



# A search-based approach for detecting circular dependency bad smell in goal-oriented models

Mawal A. Mohammed<sup>1</sup> · Mohammad Alshayeb<sup>1</sup> · Jameleddine Hassine<sup>1</sup>

Received: 16 October 2020 / Revised: 13 November 2021 / Accepted: 2 December 2021  
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2022

## Abstract

Goal-oriented models are gaining significant attention from researchers and practitioners in various domains, especially in software requirements engineering. Similar to other software engineering models, goal models are subject to bad practices (i.e., bad smells). Detecting and rectifying these bad smells would improve the quality of these models. In this paper, we formally define the circular dependency bad smell and then develop an approach based on the simulated annealing (SA) search-based algorithm to detect its instances. Furthermore, we propose two mechanisms (namely, pruning and pairing) to improve the effectiveness of the proposed approach. We empirically evaluate three algorithm combinations, i.e., (1) the base SA search algorithm, (2) the base SA search algorithm augmented with pruning mechanism, and (3) the base SA search algorithm augmented with pruning and pairing mechanisms, using several case studies. Results show that simulated annealing augmented with pruning and pairing is the most effective approach, while the simulated annealing augmented with pruning mechanism is more effective than the base SA search algorithm. We also found that the proposed pruning and pairing mechanisms provide a significant improvement in the detection of circular dependency bad smell, in terms of computation time and accuracy.

**Keywords** Circular dependency · Model-driven engineering · Requirements · Simulated annealing · GRL

## 1 Introduction

Goal-oriented requirements engineering (GORE) is concerned with assisting stakeholders to explore, elaborate, structure, analyze, negotiate, and document their requirements using goals [1]. Goal modeling has become an effective approach for capturing and describing various stakeholders' intentions and business goals and how these can be refined in terms of functional and non-functional requirements. Goals can be stated at different levels of abstraction ranging

from high-level coarse-grained strategic mission statements to lower-level fine-grained operational responsibilities. In recent years, the popularity of goal-oriented approaches has increased, leading to the development of many goal-oriented modeling languages and frameworks [2]. These frameworks can be divided into two categories: (1) goal-oriented modeling frameworks such as Non-Functional Requirements (NFR) [3, 4] and Keep All Objects Satisfied (KOAS) [5, 6], and (2) agent-oriented goal modeling frameworks such as  $i^*$  [7] and Goal-oriented Requirement Language (GRL), part of ITU-T's User Requirements Notation (URN) standard [8]. These frameworks share many common concepts, such as goal refinement and operationalization, and offer many features, such as evaluation of goal satisfaction levels, the exploration of available alternatives, etc. In addition to requirements engineering, goal-based approaches have been successfully applied to, among others, business intelligence, adaptation and variability, compliance policies, privacy, security, and trust modeling [9].

Our focus in this paper is on models developed using GRL [10], which provides several features that can be used to assist requirements elicitation, documentation, and analysis.

---

Communicated by Silvia Abrahao.

✉ Mohammad Alshayeb  
alshayeb@kfupm.edu.sa

Mawal A. Mohammed  
g201102570@kfupm.edu.sa

Jameleddine Hassine  
jhassine@kfupm.edu.sa

<sup>1</sup> Interdisciplinary Research Center for Intelligent Secure Systems, Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia

One of the most important and useful features of GRL is the qualitative/quantitative evaluation of satisfaction levels of the GRL actors and intentional elements (e.g., softgoals, goals, tasks, and resources). Satisfaction analysis starts by setting initial evaluation values (qualitative or quantitative) for some intentional elements (usually low-level tasks, representing the considered solutions). These initial evaluation values, forming a GRL strategy, are propagated to the other intentional elements of the model through the various links that connect them, reaching the high-level intentional elements (representing the actual stakeholders' goals). The outcome of the satisfaction analysis would allow for selecting the best strategy to satisfy stakeholders' goals. Hence, the requirements engineer can make an insightful decision on what requirements to be considered, what scenarios to be explored, and which set of features to be included in the system to be developed [11, 12]. Satisfaction analysis and its underlying propagation algorithms are sensitive to dependency cycles which can influence satisfaction analysis negatively through disturbing the propagation algorithms [8, 11]. These algorithms are implemented in the open-source jUCMNav tool [12], the most comprehensive GRL tool available to date.

We formulated the circular dependency as a bad smell. By definition, bad smells are not errors. They are indicators of bad quality [13]. Bad smells are found to have negative impacts on maintainability [14, 15], and software quality in general [16, 17], and they are prevalent in code [13] and software models [18]. The presence of circular dependencies in GRL models does not mean that the modeled requirements are wrong. GRL models are susceptible, by design, to cycles that we refer to as circular dependencies (In Sect. 0, we formalize the notion of circular dependency and formulate it as a bad smell), where the propagation path of evaluation values in the context of satisfaction analysis forms a cycle. These cycles do not invalidate the model when the model is used as a communication medium to document or convey requirements. However, such cycles may present several problems in the later stages of the requirements engineering process. Indeed, a GRL model that contains dependency cycles may present one or many of the following issues:

- The presence of dependency cycles hinders the application of satisfaction analysis by disturbing the propagation algorithms [8, 11, 19]. This would make the evaluation of available strategies, part of trade-off analysis, fluctuating. To conduct satisfaction analysis, the analyst defines a GRL strategy, which gives initial satisfaction values to some GRL intentional elements (usually to leaf elements). These values are then propagated to the rest of the model (a.k.a. forward propagation). As stated in the URN standard [8], a cycle is not evaluated unless one of its elements is manually overridden. Hence, manually setting a satisfaction value is similar to breaking the cycle because manual evaluations

cannot be overridden by the propagation algorithm. If none of the elements forming a cycle has a user-assigned initial satisfaction value, the propagation algorithm will not converge and will terminate without assigning satisfaction values to the elements forming the cycle, as well as all elements depending (directly or indirectly) on them. Therefore, automatic detection of these cycles facilitates resolving them. In the motivational example presented in Sect. 0, we provide a detailed illustration of this issue.

- Goal models are rarely used in isolation [20]. They are usually complemented with other models, e.g., workflow and feature models that impose additional constraints on goal model elements, more specifically on tasks (describing goals' operationalization). For example, GRL models are usually complemented with a set of scenario models expressed using the Use Case Maps (UCM) language, part of the ITU-T URN language [8]. The results of the propagation algorithm might influence the corresponding UCM scenario variables, and the results of the scenario traversal algorithm might influence the evaluation of intentional elements in the corresponding GRL model [11]. Moreover, when GRL circular dependencies exist between tasks, these circular dependencies will most likely appear in feature models [20], if any, creating a potential circular dependency between the corresponding features.
- Reuse of software artifacts at any level of abstraction is a highly recommended practice because of its potential benefits of increased productivity, quality, and reduced cost and time [21]. Goal models may be reused in their entirety [22], therefore, the existing cycles persist in future uses of these models unless resolved. Different from other software artifacts that are used in parts. In these artifacts, cycles can be broken unintentionally as a result of reusing parts of the artifact (e.g., code).

Given the potential concerns that may be caused by the circular dependencies, the main objective of this paper is to define and detect circular dependencies in GRL models. More specifically, we make the following contributions:

- Introduce and formally define the circular dependency bad smell in the context of the GRL language [10].
- Formulate the GRL-based circular dependency detection as a search-based problem and propose a fully automated simulated annealing search-based detection approach. Manual detection of circular dependencies, even in small models, may not be easy due to the use of different types of links and due to the use of intentional element' references. Hence, some cycles may not be easily visualized. The problem becomes even harder when goal models grow in size and complexity.
- To improve the efficiency of the proposed detection approach, we augmented the base simulating annealing

search algorithm with two mechanisms, namely pruning and pairing.

- Conduct an empirical study to show the applicability of three proposed variants of the adopted algorithm and assess their efficiency using several GRL case studies.
- Develop a set of guidelines that would help practitioners deploy the proposed approach more efficiently in order to detect all circular dependencies in a given GRL model.

The rest of this paper is organized as follows. The state of the art is described in Sect. 2. A motivational example of a meeting scheduler is presented in Sect. 3. Section 4 presents the problem definition where the circular dependency bad smell is defined and formulated as a search-based problem. In Sect. 5, we describe the proposed simulated annealing search-based detection approaches. The experimental evaluation is presented in Sect. 6. The results are provided in Sect. 7. In Sect. 8, we present the discussion and potential threats to validity. Finally, conclusions are drawn in Sect. 9.

## 2 State of the art

There is a large body of research on bad smell detection approaches. These approaches focus mainly on code-based and UML-based bad smells [23, 24] and use various techniques such as machine learning techniques [25] and search-based algorithms [26]. In this section, an overview of some of these techniques is presented.

### 2.1 Bad smell detection techniques

In the following subsections, an overview of the major techniques that have been used to detect bad smells in the source code and the software engineering models is discussed. Regardless of the software artifacts (i.e., code, UML model, goal model, etc.), detection of the instances of bad smells depends on the representation of these artifacts and the definition of the addressed bad smell. Different techniques might apply to the different artifacts with the right adaptations depending on the addressed bad smell. A comprehensive list of the detection techniques used in the literature can be found in [27].

### 2.2 Metric-based bad smell detection

Metrics were employed to detect bad smells by establishing an association between the subject bad smell and one or more metrics. These metrics include size, coupling, cohesion, inheritance, etc. After establishing the association, a threshold is used to mark the existence of the associated bad smell. Several studies employed metrics to detect different bad smells in different ways for different artifacts. Some of

these studies employed existing metrics, others created new metrics, and the rest used a combination of both. Bertran [28] proposed the use of metrics to detect 8 code smells related to the software architecture using direct association rules that are composed of combinations of 7 metrics. Dexun et al. [29] proposed the use of distance metrics to detect design and code bad smells. These metrics are used to quantify multiple invocation relationships between every two entities in the subject artifact. They applied their technique to detect the feature envy bad smell. Nongpong [30] proposed a new metric based on internal and external calls to detect the feature envy bad smell as well. Based on those calls, a call set is defined. This set is the basis for creating a feature envy factor which is a quantitative value between 0 and 1 that can be used to detect feature envy instances based on predefined thresholds. Fourati et al. [31] introduced an approach for detecting bad smells at the design level. They addressed 5 bad smells. New and existing class and sequence diagram metrics are used in their approach to characterize the addressed bad smells. Singh and Kahlon [32] developed two models to predict smelly classes based on metrics. The first model is binary, and the second is a categorical model. These models employed several existing metrics and two new metrics. The binary model is deemed more useful compared to the categorical model. Tahvildari and Kontogiannis [33] developed a metric-based framework for identifying improvement opportunities. They addressed structural and architectural design flaws. In their diagnosis for these flaws, two heuristics are used: key classes and single concept class. Chen et al. [34] proposed a metric-based approach to detect ten bad smells in Python code to explore their impact on the maintainability of software projects. In their work, three different approaches are used to specify metric thresholds: experience-based, statistical-based, and tuning-based. They evaluated their work on 106 python software projects. Velioglu and Selçuk [35] proposed an approach to detect smells using a combination of metrics. Their detection method used ten metrics and was applied to detect two bad smells. Thresholds used in their work were determined by a training set.

#### 2.2.1 Rule-based bad smell detection

Rules are also employed to detect bad smells. A rule in the area of bad smell detection is a descriptive query mechanism that describes a symptom (i.e., bad smell). In this approach, rules are used to identify bad smells by employing either direct associations or through a proxy such as metrics. Rules are then applied to the subject model to retrieve the corresponding smells. Czibula et al. [36] proposed a rule-based approach for detecting defective entities in software designs. Their approach is based on the mining of relational association rules in software designs. These rules describe a numerical ordering between attributes that repeatedly appear

in some datasets. Association rules mining is also exploited, but at the code level [37]. The authors used association rules mining with regression analysis to predict co-changes in the code associated with the divergent changes and shotgun surgery bad smells. Kessentini and Sahraoui [38] proposed the use of a music metaphor [39] as a harmonizing mechanism for the automatic generation of detection rules. These rules are generated based on examples and metrics from a training set to detect three bad smells. Maddeh and Ayouni [40] proposed the use of gradual rules to detect smells at the design level. These rules are mined with the GRITE algorithm [41] based on associations to 32 metrics. Palomba et al. [42] proposed a rule-based system for detecting Android-specific bad smells. Their approach is based on traversing the abstract syntax tree of the subject code to find instances of the addressed bad smells based on a rule-based description. Based on that, they developed a tool called aDoctor to detect 15 bad smells of the smells introduced in [42].

