Towards semantic merging of versions of BDI agent systems

Yingzhi Gou, Hoa Khanh Dam and Aditya Ghose

School of Computer Science and Software Engineering
University of Wollongong
New South Wales 2522, Australia
{yg452,hoa,aditya}@uow.edu.au

Abstract. Modern software development environment is based on developers' ability to work in parallel on the same codebase and perform concurrent changes, which potentially need to be merged back together. However, state-of-the-art merging systems follow text-based algorithms that focus only on modifications to text but completely ignore the semantic of the code written. This limitation significantly restricts developers' ability to perform and merge concurrent changes. In this paper, we propose a merging technique that fully understands the programming language structure of typical BDI agent systems. In addition, our approach effectively captures the semantic of an agent system using the notion of semantic effects of goals, plans and actions constituting the agent system.

1 Introduction

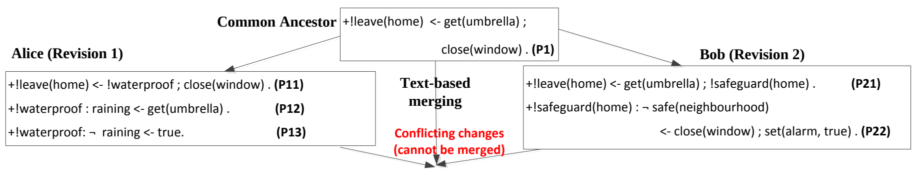
Engineering large, complex software systems is inherently a collaborative process since it requires the participation of teams of people who may work on the same product independently and concurrently (creating different versions of it). As a result, merging is a critical functionality in existing versioning systems which support the optimistic versioning process that enables different developers to work concurrently on the same set of software artefacts (e.g. source code) rather than pessimistically locking each artefact when it is changed by one developer. However, software merging remains a highly challenging and complicated process since merging should heavily depend on the syntax and semantic of the software artefacts [5]. State-of-the-art versioning systems (e.g. CVS, Subversion or Git) are usually based on textual merging techniques. Since any software program (including agent programs) can be seen as a piece of text, text-based merge tools have been dominantly used for merging software code. This flexibility however comes with a cost in which text-based merge tools do not take the specific syntax, structure and semantic of agent programs into account and thus the merging may often result in unnecessary conflicts or a merged version which has syntax errors and inconsistent semantic behavior.

Since the 1980s, intelligent agent technology has attracted an increasing amount of interest from the research and business communities, and the practical utility of agents has been demonstrated in a wide range of domains such as weather alerting, business process management, holonic manufacturing, e-commerce, and information management. This number continues to increase since there are compelling reasons to use intelligent agent technology. However, to the best of our knowledge, there has been no work on merging versions of an agent program. If we are to be successful in the development of large-scale agent systems which requires the participation of teams of people who may work on the same product independently and concurrently, the research community must provide solutions and insights that will improve the practice of merging versions of agent software.

The main purpose of this paper is to contribute towards filling that gap. We propose a merging technique specifically for Belief-Desire-Intention (BDI) agent systems. Our approach captures the essential semantic of a BDI agent system by computing the *cumulative effects* of plan execution from the *immediate effects* of the actions constituting the plan. We then merge the semantic effects of the revisions, and use them to establish a merged version. In the remaining of the paper, we will describe an example to illustrate the limitations of existing text-based merging approach, and present in detail our approach and how it overcomes those issues.

2 Illustrative example

Most of today's version controlling systems uses text-based merging techniques which consider software programs (regardless of the programming language which they are written in) merely as text files. The most common approach is to use line-based merging where lines of text are considered as indivisible units [5]. Line-based merging however cannot handle two concurrent modifications to the same line very well, which will be shown in the following scenario.



```
+!leave(home)  <- get(umbrella) ;
                   close(window) . (P1)
```

```
+!leave(home) <- !waterproof ; close(window) . (P11)
+!waterproof : raining <- get(umbrella) .      (P12)
+!waterproof: ¬  raining <- true.              (P13)
```

```
+!leave(home) <- get(umbrella) ; !safeguard(home) .      (P21)
+!safeguard(home) : ¬ safe(neighbourhood)
                 <- close(window) ; set(alarm, true) . (P22)
```

Fig. 1. An example of classical, text-based merging (unnecessary conflicts)

