in details.

A. Change taxonomy of agent systems

We now describe different types of changes in an AgentSpeak program and their relationships (refer to figure 3). 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 III), 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. Changing a literal is in turn classified into three types[6]: changing literal name, adding a new term, and removing an existing term. Since we consider a plan's triggering event as its identifier, changing the name of the literal associated with the triggering event means deleting the plan and adding a new plan.

Furthermore, since a context condition is a conjunction or disjunction of literals (e.g. $checking(slots) \land \neg\ garbage(r1)$), we consider changing a context condition as involving either adding a literal, deleting a literal or changing an existing literal. 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 an deletion (e.g. of an existing action) and addition (e.g. of a new action). Basically, using the name of a literal as one of an entity's identifiers is the way our framework can keep track of the identity of entities across different versions of an agent program.

Figure 3 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.

[6]The classification of changes made to a literal is not shown in figure 3.

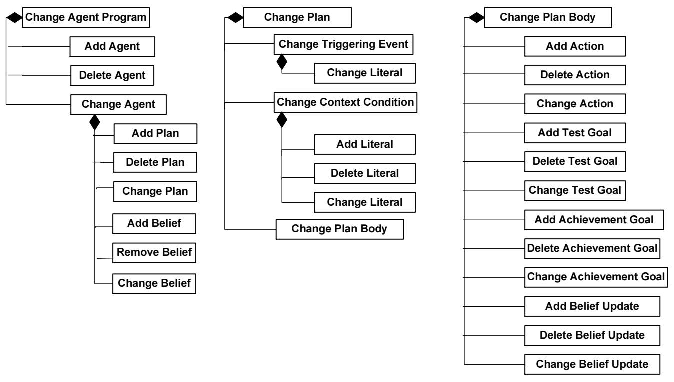

Fig. 3. A taxonomy of changes for AgentSpeak programs

B. 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 III to illustrate different types of dependencies.

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 literal are unifiable. 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 $back$).

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.

- A predicate in the context condition depends on the triggering event if their associated literals shares a common variable. 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 variable. 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 variable. 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) 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.

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 III 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 [16].

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.
- 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.
- tellHow: *msg* is a plan and consequently dependencies between this plan and other entities in agent r can be applied.
- 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.
- 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 $r2$.

[7] *askAll* can be treated similarly.

C. 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. 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* $E_i \subseteq N_i \times U$ *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*).*

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. 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.