### 2.2.2 Machine learning-based bad smell detection

Several machine learning techniques have been applied to detect bad smells, mainly as a classification problem. In these techniques, a classification model is built by identifying features that can differentiate infected from non-infected artifacts with instances of the addressed bad smells. The classification model is trained with training examples to learn how to identify infected artifacts. Most studies in this area follow a similar procedure to detect bad smells. Fontana et al. [43] applied several machine learning techniques to detect bad smells. They employed 16 different machine learning techniques to detect five bad smells in 74 software projects. In similar efforts, Hozano et al. [44] conducted a study to evaluate the performance of six machine learning algorithms to detect four bad smells. They only used a single project in evaluating their approach. Maneerat and Muenchaisri [45] proposed the use of machine learning techniques and metrics to detect bad smells. They used seven machine learning algorithms to predict seven bad smells. The classification model is built by establishing an association between 27 metrics and the addressed smells. Maiga et al. [46, 47] proposed the use of a support vector machine classifier to detect bad smells. They evaluated their technique in detecting four bad smells in three software projects. Hassaine et al. [48] proposed a new machine learning technique inspired by the human body immune system to detect bad smells in software projects. They used that technique to detect three bad smells in two software projects.

### 2.2.3 Search-based bad smell detection

Recently, combinatorial optimization search algorithms have been getting significant attention in software engineering

research and application. Several combinatorial optimization search algorithms have been proposed in the literature [49]. These algorithms differ in the way they approach the optimal solution. However, these techniques share several common concepts. In order to use these algorithms, the problem should be formulated as a search problem. There also should be a way to tell if the optimal solution is reached or not [49]. In the area of bad smell detection, the solution structure is a direct description of the addressed bad smell. These search algorithms can be used solely or together. Kessentini et al. [50] developed a parallel cooperative approach to detect code bad smells. They used a genetic algorithm and genetic programming in parallel. Genetic programming is used to generate detection rules, and the genetic algorithm is used to generate detection examples. The same approach with the appropriate adaptations is used to detect bad smells in web service architectures [51]. The cooperation paradigm was not the only paradigm to use different techniques together. A competitive approach rather than the cooperative approach is used by Boussaa et al. [52]. In this approach, two competitive populations co-evolve based on the genetic algorithm. The first population represents the rules used to detect instances of bad smells, and the second population represents examples used to learn the developed rules. A single algorithm can be used as well. Ghannem et al. [53] proposed the use of a genetic programming algorithm to detect model bad smells. The developed genetic programming algorithm is used to create metric-based rules that characterize the addressed bad smells.

### 2.3 Goal-oriented bad smell detection

In the previous subsections, several studies on bad smell detection were presented. In those studies, several techniques to detect instances of bad smells are presented. However, those studies addressed software artifacts other than goal models. Only a few studies in the literature attempted to introduce and detect bad smells in goal models. Asano et al. [54] introduced four bad smells (called symptoms in their paper) and proposed techniques to detect them. The first bad smell is “low semantic relations between a parent and its children.” This smell addresses the irrelevancy of a child to its parent. If the semantic similarity between a parent and its child is low, it is considered as an instance of this smell. For detecting low semantic relations, the authors employed a lightweight natural language processing technique called case-frames in which goal descriptions are represented as case frames. A case frame consists of a verb and the words that co-occur with it. These words are called concepts, and these concepts are used to locate the different meanings of a verb. The associated concepts with a verb are then mapped into a hierarchical dictionary to calculate the similarities between a goal and each of its children. The calculated similarities are used to identify

low semantic relations bad smell by comparing these similarities to a certain threshold. The second and third bad smells are “too many siblings” and “too few siblings.” These two smells are concerned with whether the number of children is adequate and enough to achieve the parent goal. To detect these smells, the authors used the number of children metric along with upper and lower thresholds. The upper threshold specifies the maximum number of children that can be associated with a parent, after which this number of children is considered as an instance of the too many siblings’ bad smell. The lower threshold specifies the minimum number of children that can be associated with a parent before which this number of children is considered as an instance of the too many siblings bad smell. The fourth bad smell is “course-grained leaf goal.” This smell addresses leaf goals that are not concrete enough. It investigates whether a leaf goal is concrete enough to be operationalized. This smell is detected by measuring the depth of refinement of each branch in the model. If the depth of a branch is insufficient, the associated leaf with this branch is highlighted as an instance of this smell. The major issue with this study is that the thresholds in all these smells are not defined and left to the developer to specify.

Several guidelines have been developed in the literature to improve the development of GRL models [55]. Some of these guidelines are related to the consistency and completeness attributes of GRL models. One of these guidelines, as an example, says that each actor should have at least one element with a nonzero importance value. In other words, if all elements in an actor have an importance value of zero, this actor violates this guideline. The second category of these guidelines is related to the correctness of profiling i\* language using GRL language. An extended version of these rules can be found in the iStarGuide.<sup>1</sup> Some of these guidelines can be taken as guidelines to build GRL models. One of these guidelines, as an example, says that all the dependencies between the actors should have a dependum. GRL standard presents the various configurations of dependencies between actors [56]. Based on the standard, it is optional to have a dependum, but the standard also indicates that having a dependum leads to a more complete dependency relationship. The third category includes guidelines related to unused elements. Violations to these guidelines can be considered as bad smells. For example, one of the guidelines in this category says that the model should not contain empty actors. Empty actors are an undesirable situation and, therefore, it is better to be eradicated. Besides, some other shortcomings come from the nature of the GRL language [57]. These shortcomings are not relevant to modeling practices or methodologies.

<sup>1</sup> <http://istarwiki.org/tiki-index.php?page=iStarGuide>.

## 2.4 Comparison with the related work

This work is different from the other work in the literature in several aspects. First, in this work, a new GRL goal model bad smell (i.e., circular dependency) is introduced and formally defined. Although several code and model bad smells are introduced and defined in the literature [13, 18], to the best of our knowledge, only a few studies have investigated goal models bad smells [54, 55]. Particularly, circular dependencies in goal models have never been investigated in the literature. Secondly, to detect the instances of circular dependency bad smell, we applied the simulated annealing search algorithm. In addition, a careful study of the structural properties of GRL goal models led to the development of two mechanisms, namely pruning and pairing, to improve the search algorithm. As a result, the running time of the developed algorithm is found to be feasible. In the literature, the most efficient algorithm, to find cycles in directed graphs, is Johnson’s algorithm [58]. However, this algorithm grows exponentially with the increase in the number of cycles (more information is provided in Sect. 0). Thirdly, the developed heuristics (i.e., pruning and pairing) have never been considered in the literature. These heuristics with the right adaptations can also be used with the other search techniques such as genetic algorithms because they stem from the nature of these models. Fourthly, there might exist a resemblance between the pruning and forward satisfaction analysis algorithms [59] as both of them move in a forward direction (i.e., leaf to root) and stop on cycles. However, pruning is concerned with the path only, while satisfaction analysis is concerned with the path and values generated by the path. In satisfaction analysis, the degree of contributions (Help, Make, Break, etc.), the type of links (contribution, decomposition, dependency), the nature of satisfaction analysis (quantitative or qualitative) are among the concerns of the satisfaction analysis [59]. Fifthly, we successfully set the parameters of the developed technique methodologically and validated them empirically. For instance, the temperature is set to 1000 and analytically shown to be suitable based on the adopted fitness value configuration. Furthermore, the stopping condition is set and validated based on the developed pruning mechanism to confirm the absence of the instances of dependency cycles in the model and, consequently, specify the needed number of iterations precisely.

## 3 Motivational example: meeting Scheduler

In this section, we introduce the main constructs of GRL using a slightly modified version of a meeting scheduler model introduced by Eric Yu [60] (see Fig. 1). This model is widely used for different purposes in the literature [61–64]. illustrates the basic GRL constructs along with their graphi-

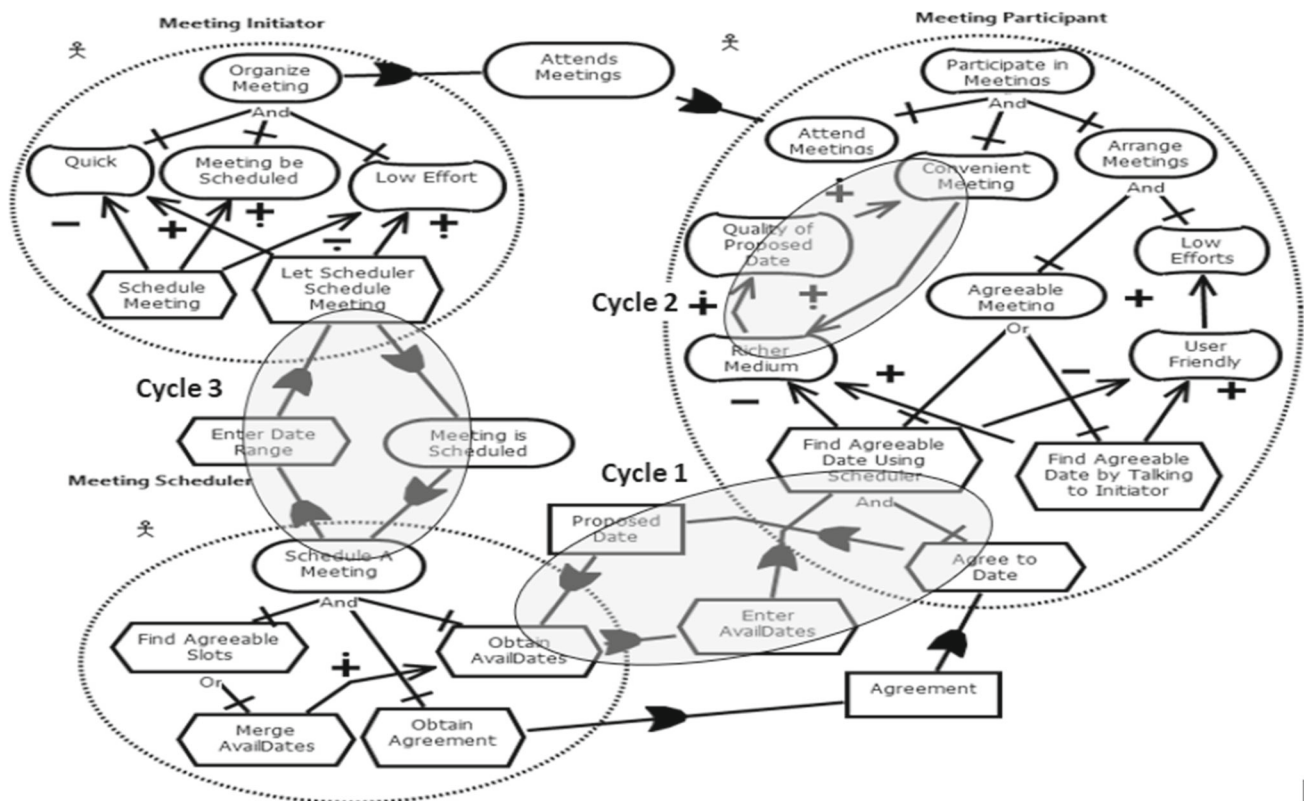









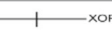

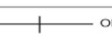













Fig. 1 A modified version of the Meeting scheduler model presented in [60]

cal notations. For a complete description of GRL, readers are referred to the ITU-T standard [10]. Next, we illustrate, using a motivational scenario, how circular dependencies hinder satisfaction analysis of GRL models. We start with a GRL model that contains three cycles. These cycles present several configurations of cycles. Two of these cycles reside between actors, and the third one resides within an actor. In addition, the first cycle consists of two different types of links (i.e., dependency and decomposition), the second cycle consists of contribution links only, and the third cycle consists of dependency links only. Then, we progressively resolve each cycle to see its impact on the satisfaction analysis of GRL models.