Figure 1 illustrates an example of two developers, Alice and Bob, concurrently work on the same agent program written in AgentSpeak(L) [6], a well-known, abstract BDI agent programming language. BDI agents' behaviour is mostly determined in terms of their plans to handle events or achieve goals. Each plan is typically of the form $G : [C] \leftarrow P$, meaning that the plan is an applicable plan for achieving goal G when context condition C is believed true. Plan P typically contains a sequence of actions that are meant to be directly executed in the world (e.g. $get(umbrella)$ in plan P1 in Figure 1) or subgoals (written as $!G$) (e.g. $!waterproof$ in plan P11) to be resolved by further plans. Both Alice and Bob check out the same piece of code (the common ancestor), which in this example, a plan (P1) to achieve goal $leave(home)$, and make different changes to it. Alice replaces the first action of the plan with a subgoal *waterproof* and creates two plans (P12 and P13) to resolve the subgoal. In the meanwhile, being unaware of Alice's changes, Bob replaces action $close(window)$ with a subgoal $safeguard(home)$ and creates a plan (P22) to resolve it. When both developers check in their own revision, existing versioning systems (which mostly rely on text-based merging) would detect unnecessary conflicts since parallel modifications has made to the same lines of code in the common ancestor. In general, text-based merging fails in these scenarios since they are heavily dependent on the position of the texts being modified and they do not consider any syntactic or semantic information in the agent code.

3 Semantic effects

A BDI agent program is built around a plan library, a collection of pre-defined hierarchical plans indexed by goals. As a result, the semantic of a BDI agent program is mostly determined by the semantic of its plans, which can essentially be captured by the semantic effects achieved by the plans. Such effects can be expressed in terms of *declarative goals* that the plans are meant to achieve. However, mainly due to practical concerns, goals in BDI agent programming languages are mostly *procedural* where a goal is a set of tasks or processes that are to be completely carried out [7]. Therefore, a description of effects achieved by a plan has to be explicitly established from the effects of its constituting steps in a *context-sensitive* manner. We note that such a description will necessarily be non-deterministic, i.e., there might be alternative effects achieved, which is due to the following reasons. First, there might be different paths in plan execution since there might be multiple ways of achieving a (sub-)goal. Second, the effects of certain plan steps might "undo" the effects of prior process steps. This is often described as the belief update or knowledge update problem – multiple alternative means of resolving the inconsistencies generated by the "undoing" of effects is another source of non-determinism.

Each action has a precondition under which an action can be successfully executed and its effect (or postcondition) on the environment. The semantic effect of action a, denoted as $e(a)$, is a *conjunctive* set of belief literals since the action's effect on the environment may eventually be perceived by the agent. The effect of action $get(umbrella)$ in the example in the previous section is the set $\{on(umbrella, hand)\}$, and the effect of action $set(alarm, true)$ is $\{alarm(on)\}$. Many agent programming languages (e.g. 3APL [4]) require an explicit specification of actions in terms of both preconditions and effects. However, our work *only* focuses on leveraging action effects to establish semantical representations of agent systems and use them for merging. A BDI agent also has a belief base $\mathcal{B}$ which encodes what the agent believes about the world. An agent's belief base may also contain rules, which allows for new knowledge to be deduced from existing knowledge. For simplicity, in this paper we assume that the context condition is expressed as a conjunctive set of belief literals, which is similar to a semantic effect.

The effect specification of actions allows us to determine, at design time, the (cumulative) effects of plan execution. We now define a number of basic definitions that are used in the procedure for computing the cumulative effects.

Definition 1. *For two effects e_1 and e_2, and the belief base $\mathcal{B}$, if $e_1 \not\models \perp$ and $e_2 \not\models \perp$, then the cumulative effects $acc(e_1, e_2)$ (accumulating es_2 onto es_1) is defined as:*

$$\begin{aligned} acc(e_1, e_2) = \{ & e_2 \cup e \mid e \subseteq e_1 \wedge e \cup e_2 \cup \mathcal{B} \not\models \perp \wedge \\ & \textit{if there exists } e' \textit{such that } e \subset e' \subseteq e_1, \\ & \textit{then } e' \cup e_2 \cup \mathcal{B} \models \perp \} \end{aligned}$$

We note that the result of $acc()$ on a pair of effects is a disjunctive set of effects, each of which represents a distinct way in which potential inconsistencies between the effects to be accumulated are resolved. For example, if $e_1 = \{m, n\}$ and $e_2 = \{x, y\}$ and there is a rule $m \wedge n \rightarrow \neg\, y$ in the belief base $\mathcal{B}$, then $acc(e_1, e_2) = \{\{m, x, y\}, \{n, x, y\}\}$.