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 entities 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 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
           $IES ::= IES \cup \{E\}$
6          if CT is not a type of Addition then
7              Let N be the node of the graph that represents E
8              $IES ::= IES \cup ComputeImpact(N, G)$
9          end
10     end
11 end
```

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              $IES ::= IES \cup \{E_M\} \cup ComputeImpact(M, G)$

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 [18] based on a (weak) assumption that if direct impacts have high potential for being true, then those further away will be less likely. 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 III. 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 IV-B 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.

V. Evaluation

In this section we describe a prototype implementation of our framework and the experiments we have conducted to measure the effectiveness of our approach.

A. Implementation

The Jason IDE on the Eclipse platform provides an environment for developing AgentSpeak agent systems, and also supports plugin development. A prototype of our framework has been implemented in AgentCIA, a change impact analysis plugin for the Jason IDE. There are three major parts in AgentCIA: parser, analyzer and viewer. The parser is responsible for extracting the AgentSpeak code and building an Inter-Agent DG. The parser also compares a given an AgentSpeak program and its modified version to detect the changes, and interprets and classifies them into atomic change elements. Note that changes that do not belong to the defined taxonomy of changes are not included in the change impact analysis.

The analyzer is mainly responsible for calculating the change impact. It is implemented in such a way that changing the impact analysis technique will not affect the framework as long as the analyzer interface remains the same. Results from the analyzer are passed to the viewer for displaying and analyzing. AgentCIA presents the impact results in a tree view showing the hierarchy of all entities in all agents in the system as shown in figure 4. The entities that have initially been modified by the software engineer are annotated and shown

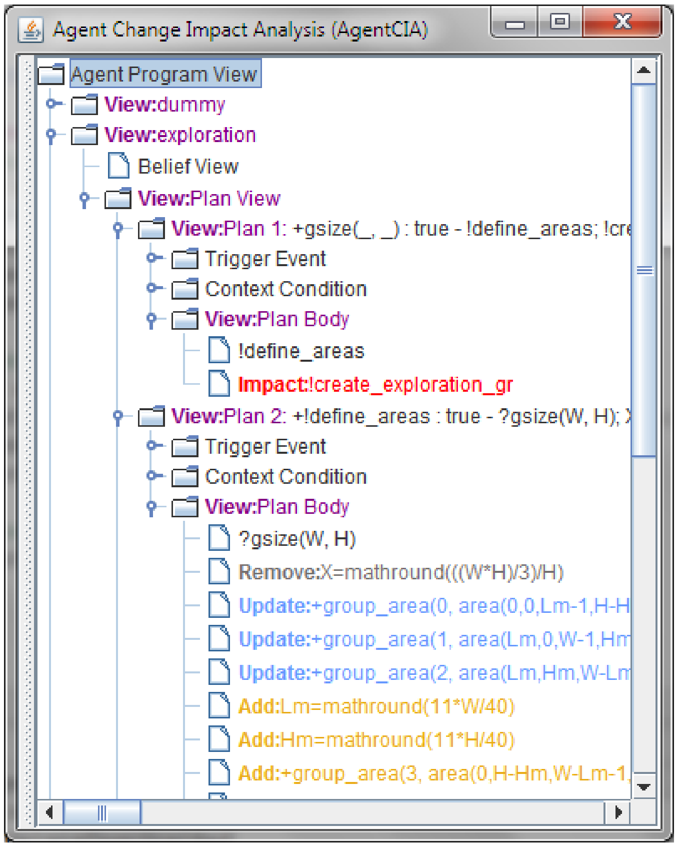
```
Agent Change Impact Analysis (AgentCIA)
Agent Program View
  View:dummy
  View:exploration
    Belief View
    View:Plan View
      View:Plan 1: +gsize(_, _) : true - !define_areas; !cre
        Trigger Event
        Context Condition
        View:Plan Body
          !define_areas
          Impact:!create_exploration_gr
      View:Plan 2: +!define_areas : true - ?gsize(W, H); X
        Trigger Event
        Context Condition
        View:Plan Body
          ?gsize(W, H)
          Remove:X=mathround(((W*H)/3)/H)
          Update:+group_area(0, area(0,0,Lm-1,H-H
          Update:+group_area(1, area(Lm,0,W-1,Hm
          Update:+group_area(2, area(Lm,Hm,W-Lm
          Add:Lm=mathround(11*W/40)
          Add:Hm=mathround(11*H/40)
          Add:+group_area(3, area(0,H-Hm,W-Lm-1,
```

Fig. 4. A snapshot of AgentCIA for Jason IDE

in different colors depending on whether a change is addition, modification or deletion, whereas the entities that are impacted by those changes are annotated as "impact" and highlighted in red.

B. Experiments

Having implemented the change impact analysis framework for agent systems, we would now like to perform an empirical evaluation of the effectiveness of our approach. The key question is how well this framework works in practice and, specifically, how useful is it likely to be to a practising software engineer who is maintaining and evolving an agent system. The effectiveness measurement we used involves two sets: the set of impacted entities obtained by an impact analysis technique (i.e. the expected set E) and the set of entities actually impacted by a given change (i.e. the actual set A). False-positives are entities that are in E but not in A while false-negatives are entities that are in A but not in E. In order to measure the effectiveness of our change impact analysis technique, we use two 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. [18]). Precision is defined as the fraction of the estimated impacted entities (i.e. E) which was actually modified (i.e. A), i.e. $Precision = \frac{|E \cap A|}{|E|}$.