Figure 1 (produced using the jUCMNav tool [12]) presents a GRL model that models the domain of a meeting scheduler. Firstly, to model active entities in the domain, actors are used (see Table 1). For example, the meeting scheduler model in Fig. 1 consists of three actors, i.e., “Meeting initiator,” “Meeting participant,” and “Meeting scheduler.” Secondly, to model the objectives of these actors, goals (a.k.a., hard goals) and softgoals are used. The meeting initiator actor, for example, wants to achieve the “Organize Meeting” goal. In addition, the meeting initiator seeks a quick and low effort meeting scheduling service, modeled using the “Quick” and “Low Effort” softgoals. Unlike goals, softgoals do not have

clear-cut criteria with respect to their satisfaction. Thirdly, to model a course of action or a process, the task element is used, and to model the needed resources, the resource element is used. Task and resource elements are used to model mechanisms to achieve goals and softgoals. Fourthly, to model the relationship between elements, contribution and decomposition links are used. For example, “Let Scheduler Schedule Meeting” contributes positively to the achievement of its parent softgoals, i.e., help contribution toward “Low Effort” and somePositive contribution toward “Quick.” On the other hand, the task “Schedule Meeting” contributes negatively to both softgoals, i.e., someNegative contribution toward “Quick” and hurt contribution toward “Low Effort.” Moreover, elements can be decomposed into other elements. For example, the meeting scheduler actor seeks to schedule a meeting, modeled using the “Schedule A Meeting” goal. This goal is refined, using an AND-decomposition link into three tasks. In order to achieve this goal, the meeting scheduler actor is required to obtain the available dates (modeled as “Obtain AvailDates” task), to get the agreement (modeled as “Obtain Agreement” task) and find agreeable slots (modeled as “Find Agreeable slots” task). Fifthly, to model dependencies between actors, dependency links are used. For example, the meeting initiator actor depends on the meeting scheduler actor to schedule a meeting; and the meeting sched-

**Table 1** The basic constructs of the GRL language

Actors and intentional elements		Links	
<b>Actor</b>		Dependency	
<b>Goal</b>		Contribution	
<b>Softgoal</b>		AND-Decomposition	
<b>Task</b>		XOR-Decomposition	
<b>Resource</b>		OR-Decomposition	
Contribution types		Evaluation levels	
<b>Text</b>	<b>Icon</b>	<b>Text</b>	<b>Icon</b>
<b>Make</b>		Satisfied	
<b>SomePositive</b>		Weakly satisfied	
<b>Help</b>		Weakly denied	
<b>Hurt</b>		Denied	
<b>SomeNegative</b>		Conflict	
<b>Break</b>		None	
<b>Unknown</b>	(missing)	Unknown	

uler depends on the meeting initiator to specify a date range. These dependencies are important to show the interactions among actors.

As we mentioned earlier, a circular dependency is an undesirable situation in GRL models. We are going to emphasize this issue further using a motivational scenario to show the impact of the presence of circular dependencies on the satisfaction evaluation of the elements within the cycles and, consequently, on the rest of the model. Before starting to analyze the impact of circular dependencies on satisfaction analysis, it is worth noting that the types of elements or types of links do not affect the way the propagation algorithm handles cycles. Indeed, propagation algorithms treat hard goals, softgoals, and tasks similarly. They compute the satisfaction values of the model elements, and it is the responsibility of the modeler/analyst to interpret the obtained satisfaction levels. For example, in Fig. 5, we can see the calculated values on top of each element. The difference between soft and hard goals is not in the way these values are calculated. The difference will be in their interpretation. The same thing applies to the different types of links in the context of satisfaction analysis. The types of links do not affect the way the propagation algorithm handles cycles. Only the order of evaluation of different link types matters. In GRL, links are evaluated in the following order: decomposition, contribution, and finally, dependency links. For example, the goal “Schedule A Meeting” in the meeting scheduler actor is part of cycle 3 and its evaluation, systematically, depends on several elements that are connected to it using several types of links as shown in Fig. 1. However, regardless of the value that is supposed to, systematically be assigned to this goal, it will not be evaluated because it is a part of a cycle unless the cycle is broken by overriding evaluation values of one of the elements of

cycle 3 or by changing the model’s structure to resolve cycle 3.

The original model introduced in [60] has three cycles. We kept the same number of cycles in the modified version. However, we slightly modified the model by removing and adding some links to avoid shared elements among cycles and to show the different configurations of cycles. In the original model, to show the impact of the cycles on satisfaction analysis, and since all cycles are interconnected through a shared element, all cycles should be resolved at the same time. In such a case, the reader might not be able to see the gradual changes of resolving cycles on satisfaction analysis. Therefore, we modified the model such that breaking one cycle does not lead to breaking the other cycles. However, in the developed tool and technique, we take this case (i.e., shared elements among cycles) into consideration. The developed technique can detect cycles that share elements (see model 3 in Sect. 0). In addition, we also modified the original model by removing one of the original cycles and adding a new cycle (i.e., Cycle 2 in Fig. 1) to show the different configurations of cycles. As a result, the model shown in Fig. 1 has three dependency cycles:

- (1) The first cycle consists of dependency links and one AND-decomposition link. It is located between the “Meeting Scheduler” actor and the “Meeting Participant” actor and is composed of the following sequence: “Agree to Date” “Proposed Date” “Obtain AvailDates” “Enter AvailDates” “Find Agreeable Date Using Scheduler” “Agree to Date.”
- (2) The second cycle consists of contribution links only. It is located within the “Meeting Participant” actor and is composed of the following sequence: “Richer Medi-

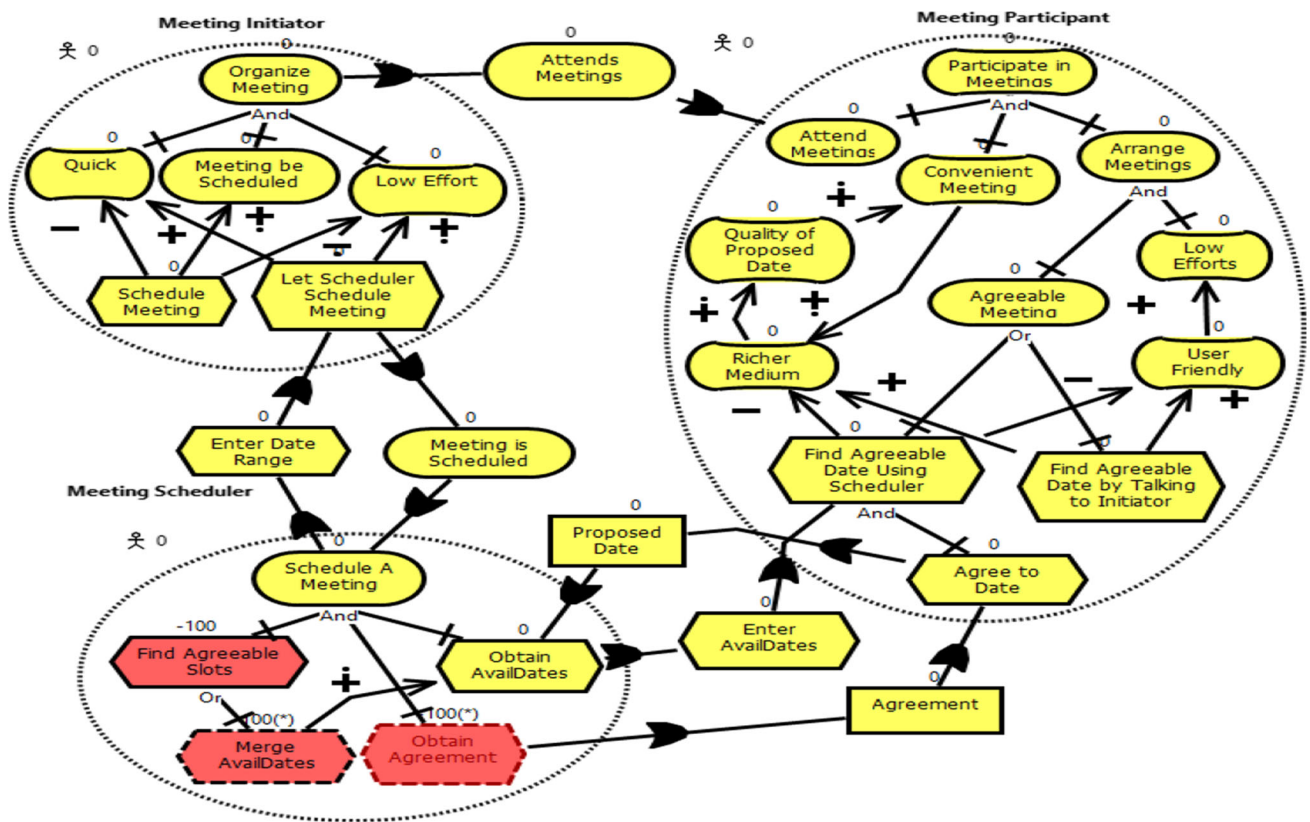


Fig. 2 Applying initial evaluation values to some elements in the model

um” “Quality of Proposed Date” “Convenient Meeting” “Richer Medium.”

- (3) The third cycle consists of dependency links only. It is located between the “Meeting Scheduler” actor and the “Meeting Initiator” actor and is composed of the following sequence: “Schedule A Meeting” “Enter Date Range” “Let Scheduler Schedule Meeting” “Meeting is Scheduled” “Schedule A Meeting.”

The rest of this section is dedicated to showing how the presence of a cycle impacts the evaluation of its elements (i.e., forming the cycle) and, consequently, the rest of the GRL model. To this end, we use our running example of Fig. 1, and we consider an evaluation strategy that fully denies (i.e., satisfaction value of -100 shown in red color in Fig. 2) tasks “Merge AvailDates” and “Obtain Agreement,” part of the actor “Meeting scheduler.” To perform satisfaction analysis, forward propagation algorithms (those suggested in the standard and implemented in the jUCMNav tool) are usually used to propagate (in a bottom-up fashion) the initial evaluation values assigned by the strategy to the rest of the model. For example, in Fig. 2 the “Schedule A Meeting” goal is refined, using an AND-decomposition link, into three tasks: “Obtain AvailDates,” “Obtain Agreement,” “Find Agreeable slots.” In an acyclic model, the “Schedule A Meeting” goal would

have been evaluated to the minimum of the evaluation values of its children (for a complete description of the propagation of the other types of links, please refer to the ITU-T standard [10]). However, due to the presence of cycle 1, the propagation algorithm was not able to propagate the evaluation of the “Merge AvailDates” task to evaluate the satisfaction of the “Obtain AvailDates” task. Furthermore, the evaluation of the satisfaction of the goal “Schedule A Meeting” also failed due to the presence of cycle 3. Therefore, the propagation algorithm terminates without assigning satisfaction values to the elements forming the cycle, as well as to those elements depending (directly or indirectly) on them.

To unblock the propagation, we can break the first cycle, as shown in Fig. 3, by removing the dependency link between “Obtain AvailDates” task and “Enter AvailDates” task. As a result, the evaluation values are now propagated throughout the “Meeting Participant” actor. However, not all elements in the “Meeting Participant” actor were evaluated due to the presence of a second cycle (denoted as Cycle 2 in Fig. 1.) within this actor. Once the second cycle is broken (by removing the contribution link between “Convenient Meeting” softgoal and “Richer Medium” softgoal), the values propagate through the elements of this cycle, as shown in Fig. 4. Furthermore, the evaluation values failed again to propagate to the “Meeting Initiator” actor due to the presence of a third



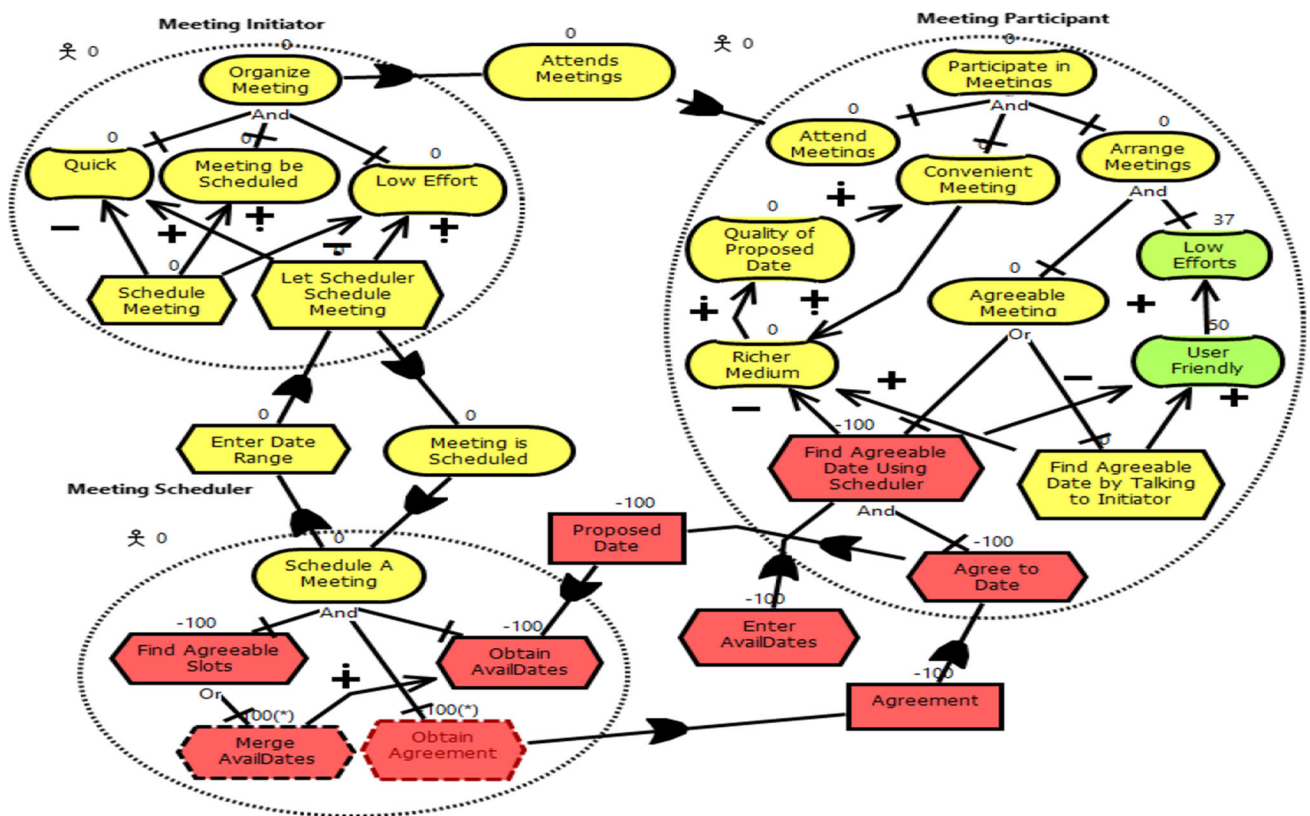


Fig. 3 The propagation of evaluation values when the first cycle is broken

cycle (denoted as Cycle 3 in Fig. 1) between the “Meeting Scheduler” actor and “Meeting Initiator” actor.