Definition 2. *The cumulative effects of the two disjunctive sets of effects ES_1 and ES_2 are defined as:*

$$ES_1 \oplus ES_2 = \bigcup_{es_i \in ES_1, es_j \in ES_2} acc(es_i, es_j)$$

The operator $\oplus$ performs the pair-wise effect accumulation $acc()$ on every pair of $(es_i, es_j) \in ES_1 \times ES_2$ to form a new disjunctive set of effects. For example, if $ES_1 = \{e_1\}$ and $ES_2 = \{e_2\}$, then $ES_1 \oplus ES_2 = \{\{m, x, y\}, \{n, x, y\}\}$. The cumulative effects of a plan are represented as a disjunctive set of effects where each effect (also called an effect scenario) corresponds to a particular path of the plan execution (i.e. a particular scenario). In order to compute a plan's cumulative effects, we need to simulate the plan execution, particularly the context-sensitive subgoal expansion and plan selection. For example, assume that a plan has executed a number of actions, which gives cumulative effects $ES = \{\{m, x, y\}, \{n, x, y\}\}$, and is about to expand a subgoal g, which can be resolved by plan P under the context condition $c = \{\neg\, n\}$. Since only the effect scenario $\{m, x, y\}$ is consistent with the context condition, the cumulative effects just before plan P is executed would be $ES \ominus c = \{\{m, x, y, \neg\, n\}\}$. The operator $\ominus$ which eliminates effects that are inconsistent with a plan's context condition is defined as below.

Definition 3. *The effect elimination of ES with regard to context condition c (which is a set of literals and can be considered as an effect) is defined as:*

$$ES \ominus c = \bigcup_{es_i \in ES} \{es_i \cup c\}, \textit{where } es_i \cup c \not\models \perp$$

Algorithm 1 Computing semantic effects for a given node in the goal-plan tree (initial call is $\mathit{SemanticEffect}(\mathit{root}, \varnothing)$)

```
procedure SemanticEffect($n$, $ES$)
    if $n$ is a plan node then
        $ES \leftarrow ES \ominus \mathit{context}(n)$
        if $ES \neq \varnothing$ then
            for each child $n'$ of $n$ from left to right do
                $ES \leftarrow \mathit{SemanticEffect}(n', ES)$
            end for
        end if
    else if $n$ is an action node then
        $ES \leftarrow ES \oplus \{e(n)\}$
    else if $n$ is a goal node then
        $ES' \leftarrow \varnothing$
        for each child $n'$ of $n$ do
            $ES' \leftarrow ES' \cup \mathit{SemanticEffect}(n', ES)$
        end for
        $ES \leftarrow ES'$
    end if
    return $ES$
end procedure
```

A BDI agent program can be represented as a number of goal-plan trees where each goal has as children the plans that are relevant to it, and each plan has as children its actions and/or subgoals. The goal-plan tree is an "and-or" tree: each goal is realised by one of its relevant plans ("or") and each plan needs all of its actions to be executed and its sub-goals to be achieved ("and"). Therefore, computing semantic effects for a BDI agent program is reduced to computing semantic effect for a goal-plan tree. Algorithm 1 describes how we traverse a goal-plan tree to compute the cumulative effects for a particular node in the tree. The cumulative effects are stored in the set ES which are accumulated as we visit each node of the tree in the depth-first search manner. If the node n is a plan node (lines 2–8), we obtain the context condition of the plan (i.e. $\mathit{context}(n)$) and apply the effect elimination operator $\ominus$ onto the set of cumulative effects ES. If the outcome is *not* an empty set, reflecting there exists at least a scenario in which the plan is applicable, we visit each node of the plan's children (i.e. which is either an action or subgoal node) to accumulate its semantic effects. If the node is an action node (lines 9–10), we simply use the operator $\oplus$ to accumulate the action's effect. Finally, if the node is a goal node (lines 11–16), we visit each node of the goal's children (i.e. which are plan nodes), compute its semantic effects and add them into the set of cumulative effects.