#	Gold Miner	Cows and Herders
Agents	9	6
Initial beliefs	4	2
Plans	86	107
Entities	809	864
Lines of code	597	992

Fig. 5. Details of the Gold Miner and the Cows and Herders agent systems

Recall is the fraction of the actually modified entities (i.e. A) that was estimated (i.e. E), i.e. $Recall = \frac{|E \cap A|}{|A|}$. A perfect precision score of 1.0 means that every entity identified as being impacted by an impact analysis technique was actually changed whereas a perfect recall score of 1.0 means that all changed entities were identified as being impacted by the technique.

We have selected two agent systems which were developed by the Jason team to compete at the Multi-Agent Programming Contest[8] in 2007 and 2009. The first one (referred to as 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 second agent system (referred to as the Cows and Herders) used in our experiment 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 and for each of them we have selected two versions. Table 5 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.

Our experiments have been conducted for each of the two systems as follow. First, we extracted all the changes (i.e. CS) that were actually made to the first version of the agent system (compared to the second version). These changes classified according to our change taxonomy: CS consists of a set of atomic changes $< C_1, C_2, \ldots, C_n >$. We also obtained the set of entities that were actually changed between the two versions (i.e. the actual set A). We then applied the change C_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. Next, we applied the changes $< C_1, C_2 >$ 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 C_3, C_4 and so on to C_n. Each sequence of initial changes (i.e. $< C_1, C_2, \ldots, C_k >$, $k \leq n$) represents a change scenario in practice in which the software engineer makes a certain initial changes to an existing version of the system and would like to know the impact of those changes.

Figures 6 and 7 shows precision and recall for different

[8]http://www.multiagentcontest.org

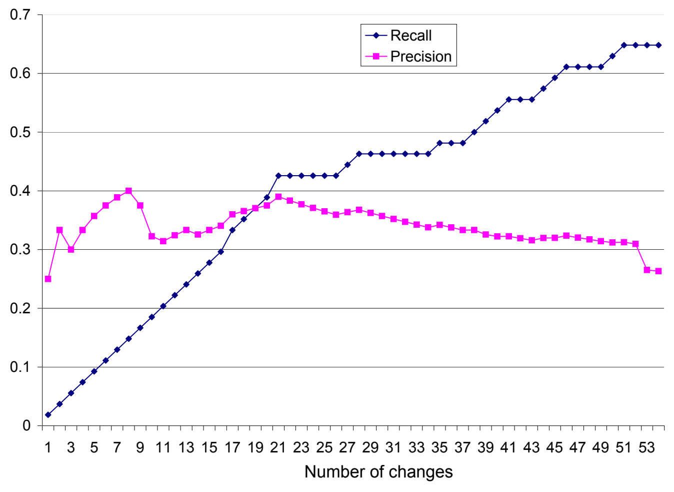

Fig. 6. Precision and Recall for the Gold Miner agent system

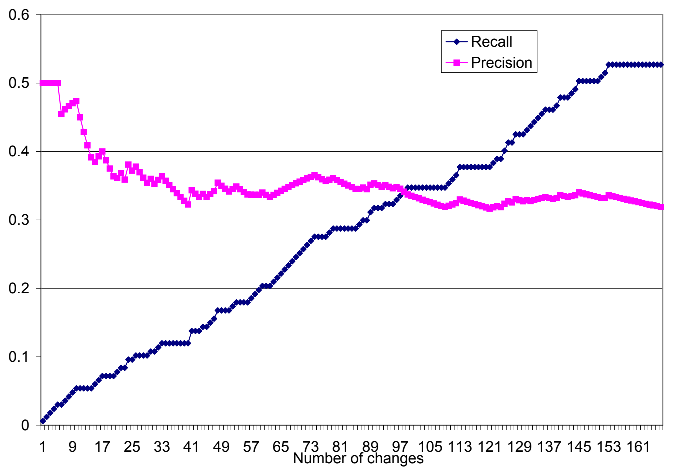

Fig. 7. Precision and Recall for the Cows and Herders agent system

scenarios of changes made to the Gold Miner and Cows and Herders agent systems. 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 0.526 when there are 167 initial changes made to the Cows and Herders (0.02 and 0.65 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. The evaluation's results indicated that our approach is relatively effective given that a reasonable amount of primary changes are provided.

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 acknowledge that further experiments on larger-scale agent systems are needed for generalisation of our results and are in fact an important part of our future work. However, since most of BDI-style agent programs consist of plans, goals and beliefs, we would expect that our results would generalise to other agent-oriented programming languages.

VI. Related work

Existing techniques for change impact analysis tend to fall into two groups: static and dynamic analysis. Static impact analysis techniques (e.g. [9] for object-oriented systems or a number of techniques reported in [8]) usually perform either program slicing or graph traversals to compute impact sets in terms of collecting data related to potential impacts. Our work falls in this category. 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 [13] 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 framework can be extended in this direction to variably cover a range of granularity levels (e.g. agents or plans or literals) in agent systems.