The propagation algorithms suggested in the standard do not allow for overriding initial values assigned as part of a strategy, even if they cause conflicts. Therefore, this property can be used to unblock the propagation through cycles. For example, contrary to cycles 1 and 2 (where we have removed a link to break the cycle), we have unblocked the propagation of the satisfaction values in Cycle 3 by overriding the satisfaction value of “Enter Date Range” task to make it fully satisfied (i.e., 100 shown in green color in Fig. 5) and add it to the strategy, as shown in Fig. 5. Although the initial satisfaction value given to the “Enter Date Range” task (i.e., 100), conflicts with the value that should have been assigned by the propagation algorithm (i.e., -100), this assignment was not overridden by the propagation algorithm. If the propagation algorithm allows for overriding initial satisfaction values defined as part of the strategy, this resolution will not be applicable.

#### 4 Problem definition and formulation

As described earlier, in this work, we define a new bad smell, i.e., circular dependency in GRL goal models. Cycles can be

a problematic issue in several settings in computer science [65–68]. They are also an issue that should be taken into consideration in the context of the satisfaction analysis of GRL models as we explained earlier in Sect. 0 & Sect. 0. In this section, we formally define the problem and formulate it as a search problem.

#### 4.1 Circular dependency in GRL models

In the context of satisfaction analysis, a GRL model can be viewed as a network of propagation paths of evaluation values among the elements of the model. In this network, evaluation values propagate from source elements to destination elements creating evaluation dependencies. In these evaluation dependencies, the destination element depends in its evaluation on the source element. The direction of propagation depends on the propagation algorithm. In forward propagation algorithms, evaluation values are propagated in a bottom-up fashion. Hence, evaluation dependencies are created among the elements of the model in a bottom-up fashion as well.

To differentiate evaluation dependencies created by dependency links and evaluation dependencies created by the other types of links, two types of evaluation dependencies can be distinguished, (1) explicit evaluation dependencies

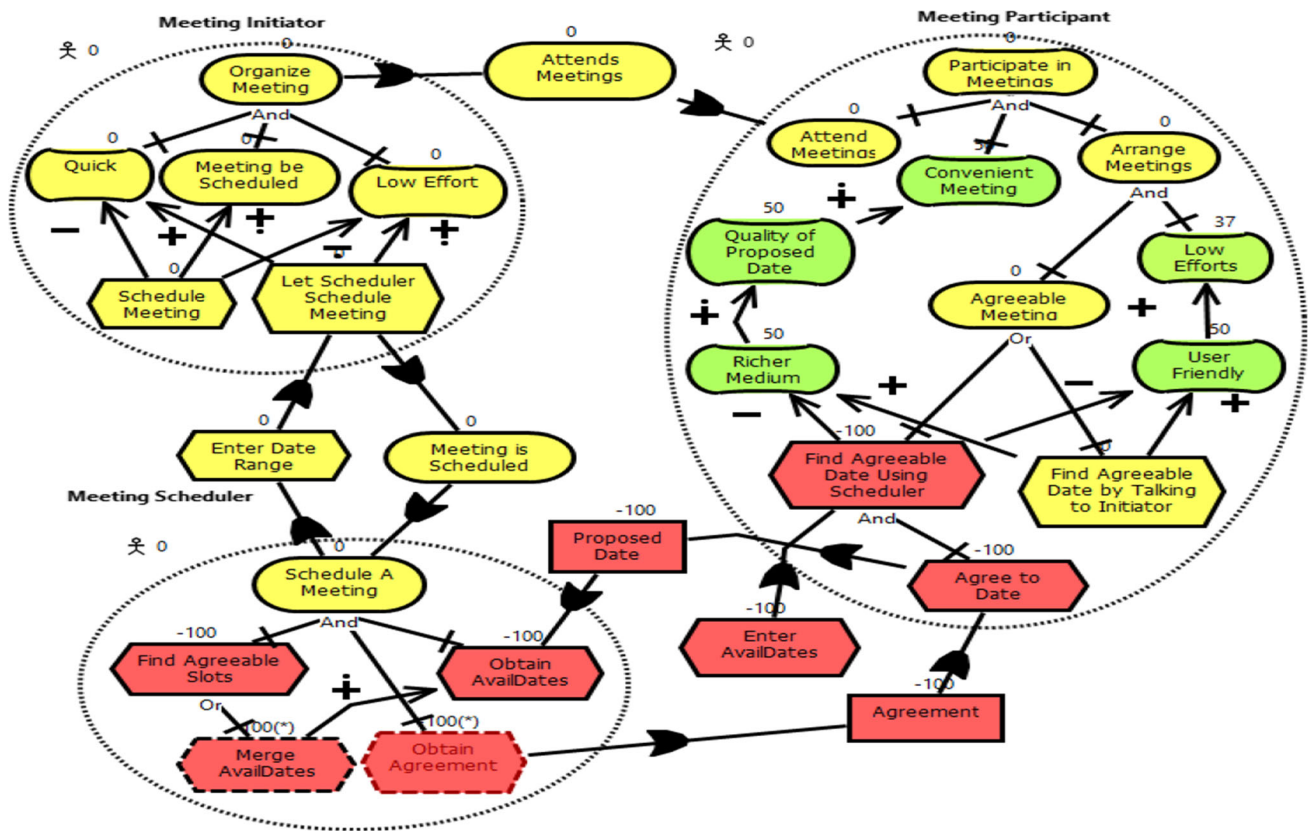


Fig. 4 The propagation of evaluation values when the second cycle is broken

expressed using the dependency links, and (2) implicit evaluation dependencies expressed using contributions, correlations, and decomposition links [69]. In contribution links, for example, in Fig. 1, the “Low Effort” softgoal is the destination of the propagation and the “Let Scheduler Schedule Meeting” task is the source of this propagation. Therefore, the “Low Effort” softgoal, partially, depends, in its evaluation, on the “Let Scheduler Schedule Meeting” task. In decomposition links, for example, in Fig. 1, the “Organize Meeting” goal is the destination of the propagation and the “Quick” softgoal is the source of this propagation. Therefore, the “Organize Meeting” goal, partially, depends, in its evaluation, on the “Quick” softgoal. In dependency links, for example, in Fig. 1, the “Let Scheduler Schedule Meeting” task is the destination of the propagation and the “Meeting is scheduled” is the source of this propagation. Therefore, the “Let Scheduler Schedule Meeting” task, partially, depends, in its evaluation, on the “Meeting is scheduled” goal.

We define the circular dependency bad smell as follows:

- Let:**  $G$  denotes a GRL model,
- $L$  denotes the set of links in  $G$ ,
- $E$  denotes the set of elements in  $G$ , and
- $C$  denotes the set of cycles in  $G$ .

**Let:**  $k$  be a positive integer,

$n$  be the number of links in  $G$ , i.e.,  $n = |L|$ ,  
 $s$  stands for the dependency Source,  
 $d$  stands for the dependency Destination, thus,  
 $l_{i,s}$  stands for the source element of link  $i$ , and similarly,  
 $l_{i,d}$  stands for the destination element of link  $i$ ,

In addition, in the GRL standard, two distinct elements cannot be directly linked to each other using more than one link; hence,  $k$  should be greater than 2. Given all of that, then, we can say:

$$\forall k \in [3, n], n \geq 3$$

$$\forall l \in L,$$

$$\forall l_s, l_d \in E,$$

$$\forall l_1, l_2, \dots, l_k \subseteq L \in C \Leftrightarrow$$

$$\forall l_i \in C \exists l_j, l_k \in C : l_{i,s} = l_{j,d} \wedge l_{i,d} = l_{k,s}$$

This definition states that for a set of links of size  $k$ , where  $k$  is a positive integer in the range  $3 \leq k \leq n$ , and  $n$  is the number of links in the model, then, this set of links is said to

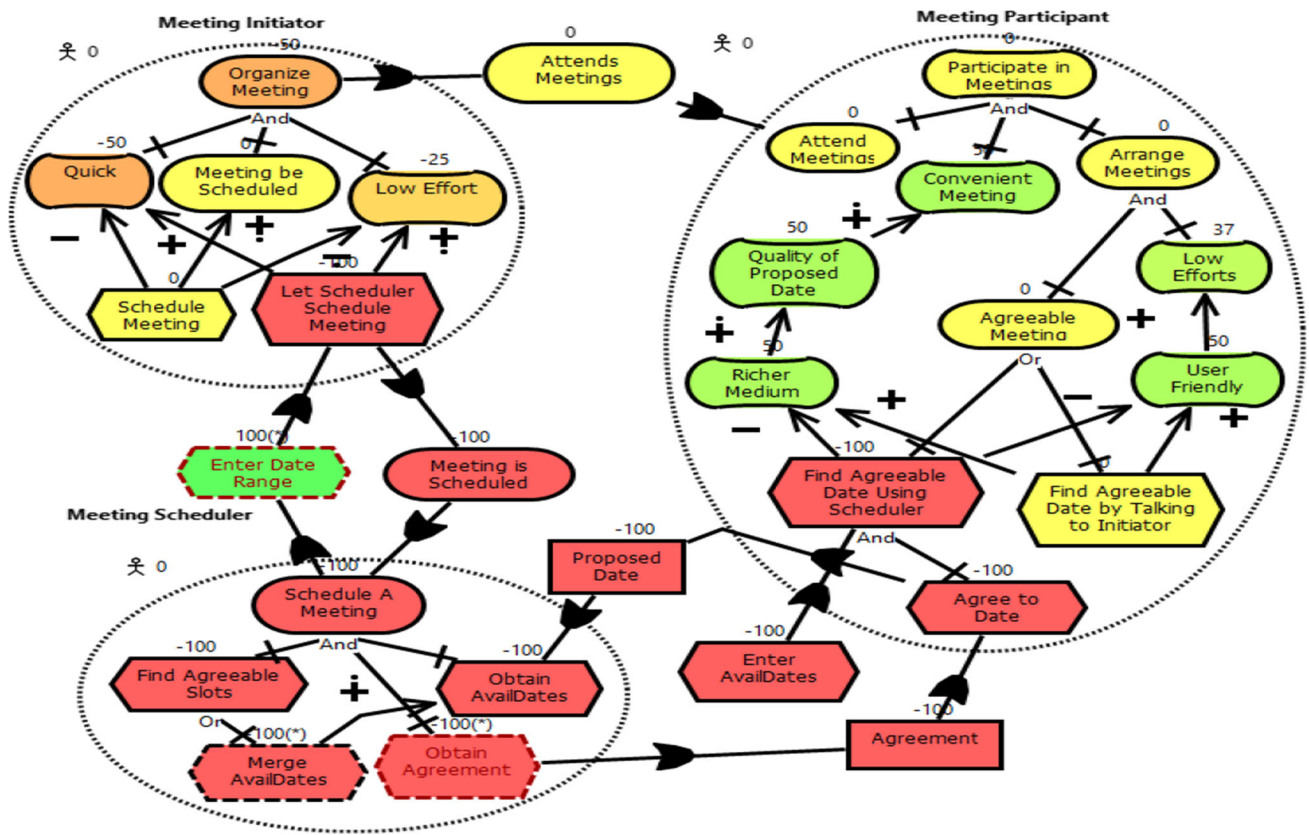
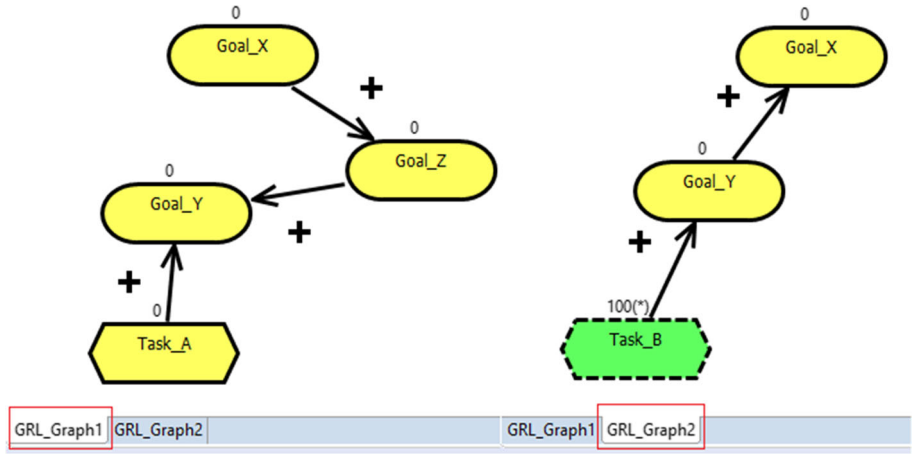