Figure 2 shows the goal-plan trees for the agent program and its two revisions in Figure 1. Using the procedure described in Algorithm 1, we can compute that the cumulative effects of goal $+!\mathit{leave}(\mathit{home})$ in the ancestor version are $ES_{base} = \{\{\mathit{on}(\mathit{umberalla}, \mathit{hand}), \mathit{window}(\mathit{closed})\}\}$, and in revisions 1 and 2 are $ES_1 =$

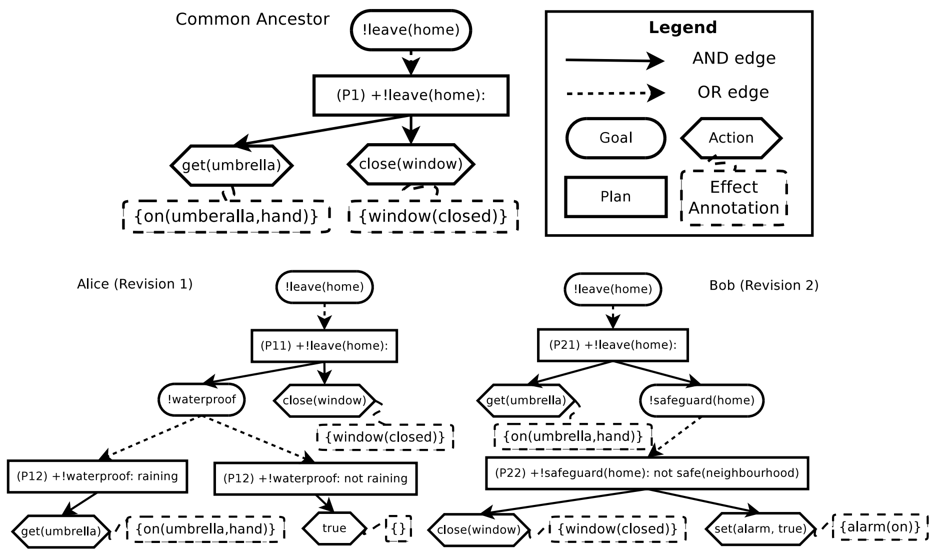


Fig. 2. Goal plan trees annotated with semantic effects for programs in Figure 1

$\{\{raining, on(umberalla, hand), window(closed)\}, \{\neg raining, window(closed)\}\}$ and $ES_2 = \{\{on(umberalla, hand), \neg safe(neighbourhood), window(closed), alarm(on)\}\}$.

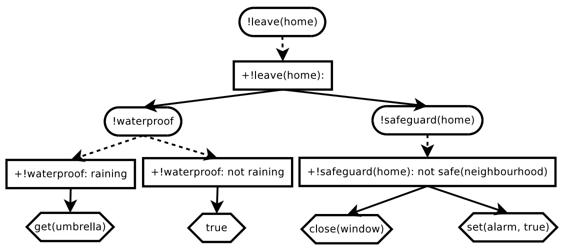
4 Semantic merging

We now outline the process of merging two BDI agent programs using semantic effects. Our approach follows the popular three-way merging [5] which requires a common ancestor program as the base and two revisions.

1. Compute the semantic effects of the (common) ancestor agent program and its two revisions, which gives us three set of semantic effects ES_{base}, ES_1 and ES_2.
2. Use those semantic effects to identify the essential differences between the ancestor and the revisions. The difference of two semantic effects ES_i and ES_j is defined as $\delta(ES_i, ES_j) = ES_j \backslash ES_i$, which contains the semantic effects that are in ES_j but not in ES_i. Note that δ is asymmetric, that is $\delta(ES_i, ES_j) \neq \delta(ES_j, ES_i)$.
3. Compute the merged semantic effects $ES_{merge} = (ES_{base} \cup \Delta^+) \backslash \Delta^-$ where $\Delta^+ = \delta(ES_{base}, ES_1) \cup \delta(ES_{base}, ES_2)$ and $\Delta^- = \delta(ES_1, ES_{base}) \cup \delta(ES_2, ES_{base})$. Intuitively, Δ^+ is the effects newly created in the revisions, and Δ^- is the effects that are removed in the revisions. The merged version therefore has the behaviours in the base program that are preserved in both revisions and the new behaviours coded in the revisions. Note that the order of merging is not important since merging is done here in terms of set union.
4. Construct the merged program from the merged semantic effects.