In contrast to static analysis, dynamic analysis techniques (e.g. [10, 11]) compute impacts using data obtained from executing a program. 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. Exploring dynamic impact analysis for agent systems is part of our future work.

There has been a range of work on slicing logic programs. In particular, Zhao et. al. [14] proposed a Literal Dependency Net (LDN) representing different dependencies in concurrent logic programs that are in the form of Guarded Horn Clauses (GHC) [20]. 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. [15] adopted the LDN proposed in [14] 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 work is also inspired by notions and ideas proposed by Zhao et. al. [14]. 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 [15] 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.

VII. Conclusions and future work

In this paper, we have presented a framework to support change impact analysis during the maintenance and evolution of agent systems, specifically for those based on the BDI architecture and written in AgentSpeak. The core part of our framework is a taxonomy of dependencies that capture various dependent relationships within an agent system. These include 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 framework has been implemented in AgentCIA, a plugin for the Jason IDE, a well-known platform for agent development. We have also performed several experiments on real agent systems and the results have indicated the effectiveness of our approach given a reasonable number of primary changes performed.

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 [19] 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. Furthermore, we would like to investigate dynamic impact analysis for agent systems (i.e. observing the behaviour of the agent system at run time) and compare it with the static analysis technique proposed in this paper. Finally, to allow for generalization of our results, large-scale experimental evaluation is necessary and is also an important topic of our future work.

Acknowledgements

We would like to thank Michael Winikoff for his comments on this work and Vinh San To for helping implement the AgentCIA plugin for Jason.

References

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

[2] M. Pěchouček and V. Mařík, "Industrial deployment of multi-agent technologies: review and selected case studies," *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, vol. 17, pp. 397–431, 2008.

[3] S. Munroe, T. Miller, R. A. Belecheanu, M. Pěchouček, P. McBurney, and M. Luck, "Crossing the agent technology chasm: Lessons, experiences and challenges in commercial applications of agents," *Knowledge Engineering Review*, vol. 21, no. 4, pp. 345–392, 2006.

[4] S. S. Benfield, J. Hendrickson, and 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*. New York, NY, USA: ACM, 2006, pp. 10–15.

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

[6] H. K. Dam and M. Winikoff, "An agent-oriented approach to change propagation in software maintenance," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. ?, no. ?, p. ?, To appear in print; published online 1/1/2011.

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

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

[9] L. Li and 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.

[10] T. Apiwattanapong, A. Orso, and 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*. New York, NY, USA: ACM, 2005, pp. 432–441.

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

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

[13] M. Petrenko and 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.

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

[15] R. H. Bordini, M. Fisher, M. Wooldridge, and W. Visser, "Property-based slicing for agent verification," *Journal of Logic and Computation*, vol. 19, pp. 1385–1425, December 2009.

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

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

[18] M. C. O. Maia, R. A. Bittencourt, J. C. A. de Figueiredo, and 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.

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

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