Fig. 5 The propagation of evaluation values when the third cycle is broken

Fig. 6 A model of two graphs and a cycle



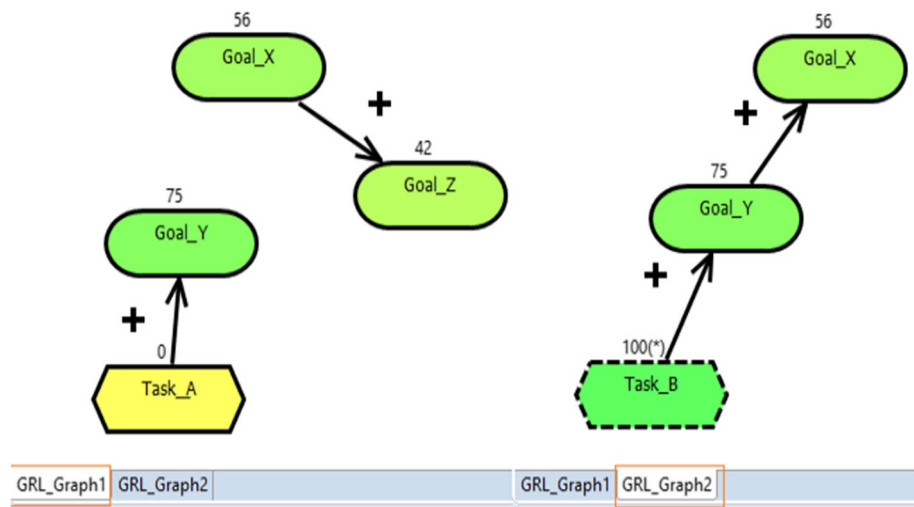
be a dependency cycle if and only if the source element of each link in this set is the destination element of another link in the same set and the destination element of each link in the set is the source element of another link in the same set.

### 4.2 Problem formulation

The problem of detecting cycles can be formulated as a search problem and solved using a search technique. Before deciding on the technique to be used in detecting the instances of

circular dependency bad smell, we analyzed the complexity of the problem’s search space. We found that the complexity comes from three sources. The first source of complexity is the length (i.e., the number of links in a cycle) of cycles. The smallest cycle in a GRL model consists of three links, but this length may grow, in some cases, to include all the links in the addressed model. The second source of complexity is the number of cycles in the addressed model. The number of cycles can grow exponentially (more in Sect. 0). The third source of complexity is the huge number of permutations

**Fig. 7** The model shown in Fig. 6 after breaking the cycle



(i.e., a list of distinct links) associated with each possible length of cycles. For example, if the length of a cycle is 3, the maximum number of permutations is  $n(n-1)(n-2)$  which is  $O(n^3)$ , where  $n$  is the number of links in the model. Hence, the maximum number of permutations for all cycles can be computed as follows:

$$n(n-1)(n-2) + n(n-1)(n-2)(n-3) + \dots \\ + n(n-1)(n-2) \dots (3)(2)(1)$$

$$\sum_{k=3}^n n(n-1) \dots (n-k+1), \text{ k being the length of the cycle} \\ O(n^3) + O(n^4) + \dots + O(n^n) \approx O(n^n)$$

Although, in GRL models, the average number of links connected to each element is usually lower than the maximum possible number, in our analysis, we build on the maximum number to consider the worst-case scenario.

### 4.3 Solution identification

As we described earlier, the objective of this work is to detect dependency cycles that can hinder the application of satisfaction analysis. Manual detection of these cycles, even for small models, can be difficult, if not impossible, due to the nature of propagation among model elements and due to the use of elements' references. First, the source and destination of the propagation might not be easily recognized visually for a link. This is because the different types of links (dependency links, decomposition links, and contribution links) do not always provide visual guidance in the direction of the propagation, especially for dependency links. Therefore, cycles may not be easily recognized visually, especially when cycles include links of different types. For example, if a cycle consists of two contribution links and one dependency link.

When the contribution links are pointing in one direction, the dependency link will be pointing in the opposite direction. Second, the standard of the language introduced the concept of elements' references which enables modelers to reuse elements in different graphs within the same model [8]. However, for detecting cycles, this feature complicates the problem. To illustrate the impact of elements' references on detecting dependency cycles, consider the GRL model in Fig. 6 which is composed of two graphs – GRL\_Graph1 and GRL\_Graph2.

Each graph resides in a different tab. Two of the elements in GRL\_Graph1 have references in GRL\_Graph2 (i.e., Goal\_Y and Goal\_X). In GRL\_Graph1, these two elements are not directly linked, but their references in GRL\_graph2 are directly linked using a contribution link. We can also see that Task\_B in GRL\_Graph2 is evaluated to “satisfied” (100, green color). However, the propagation algorithm failed to propagate the evaluation values to the other elements in the same graph (i.e., GRL\_Graph2) despite that the graph does not visually seem to have cycles. In fact, this model contains a cycle involving three elements: Goal\_X Goal\_Z Goal\_Y Goal\_X, but it is distributed over both graphs. We can see that elements Goal\_X and Goal\_Z and elements Goal\_Z and Goal\_Y are connected in a clockwise direction in GRL\_Graph1. The rest of the cycle is located in GRL\_Graph2. We can see that elements Goal\_Y and Goal\_X are connected in a clockwise direction, as well, in GRL\_Graph2. Although these fragments are distributed and do not visually manifest as a cycle, they constitute a cycle that prevents the propagation of evaluation values, as shown in Fig. 6. Therefore, when the link between Goal\_Z and Goal\_Y is removed (to break the cycle), the values propagate through all elements of the cycle in the two graphs, as shown in Fig. 7

As we described earlier, manual detection of dependency cycles is not easy. Besides, scalability is the most widely known problem associated with  $i^*$ -based frameworks includ-

ing GRL [70, 71]. Therefore, the problem becomes even harder as models grow in size and complexity. Hence, to proceed with automatic detection, we will start by looking at the available algorithms for detecting all elementary cycles in a directed graph. The problem of detecting such cycles is an NP-hard problem, and its complexity has been proven to be exponential [58]. The most well-known and efficient algorithm to solve this problem is Johnson's algorithm [58]. This algorithm is based on depth-first search and its time complexity is  $O((n+e)(c+1))$ , where  $n$  is the number of vertices,  $e$  is the number of edges, and  $c$  is the number of cycles. However, this algorithm grows exponentially with the number of cycles as the number of cycles grows faster with  $n$  than the exponential of  $2^n$  [58]. Therefore, to provide a scalable technique, we turn our attention to metaheuristic techniques.

Metaheuristic techniques [73] can be classified into local and global search algorithms [49]. Local search algorithms, such as hill-climbing, are susceptible to sticking into local optima [49] which makes them less effective in non-gradient search spaces. In our case, paths, for example, can be considered as local optima as each path constitutes a configuration of links that is very close to a cycle. In such a case, a hill-climbing algorithm is expected to stick into those paths unable to proceed further. Therefore, to avoid sticking to local optima, we need to employ a global search algorithm. In the literature, several global search algorithms are proposed such as genetic algorithms, tabu search, simulated annealing. Most of these algorithms perform similarly if the problem formulation (i.e., fitness function, solution representation, tweaking mechanism, etc.) is similar [74]. Since we designed the developed tool to retrieve a single cycle each time, we selected a single solution metaheuristic technique (i.e., simulated annealing). Simulated annealing has been successfully used to solve many computer science problems. To the best of our knowledge, it has not been used to detect cycles in GRL goal models.

After defining the circular dependency bad smell and deciding on the technique to be used in detecting its instances, we present the details of the problem formulation in the following subsections.

#### 4.3.1 Model representation

To prepare a GRL model for detecting cycles, we need to represent the model in an easy to process representation. According to the definition of circular dependency in Sect. 0, a dependency cycle can be defined as a sequence of links in which the destination element of a link is the source element of the next link, and the destination element of the last link is the source element of the first link. Based on this definition, we need three pieces of information to represent a GRL model. These pieces of information include the list of links in the model, the destination element of each link, and

**Table 2** Model representation—(nx3) matrix

Link ID	Destination element ID	Source element ID
11	1	3
12	2	3
13	4	3
14	3	5
15	6	4
16	5	6
17	6	7
18	6	8
19	6	9

the source element of each link. To this end, we represent a GRL model as an (nx3) matrix, where  $n$  is the number of rows, and it equals the number of links in the model. For columns, the first column is used to store links, the second column is used to store the destination element of the corresponding link, and the third column is used to store the source element of the corresponding link. These pieces of information are stored using their IDs and used to construct candidate solutions (using link IDs) and track sequences to identify cycles (using destination element IDs and source element IDs). Table 2 illustrates the matrix corresponding to the GRL model in Fig. 8(a) stored as element and link IDs as shown in Fig. 8(b).

#### 4.3.2 Candidate solutions specification

In the context of this work and to describe the components of the developed technique, "candidate solution" is used to refer to a potential cycle in a GRL model. It represents a list of links that might or might not turn to be a complete cycle. Each list of links is a permutation of some or all the links in the addressed model. Hence, the structure of candidate solutions is specified as a list of integers of size  $n$ , where  $n$  is the number of links in the model. Each cell in the list contains zero (i.e., no link is selected) or a positive integer (i.e., a link ID). Zero cells are used to allow for scalable candidate solutions using the array data type (in C-like languages), i.e., its length grows and shrinks as needed. For example, the permutation "13 14 15" or the permutation "14 19 13 15" can be constructed without any change to the structure of candidate solutions.