We now illustrate how a merged version can be obtained in our running example by following the above steps. In the previous section, we have computed the semantic effects ES_{base}, ES_1 and ES_2. Now, we compute the semantic differences $\delta(ES_1, ES_{base})$ and $\delta(ES_2, ES_{base})$ which are the same, and equal to $\{\{on(umberalla, hand), window(closed)\}\}$. The set of merged semantic effects are then computed as follows.

$$\begin{aligned}
ES_{merge} =&(ES_{base} \cup \Delta^{+}) \setminus \Delta^{-} = (ES_1 \cup ES_2) \setminus \Delta^{-} \\
=&\{\{raining, on(umberalla, hand), window(closed)\}, \\
&\{\neg raining, window(closed)\}, \\
&\{on(umberalla, hand), \neg safe(neighbourhood), window(closed), alarm(on)\}\}
\end{aligned}$$



```
+!leave(home):
  <-   !waterproof;
       !safeguard(home).
+!waterproof: raining
  <-   get(umbrella).
+!waterproof: not raining
  <-   true.
+!safeguard(home): not safe(
     neighbourhood)
  <-   close(window);
       set(alarm, true).
```

Fig. 3. A merged goal-plan tree and program for the example in Figure 1

The final step of the merging process involves reconstructing a program from the merged semantic effects. This involves reconstituting a *feasible* goal-plan tree such that the semantic effects of this tree, denoting ES_r, satisfy the following condition: $\forall\, es \in ES_{merge}, \exists\, es' \in ES_r, es' \cup \mathcal{B} \models es$. Figure 3 shows a feasible goal-plan tree whose semantic effects $ES_r = \{\{raining, on(umberalla, hand),$ $\neg safe(neighbourhood), window(closed), alarm(on)\}, \{\neg raining, \neg safe(neighbourhood),$ $window(closed), alarm(on)\}\}$ satisfy the above condition. If we cannot reconstitute any feasible goal-plan tree from the merged semantic effects, there must be conflicting changes made in the revisions that need to resolved. Our future work involves identifying those conflicting changes using the semantic effects. We also note that there may be more than one feasible goal-plan tree and they should be presented to the software engineers for selection. Future work would involves developing a search algorithm to find all of those feasible goal plan trees. Computing the differences and the merged semantic effects are essentially set operations, which grow linearly with the size of the programs. The number of ways to resolve conflicts is constrained within the changes made in the revisions to be merged. Therefore, we expect the approach does scale to standard programs.

5 Conclusions

Although there have been some recent work on providing support for the maintenance and evolution of agent systems (e.g. [2, 3]), there is still a big gap in addressing the versioning and merging issues of agent systems. Text-based merging is the dominant approach used in most today's versioning systems. Due to its limitations, a number of approaches (e.g. [1] or see [5] for a comprehensive survey) have been proposed to merge classical programs (e.g. procedural or object-oriented) in a semantical manner. Recently, there have been some work (e.g. the recently released commercial SemanticMerge software[1]) on refactoring-aware merge (which preserves the semantics), but they are limited to object-oriented programming languages. Such approaches are not readily applied to a BDI agent program due to its distinct syntax, structure and semantics. In addition, traditional approaches which rely on program slicing or dependency graph do not really capture the true semantics of agent programs. In particular, they cannot capture the semantic effects of agent actions. We have proposed a novel approach that enables merging versions of a BDI agent program semantically. Since the approach is built upon an abstract BDI notation, it can generally be extended to any BDI agent programming languages. Future work involves further refining and implementing our merge approach.

References

1. V. Berzins. Software merge: semantics of combining changes to programs. *ACM Trans. Program. Lang. Syst.*, 16(6):1875–1903, Nov. 1994.
2. H. K. Dam and A. Ghose. Automated change impact analysis for agent systems. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 33–42, Washington, DC, USA, 2011. IEEE.
3. K. H. Dam and M. Winikoff. Cost-based BDI plan selection for change propagation. In Padgham, Parkes, Müller, and Parsons, editors, *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 217–224, Estoril, Portugal, May 2008.
4. M. Dastani, M. Birna Riemsdijk, and J.-J. Meyer. Programming multi-agent systems in 3APL. In R. Bordini, M. Dastani, J. Dix, and A. Fallah Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 39–67. Springer US, 2005.
5. T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.
6. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. Perrame, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, pages 42–55. Springer Verlag, 1996. LNAI, Volume 1038.
7. S. Sardina and L. Padgham. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70, July 2011.

[1] `http://www.semanticmerge.com/`