To constitute a cycle, the links in a candidate solution should be consecutive, and the last link should be consecutive to the first link as well. To determine whether a list of links is consecutive, we need to use the destination element and the source element of each link in the list. Therefore, we have to refer to the representation of the model as shown in Table 2. For example, link 16 is consecutive to link 15, as the destination element (i.e., 6) of link 15 is the source element

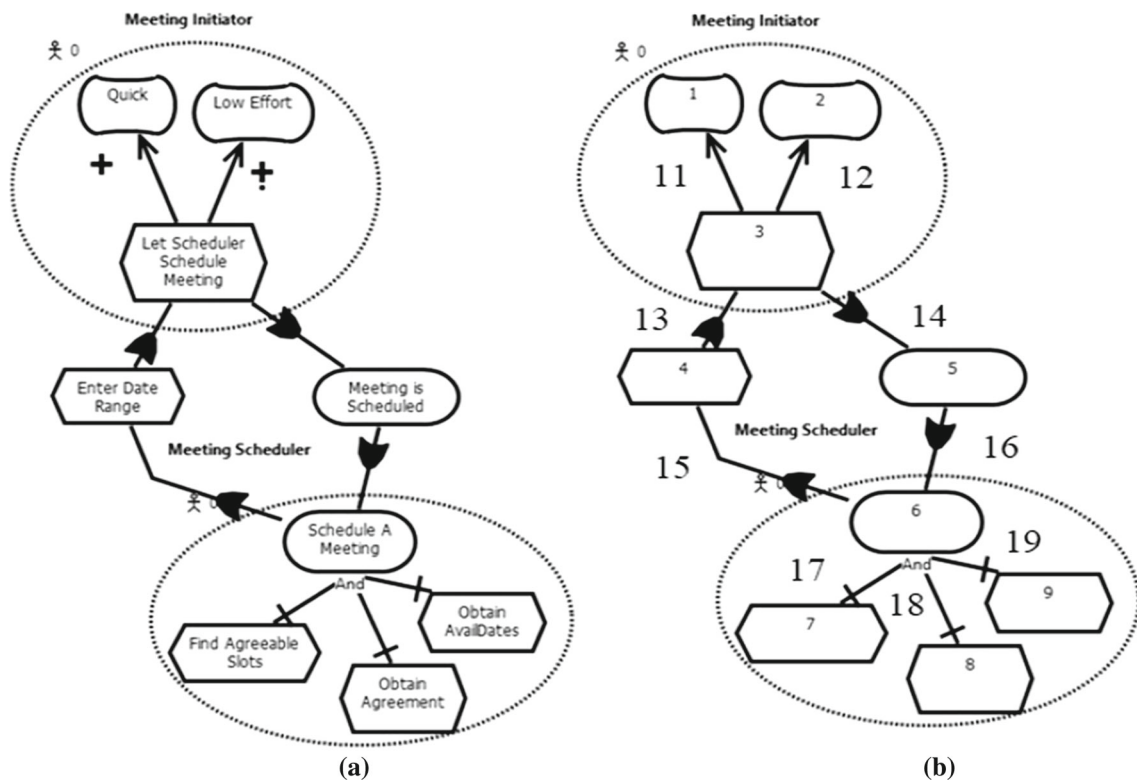


Fig. 8 A snippet of the running example in Fig. 1 and its encoded format

12	0	15	0	0	11	0	13	14
(a)								
0	13	0	14	0	16	15	0	0
(b)								

Fig. 9 Examples of candidate solutions

of link 16 (i.e., 6). Figure 9 shows two examples of two candidate solutions. By referencing the model representation as shown in Table 2, we can see that the first candidate solution (Fig. 9a) does not represent a cycle, while the second candidate solution (Fig. 9b) represents a cycle.

### 5 Circular dependency detection approach

After articulating and formalizing circular dependency as a bad smell, we developed a detection technique to detect its instances in GRL goal models. The proposed detection approach is constructed in two phases. In the first phase, the simulated annealing search algorithm is adopted as a searching mechanism. In the second phase, the structural properties of GRL models are investigated to develop heuristics that can help improve the performance of the base SA search algorithm.

### 5.1 Simulated annealing

Simulated annealing is inspired by physics and physical laws in the process of annealing in metallurgy. This algorithm is used to solve optimization problems by optimizing random search [75]. The algorithm relies on a controlled random process for finding global—upper or lower—optimum. The annealing process starts from high temperatures and, slowly, cools down. The internal thermal energy follows the decrease in the applied temperature, but it increases from time to time according to the Boltzmann law. These random increases cause the molecules to randomly change their positions and velocity each time. The same principle is applied in the SA algorithm. Starting from a high temperature, the algorithm is given enough time to explore the search space. With slight and repeated decrease in the temperature, the algorithm gradually moves from exploring the search space to exploiting parts of the search space that were found in the exploring phase to be more feasible. In some cases, the algorithm sticks to a local optimum. In this case, the random increases in the temperature allow the SA algorithm to escape this local optimum by accepting random solutions from time to time at a high rate at the beginning and a very low rate at the end.

In general, this algorithm looks for the best possible candidate solution by generating an initial random candidate solution and then exploring the adjacent areas to the cur-

rent candidate solution for a new candidate solution. If the new candidate solution is better than the current one, the new candidate solution is adopted. Otherwise, the algorithm keeps the current candidate solution. However, there might be times when the algorithm sticks to a local sub-optimal solution—local optimum. To be able to overcome this situation, the SA algorithm is designed to accept a worse candidate solution with an acceptance probability. This probability is a function of the difference between the fitness of the new candidate solution and the fitness of the current candidate solution divided by the temperature. The main flow of the algorithm is shown in Algorithm 1. The details of the main steps are presented in the following subsections.

solution by employing a tweaking mechanism. Then, the fitness values of the current and the new candidate solutions are calculated. Based on the resulting fitness values, two different cases can be distinguished. First, if the fitness of the new candidate solution is better than the fitness of the current one, the new candidate solution is adopted. Otherwise, the new candidate solution “may” be adopted with a probability (i.e., acceptance probability). This probability is brought to help the search algorithm escape local optima. At the end of each iteration, if the resulting candidate solution (i.e., “Best”) represents a cycle (has a fitness of 100%, see Sect. 0) that has not been discovered before, it will be considered. Otherwise, the resulting solution will be discarded, and the algorithm will start the next iteration. In other words, only valid cycles are kept and reported.

---

### Algorithm 1: The simulated annealing search algorithm

---

```

1: Input: M: a goal model
2: Input: T: initial temperature // 1000 in this work
3: Input: Cr: Cooling rate // 0.05 in this work
4: Process: // represents a single iteration
5: S ← generateInitialCandidateSolution ()
6: Best ← S
7: Repeat
8:   H ← tweak (S)
9:   r ← random [0,1]
10:  If ((fitness(H) > fitness(S)) ∨ (r < acceptanceProbability)) Then
11:    S ← H
12:  End If
13:  T = T – T * Cr
14:  If fitness(S) > fitness (Best) Then
15:    Best ← S
16:  End If
17: Until T < 1
18: Output: Best.

```

---

#### 5.1.1 Steps of the search algorithm

Algorithm 1 provides a coarse-grain overview of the main steps of the simulated annealing algorithm. It starts by generating an initial random candidate solution and adopting it as the current candidate solution. After that, a new random candidate solution is generated from the current candidate

The number of iterations needed to run this algorithm can be set using a trial-and-error method since this approach is acceptable in the area of search-based software engineering [76, 77] or it can be set based on some heuristics related to the problem being solved (more later in Sect. 3). In the experimental validation of our work and in order to compare the developed approaches, we set the number of iterations to 1,000,000. After comparing the different approaches, an application procedure is developed to support the developed detection technique in setting its parameters to find all the cycles in the model, if any (see subsection 3.1).

### 5.1.2 Generating initial candidate solutions

To be able to generate a random candidate solution (Algorithm 1 – Step 5), we start by constructing a list of all the links as shown in Table 2. Then, a random initial candidate solution is constructed by selecting some or all of the links, as shown in Fig. 9. To this end, the following steps are followed by the developed approach:

1. An empty list of length  $n$  (i.e., number of links in the model) is created.
2. For each position in the list, a probability is generated for its filling as part of the candidate solution. This probability is used to control the length of the solution (i.e., the sequence to be tested). For example, if that probability is set to 0.5, probably, half of the positions in the list will be links and the other half will be zeros (i.e., empty).
3. Go through all positions in the list. If that position is not chosen for filling (in step 2), it will be initialized to 0. If it is chosen, it will be initialized to a randomly selected link from the list of links in the model such that it was not already included in the solution being constructed.

By performing these steps, an initial random candidate solution will be constructed. In order to measure its fitness, we need to develop a fitness function.

### 5.1.3 Fitness function

This section presents the developed fitness function intended to be used in the context of the simulated annealing algorithm (i.e., Algorithm 1, steps 10 & 14). The developed fitness function is used to measure the proximity of a candidate solution to a complete cycle. This proximity is used to evaluate the quality of the candidate solutions. Based on this proximity, the simulated annealing algorithm processes candidate solutions. Since cycles are defined as a series of consecutive links that start and end at the same vertex, to define the fitness function, the number of consecutive links is used as the main component of evaluating the fitness of candidate solutions. To accommodate the different lengths of cycles, we divide the number of consecutive pairs of links in a candidate solution by its length. Therefore, in evaluating the fitness of candidate solutions, the more the consecutive links in a candidate solution, the closer this candidate solution is to a complete cycle.

As we described earlier, in this work, computing the fitness value of a solution is concerned with reflecting how a list of links is close to a complete cycle. Hence, two cases can be distinguished:

1. The candidate solution contains 0 or 1 link only the fitness value equals 0% as no consecutive links exist.

2. Otherwise, the fitness value is computed using the following equation:

$$\text{Fitness} = \frac{\text{Number of consecutive pairs of links in the solution}}{\text{Number of included links in the solution}} * 100$$

It is important to notice that the first link should be examined whether it is consecutive to the last link in the evaluated candidate solution. Hence, the application of the fitness function on the examples in Fig. 9 results in the following values:

- For the candidate solution in (a)  $(1/5) * 100 = 20\%$ . For the numerator, “1” represents the number of consecutive pairs of links in the candidate solution (a). This pair is “13, 14.” For the denominator, “5” is the number of included links in the candidate solution (a): “12, 15, 11, 13, 14.”
- For the candidate solution in (b)  $(4/4) * 100 = 100\%$ . The numerator (i.e., 4) represents the number of consecutive pairs of links in the candidate solution (b). These pairs are “13,14,” “14,16,” “16,15,” “15,13.” For the denominator, “4” represents the number of included links in the candidate solution (b): “13, 14, 16, 15.”

### 5.1.4 Tweaking mechanism

The tweaking mechanism is used in the context of the simulated annealing algorithm (i.e., Algorithm 1, step 8), to generate a new adjacent random candidate solution to exploit the neighborhood of the current candidate solution. To this end, a small change is introduced to the current candidate solution using the following simple procedure:

1. Copy the old candidate solution to a new candidate solution.
2. Pick a random position in the new candidate solution.
3. If the value of the selected random position is not zero, change it to zero. This corresponds to the deletion of a link from the old candidate solution.
4. If the value of the selected random position is zero, pick a random link from the list of links in the model such that it was not already included in the new candidate solution. Then, insert this link into the selected random position. This corresponds to the addition of a new link to the old candidate solution.

### 5.1.5 Acceptance function

The acceptance function is developed to calculate the acceptance probability used in the context of the simulated annealing algorithm (i.e., Algorithm 1, step 10). Mathematically, the starting temperature must be high enough to allow the SA algorithm to accept new random candidate solutions at a high rate at the primitive stages. By doing so, it is possible to move to any new random candidate solution exploring the



search space effectively resembling a random search. Otherwise, the algorithm will stick to the first initial candidate solution or a very close one resembling the hill-climbing search. Every time the temperature is reduced, the probability of accepting a new worse candidate solution is reduced, until finally, the probability becomes close to zero. In such a case, rarely, a new worse candidate solution is accepted. To manage the ratio of reducing the temperature, a geometric reduction mechanism is adopted in this work. This mechanism gradually reduces the temperature geometrically. This reduction mechanism is presented in Eq. 1.

$$T' = T * (1 - Cr) \quad (1)$$

where.

- $T'$  is the new temperature,
- $T$  is the old temperature,
- $Cr$  is the cooling rate.

Based on the already described parameters, the acceptance probability is evaluated according to Eq. 2.

$$P = e^{-(Q' - Q)/T} \quad (2)$$

where.

- $P$  is the probability of acceptance. It is compared to a randomly generated probability  $P'$ . If  $P > P'$ , the new worse candidate solution will be accepted. Otherwise, the new worse candidate solution will be rejected,
- $Q'$  is the fitness value of the new candidate solution,
- $Q$  is the fitness value of the old candidate solution.

In this work, the maximum fitness value ( $Q$ ) is 100, the initial temperature ( $T$ ) is set to 1000, and the cooling rate ( $Cr$ ) is set to 0.05. Analytically, these values allow the SA algorithm to start exploring the search space and to end exploiting the candidate solution neighborhood. The following examples are provided to clarify this idea. Assume fixed values for the fitness of the old and the new candidate solutions (i.e.,  $Q = 95$  and  $Q' = 40$ ). In the beginning, the value of temperature is 1000, and close to the end, the value of the temperature can be 10. To calculate the acceptance probability at the beginning and close to the end, we apply Eq. 2 in both cases as follows:

$$Q = 95, \quad Q' = 40, \quad T = 1000 \text{ (i.e., at the beginning)}$$

$$\rightarrow P = e^{-(40-95)/1000} = 0.95$$

$$Q = 95, \quad Q' = 40, \quad T = 10 \text{ (i.e., close to the end)}$$

$$\rightarrow P = e^{-(40-95)/10} = 0.004$$

The obtained probabilities indicate that the algorithm starts exploring the search space by accepting new worse candidate solutions at a high rate, as the temperature is still high. With the repeated reductions in the temperature and close to the end, the algorithm starts exploiting the neighborhood of the current solution and hardly accepts worse candidate solutions.

To summarize the acceptance procedure, we calculate the quality (i.e., fitness) for the new and current candidate solutions. If the new candidate solution is better, it will be accepted. Otherwise (i.e., if the new candidate solution is worse than the old candidate solution), the probabilities  $P$  and  $P'$  are used to determine which candidate solution will be kept. If  $P > P'$ , the new candidate solution is adopted. Otherwise, the old candidate solution is kept.

## 5.2 Structural properties of GRL models

The structural properties of a software artifact can effectively contribute to processing it for various purposes. With respect to detecting the instances of circular dependency bad smell in GRL models, the structure of GRL models is found very helpful. Based on our analysis of the structure of these models, we introduce two heuristics, namely pruning and pairing, that can improve the performance of the SA algorithm. The pruning heuristic is intended to develop a mechanism to prune the addressed model before starting the search algorithm or after resolving each cycle. The pairing heuristic is intended to develop a mechanism to improve the performance of the tweaking mechanism.

### 5.2.1 Pruning mechanism

As shown in Sect. 0, the search space of the potential cycles in GRL models is exponential with the number of cycles in the model. Therefore, the availability of a heuristic that can reduce the search space will be extremely handy. With this in mind, we analyzed GRL models and observed that these models usually have a tree-like structure concerning the direction of evaluation propagation. Hence, several elements usually have either incoming or outgoing links only. These elements are either destination elements or source elements only. In other words, these elements cannot be part of a cycle as each element in a cycle should have at least one incoming link and one outgoing link. Thus, we can prune these elements and their associated links without affecting our endeavor of detecting circular dependencies. Pruning these elements, in turn, might expose other elements that have either incoming or outgoing links only. The process of pruning these elements continues until each element in the model has at least one incoming link and one outgoing link. This process results in reducing the size of the GRL model, and consequently, a reduction in the search space. For example, in Fig. 8(b),

the number of links in the model is 9 and the number of elements is 9. Each of the elements 1, 2, 7, 8, and 9 is linked to the rest of the model with a single link. Therefore, they cannot be part of a cycle. Consequently, these elements and their associated links will be pruned, resulting in a pruned model (see Fig. 10) and a reduced search space. The steps of the pruning mechanism are presented in Algorithm 2. This algorithm iterates over every element in the model. For each element, it extracts the set of source elements and the set of destination elements. If any of these sets is empty, this element is pruned. This process continues until all elements in the model are left to have at least one source element and one destination element.

Algorithm 2: The pruning mechanism

---

```

1: Input: M: a model
2: Input: N: # of links in M
3: Process:
4: Repeat
5:   For All  $e \in$  elements in M Do
6:      $srcs \leftarrow extractSources(e)$ 
7:      $dsts \leftarrow extractDestinations(e)$ 
8:     If  $isEmpty(srcs) \vee isEmpty(dsts)$  Then
9:        $removeFrom(e, M)$ 
10:    End If
11:  End For
12: Until  $\forall e \in$  elements in M  $\exists$  at least one source and one destination
13: Output: PM: a pruned model

```

---

### 5.2.2 Pairing mechanism

After applying the pruning mechanism, all the remaining elements in the pruned model are left connected to at least two links—one incoming and one outgoing link—see Fig. 10 as an example. This property can be utilized to improve the search process by considering pairs of links connected to each element as follows:

1. Randomly pick an element from the set of the elements in the pruned model.
2. Randomly select one incoming link and one outgoing link from the set of links connected to the selected element (i.e., pairing). It is important to mention that outgoing and incoming terms refer to the direction of the propagation between elements as explained in Sect. 0.
3. Put these two links in randomly selected consecutive positions in the candidate solution being constructed.

This property is adopted and integrated into the tweaking mechanism as shown in Algorithm 3. It starts by copying the old candidate solution into a new candidate solution. Then, a random position in the new candidate solution is picked. If the value of the selected random position is not zero, change it to zero. This corresponds to deleting a link from the old candidate solution. Otherwise, if the value of the selected

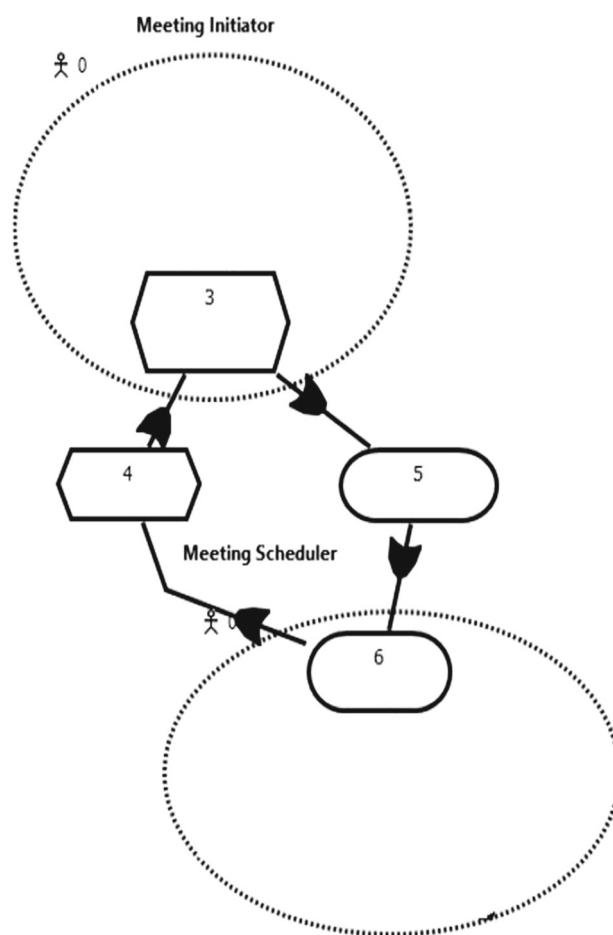


Fig. 10 The snippet in Fig. 8 after pruning – A pruned model (PM)

random position is zero, pick a random element from the list of elements in the pruned model. Then, from the list of links connected to that element, randomly pick one incoming and one outgoing link. After that, consider the following cases:

- If this pair of links does not exist in the old candidate solution insert them in two consecutive positions starting by the randomly selected position, regardless of the content of the next position, such that the outgoing link is in a consecutive position to the incoming link.
- If one of the links in this pair already exists in the old candidate solution and the other one does not insert the new link such that the outgoing link is in a consecutive position to the incoming link whatever the content of the new position was.
- If this pair of links does exist in the old candidate solution swap the outgoing link to be in a consecutive position to the incoming link if they are not.

The new tweaking mechanism is developed to replace tweak (S) that is presented in Sect. 5.1.4 and used in Algorithm 1—Step 8. To clarify this mechanism, we provide an example to show how a new candidate solution can be gen-

erated from an old candidate solution using this mechanism. Referring to the pruned model in Fig. 10, assume that the old candidate solution is “13, 16, 15, 14.” To generate a new candidate solution from this candidate solution, a random position in the old candidate solution is selected. Assume position 2 is selected (i.e., position 1 in C-like programming languages). Since the content of this location is “16” which does not equal 0, a random element will be selected from the set of elements in the pruned model. Assume element 4 is selected. Then, one of its incoming dependency links and one of its outgoing dependency links will be selected. Assume links “15” and “13” are selected. Since they are already included in the old solution, the outgoing link (i.e., 13) will be moved to the consecutive location of the incoming link (i.e., 15) and the rest of the candidate solution will be adjusted accordingly. Hence, the resulting new candidate solution will be “14, 16,15,13” which constitutes a cycle.

Algorithm 3: A new tweaking mechanism based on pairing

```

1: Input: PM: Pruned model
2: Input: N: # of the remaining links in PM
3: Input: OS: The old candidate solution
4: Process:
5:   Create an empty list NS
6:   NS ← OS
7:   r ← Random [0, N [
8:   If NS[r] ≠ 0 Then
9:     NS[r] = 0
10:  Else
11:    pick a random element e from PM
12:    inLink ← extractIncomingLinkRandomly (e)
13:    outLink ← extractOutgoingLinkRandomly (e)
14:    If (inLink ∉ NS) ∧ (outLink ∉ NS) Then
15:      NS[r] ← inLink
16:      NS[(r+1)%N] ← outLink
17:    Else If (inLink ∈ NS) ∧ (outLink ∉ NS) Then
18:      l ← findPosition (inLink, NS)
19:      NS[(l+1)%N] ← outLink
20:    Else If (inLink ∉ NS) ∧ (outLink ∈ NS) Then
21:      l ← findPosition (outLink, NS)
22:      NS[(l+N-1)%N] ← inLink
23:    Else If (inLink ∈ NS) ∧ (outLink ∈ NS)
24:      Then
25:        l ← findPosition (inLink, NS)
26:        NS[(l+1)%N] ← outLink (by swap)
27:    End If
28:  End If
29: Output: NS: A new candidate solution

```

### 5.3 Constructed approaches

We developed a search-based technique to detect the instances of circular dependency bad smell by integrating the

Table 3 The developed approaches

Approach	Simulated annealing	Pruning mechanism	Pairing mechanism
SA	✓	✗	✗
SAP	✓	✓	✗
SAPP	✓	✓	✓

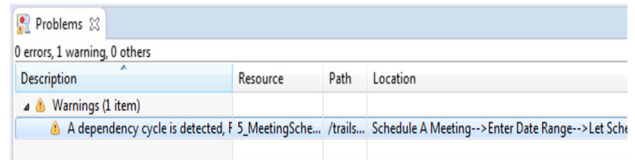


Fig. 11 A snapshot of the results as presented in the developed plugin

pruning and pairing mechanisms into the simulated annealing search algorithm. To evaluate the efficiency of the simulated annealing search algorithm and the developed mechanisms, we constructed three approaches. Each of these approaches is built on top of the previous one by adding a new mechanism as presented in Table 3. Approach 1 uses only the Simulated Annealing search algorithm (abbreviated as SA). Approach 2 uses the Simulated Annealing search algorithm along with the Pruning mechanism (abbreviated as SAP). Approach 3 uses the Simulated Annealing along with the Pruning and Pairing mechanisms (abbreviated as SAPP).

The SA approach uses the base simulated annealing to search for cycles (i.e., Algorithm 1) and operates on the original model. The SAP approach also uses the base simulated annealing to search for cycles. However, it does not operate on the original model. Rather, it operates on a pruned model obtained by applying the pruning mechanism (i.e., Algorithm 2) on the original model. The SAPP approach also uses the base simulated annealing to search for cycles. However, it operates on a pruned model (i.e., Algorithm 2) and employs a new tweaking mechanism (i.e., Algorithm 3).

### 5.4 Tool support

We developed a new tool implementing the SAPP approach to support the adoption of the outcomes of this study in practical settings. This tool is developed as an open-source Eclipse plugin. It is free to use and it works on top of the Eclipse framework and the jUCMNav tool. Figure 11 illustrates a snapshot of the results of running the tool showing an example of a detected cycle. The detected cycle and its information are shown in the “problems” view of the Eclipse workbench.

This tool is developed according to the application guide (see Sect. 3.1, algorithm 4). It detects a cycle and then asks the modeler to resolve it and runs the tool again. When all cycles are resolved, or when the model is free of cycles, the

tool informs the modeler of the absence of cycles. Installation and using instructions along with the source code of this tool are available on GitHub.<sup>2</sup>

## 6 Experimental evaluation

In this section, we introduce GRL models and performance metrics used in the experimental evaluation and comparison of the proposed approaches. The objective of presenting and comparing the different approaches is to show the improvements brought by the pruning and pairing mechanisms. Therefore, we started by evaluating the performance of the SA approach. Then, to evaluate the effectiveness of the pruning mechanism, we compared the performance of the SAP with the performance of the SA. Next, to evaluate the effectiveness of the pairing mechanism, we compared the performance of the SAPP approach with the performance of the SAP approach.

### 6.1 Description of the experimental GRL models

Seven GRL models are used to evaluate and compare the developed approaches, as shown in Table 4. These models have different characteristics in terms of the number of injected cycles (i.e., instances of the circular dependency bad smell), number of actors, number of elements, and number of links. The injected cycles are intentionally designed to address the scalability and complexity aspects. Table 4 summarizes the GRL evaluation models and the intent behind each model. Model 1 includes several cycles, where some links participate in more than one cycle. It is intended to evaluate and compare the developed approaches when cycles are overlapping. On the contrary, GRL model 4 is intended to evaluate and compare the developed approaches when the model is not dense with cycles. GRL models 2 and 3 are intended to evaluate and compare the developed approaches on models with different configurations of cycles. GRL model 5 is intended to evaluate and compare the performance of the developed approaches when the model contains a single cycle that includes all links in the model. Model 6 is used to evaluate the performance of the developed approaches when the model is acyclic (i.e., when the model is free of circular dependencies). Models 1 up to 6 are artificially constructed to evaluate the effectiveness of the developed techniques.

To increase our confidence in the developed tool and ensure its scalability, we used a real-world model (i.e., Model 7 as shown in Table 4). Model 7 is constructed in the context of the Adapting Service lifeCycle toward Efficient Clouds

(ASCETiC) project<sup>5</sup> & <sup>6</sup>. This project is intended to develop tools and methods covering the whole lifecycle of a cloud service taking energy efficiency into account. Specifically, Model 7 was developed to model the application of an experiment-based methodology to determine an appropriate application cloud deployment. Real-world models, as in Model 7, tend to spread over several graphs. Therefore, elements can be referenced in many graphs in the same model using element's references. Each element's reference might be linked to different elements compared to the other references in the same model. With respect to its size, model 7 consists of 15 graphs; each graph contains elements and links. Some of these elements are repeated as references in different graphs within the same model. Although the number of distinct elements is 151 (as shown in Table 4), the number of elements and references is 288. Furthermore, the model contains other types of constructs such as beliefs and indicators. Although beliefs and indicators do not influence satisfaction analysis of dependency cycles, they make the manual detection of cycles harder as they overwhelm the model visually. In total, model 7 contains 763 visually recognizable elements. The model is originally free of cycles; hence, we can control the injection of cycles. We add one cycle at a time to the model. Every time we introduce a cycle, we test the introduction of that cycle separately to ensure that the introduction of each cycle does not introduce more than one cycle as some of the elements in this model have more than one reference. This is important in order to have the right number of cycles to calculate the recall correctly.

### 6.2 Performance measurement

To measure the performance of the proposed approaches, two performance aspects need to be considered: accuracy and time. Precision and recall metrics are frequently used to measure the accuracy in the literature [71]. In the context of this research, we define precision and recall as follows:

$$\text{Precision} = \frac{(\text{number of retrieved cycles})}{(\text{number of retrieved valid solutions})}$$

$$\text{Recall} = \frac{(\text{number of retrieved cycles})}{(\text{number of cycles in the model})}$$

Since the developed approaches are designed to return complete cycles only (i.e., valid instances of the circular dependency bad smell), the precision metric returns "1" (i.e., 100%) all the time. In other words, when one of the developed approaches is evaluated on an evaluation model, the evaluated approach either returns complete cycles or does not return anything at all. Therefore, the precision measure

<sup>2</sup> <https://github.com/MawalMohammed/Circular-Dependency-in-GRL>

<sup>5</sup> <http://www.ascetic.eu/>

<sup>6</sup> <https://github.com/ascetictoolbox/ascetictoolbox>

**Table 4** The summary of evaluation models

Model	Intent	No. of actors	No. of elements	No. of links	No. of cycles	Cycles' length
Model 1	To test the capability of the developed approaches when a link participates in more than one cycle	2	11	15	5	4, 4, 4, 5, 6
Model 2	To test the capability of the developed approaches to detect cycles when they expand over several actors	3	17	18	2	3, 6
Model 3	To test the capability of the developed approaches to detect cycles within cycles. It is also intended to test the scalability of the developed approaches in detecting lengthy cycles	4	25	28	3	4, 4, 9
Model 4	To evaluate the scalability of the developed approaches. This model grows big in size compared to the previous models. It also includes a lengthy cycle. Furthermore, this model is intended to test the capability of the developed approaches to detect relatively small cycles in a large pool of links	5	67	71	4	4, 4, 5, 13
Model 5	To investigate the behavior of the different approaches when all the links in the model participate in one cycle	4	9	9	1	9
Model 6	To investigate the behavior of the different approaches when the model does not contain any cycle	5	67	67	0	NA
Model 7 <sup>4</sup>	To evaluate the behavior of the developed approaches on a real-world case study	0	151	248	3	3, 5, 7

here is irrelevant as it does not help differentiate the evaluated approaches in terms of their accuracy. To clarify this point, assume that the first approach was run on an evaluation model and returned 2 cycles out of the 4 cycles in that model, and assume that the second approach was run on the same evaluation model and returned 3 cycles out of the 4 cycles. In this case, the precision of the first approach is  $2/2$  which equals 1 and the precision of the second approach is  $3/3$  which equals 1 as well. Hence, precision is not a differentiating measure. Therefore, the recall measure is used to evaluate the accuracy of the developed approaches. Contrary to precision, the recall measure provides differentiating ratios that can help assess the accuracy of the proposed approaches. To clarify this, assume that the first approach was run on an evaluation model and returned 2 cycles out of the 4 cycles in that model, and assume that the second approach was run on the same test model and returned 3 cycles out of the 4 cycles. In this case, the recall of the first approach is  $2/4$  which equals 0.5 and the recall of the second approach is  $3/4$  which equals 0.75. We can see that the recall ratios of the two approaches are different.

The time aspect of the developed approaches is measured in terms of the number of iterations. The number of iterations was chosen because it is independent of the hardware used. However, to give the reader a perception of the time required to run the iterations in the different approaches, the time needed—in terms of seconds—to run the different approaches is also provided. In this paper, the algorithm was run on a laptop with an Intel Core i5 2.6 GHz microprocessor and 4G RAM.

### 6.3 Stopping condition setting

For experimental purposes, the stopping condition should be chosen such that it allows for assessing and analyzing the behavior of the developed approaches in various situations exposing their strengths and weaknesses. Therefore, we designed the stopping condition in the experimental evaluation of the developed approaches such that it is reached in one of two cases: either by detecting all the cycles in the subject evaluation model or reaching the maximum number of iterations. Since the number of cycles in each model is known, we are left with specifying the maximum number of iterations. To this end, to be effective in evaluating the developed approaches, the maximum number of iterations should be high enough to raise questions if it was reached without detecting all or some of the cycles. Based on experimental pilot trials, the maximum number of iterations is set to 1000,000. This condition allows us to study the performance of the different approaches in various situations. However, it is not practical in real-world settings (more in Sect. 3.1).

## 7 Results

The developed approaches (i.e., SA, SAP, and SAPP) are evaluated using the seven GRL evaluation models introduced in Table 4. Before discussing the obtained results, it is important to mention that only the summaries of the results are presented in this section, as shown in Table 5. The detailed results along with the source code used to implement the proposed approaches are uploaded with the experimental data on

the paper website.<sup>7</sup> The instructions on how to run the source code and collect the obtained results are also available in the readme file associated with these materials.

Metaheuristic search algorithms, such as the simulated annealing search algorithm, might have slightly different behavior and results every time they run. Hence, to build our conclusions on solid ground, each of the proposed approaches was run 10 times on each evaluation model (i.e., a total of 210 runs; each of the three proposed approaches was run 10 times on each of the seven evaluation models). Table 5 shows the results of evaluating each approach on each evaluation model. In each model, the mean ( $\mu$ ) and standard deviation ( $\sigma$ ) of the number of iterations performed to detect each cycle when each of the proposed approaches is run 10 times on each evaluation model are presented. For example, in model 1 cycle 1, the mean and standard deviation of the number of iterations needed to detect the first cycle when the SA approach was run 10 times are 67.4 and 54.05, respectively.

### 7.1 First approach: SA

This approach is solely based on the simulated annealing search algorithm as shown in Algorithm 1. In this approach, no pruning nor pairing has been used to augment the simulated annealing search algorithm.

Table 5 shows the results of evaluating this approach on each evaluation model (i.e., the SA row in each model). Specifically, it shows the mean and standard deviation of running this approach on each model 10 times. This approach successfully retrieved all the cycles in model 1 and model 2. Therefore, the obtained recall was 100% in each case. In model 3 and model 4, the obtained recall values were 66.7% and 0%, respectively. Only 2 cycles out of the 3 cycles in model 3 were detected and no cycle at all is detected in model 4. This can be justified by the size of the model being evaluated in terms of the number of links and the length of cycles. Models 1 and 2, having sizes of 15 and 18, respectively, are smaller compared to the sizes of models 3 and 4 (28 and 71, respectively) as shown in Table 4. Furthermore, the longest cycles in models 1 and 2 (of length 6) are smaller than the longest cycles in models 3 and 4 (of lengths 9 and 13, respectively) as shown in Table 4.

As mentioned earlier in the experimental evaluation section (i.e., Sect. 0), the stopping condition of the developed approaches is either detecting all the cycles in the subject model or reaching the maximum number of iterations, which is 1,000,000. For models 3, 4, and 7, in which the condition of detecting all the cycles is not met, the SA approach performed the maximum number of iterations. Investigating the obtained results with model 3 indicates that this approach

failed to detect the third cycle which is the longest cycle in this model. Hence, the SA approach performs unsatisfactorily when the existing cycles are lengthy. In addition, this approach has another weakness when it comes to models with a large number of links. This is apparent in the results associated with model 4, which has 71 links. Although the model includes cycles of short length (i.e., 4 and 5), the algorithm failed to retrieve any. The SA approach was also unable to detect any of the cycles of model 7 due to the large size of this model as well. It also took a longer time when compared to the time it took with model 4 for the same reason.

Model 6 does not contain cycles (i.e., a non-cyclic model), as shown in Table 4. However, the obtained results showed that the SA approach reached the maximum number of iterations (i.e., 1,000,000, until the stopping condition is met). The SA approach needed to perform the maximum number of iterations as it does not have any mechanism to recognize the absence of cycles.

As we can see in this section, the size of the model and the length of the cycles are the two factors that limit the capability of the simulated annealing search algorithm to retrieve dependency cycles. This confirms our analysis of the complexity of the problem in Sect. 0. Hence, augmenting the simulated annealing algorithm with additional information (see Sect. 0) to obtain a better convergence rate is very helpful as it will be discussed in the following two subsections.

### 7.2 Second approach: SAP

In this approach, the developed pruning mechanism is used to improve the performance of the simulated annealing search algorithm by reducing the size of the subject model. This reduction in the number of links results in a reduction in the search space which, in turn, results in an improvement in the obtained results using the SAP approach compared to the results obtained using the SA approach. Before applying the simulated annealing algorithm, the sizes of evaluation models are reduced using the developed pruning mechanism as described in Algorithm 2. Table 2 shows the percentage of reduction in the number of links using the pruning mechanism in each evaluation model. We can notice that the number of links is reduced across all models with different percentages except for model 5 and model 6. In model 5, no link was pruned because all the links in model 5 participate in the only cycle in that model. In model 6, all links are pruned because this model does not have any cycle, as shown in Table 4.

The results of applying the SAP approach to the seven evaluation models are presented in Table 5. The obtained results show significant improvements compared to the SA approach. In terms of the recall metric, both approaches performed perfectly with model 1 and model 2. However, the differences started to show in model 3. In model 3, there is an improvement in the recall metric by 6.6% using the SAP

<sup>7</sup> <http://softwareengineeringresearch.net/GRLSBSE.html>.

**Table 5** The results of evaluating the proposed approaches on the evaluation models for 10 runs

Approach	Cycle 1		Cycle 2		Cycle 3		Cycle 4		Cycle 5		Recall		Time (s)	
	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$
<i>Model 1</i>														
SA	67.4	54.05	169.6	70.57	271.4	105.6	571.7	286	8341.2	6747.3	1	0	0.731	0.57
SAP	10	8.1	54.8	30.8	96	53.09	290.2	272.8	2279.4	1832.8	1	0	0.179	0.12
SAPP	1.8	1.24	2.9	1.97	5.3	2.36	10.1	5.68	16.9	9.73	1	0	0.005	0.007
Approach	Cycle 1		Cycle 2		Recall		Time (s)							
	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$						
<i>Model 2</i>														
SA		337.4		444.2		21,560.2		13,217.5	1	0	2.032		1.28	
SAP		22.5		15.19		3141.4		3446.4	1	0	0.2617		0.27	
SAPP		7.9		7.3		19.4		12.3	1	0	0.0067		0.011	
Approach	Cycle 1		Cycle 2		Cycle 3		Recall		Time (s)					
	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$				
<i>Model 3</i>														
SA	54,812.1	57,505.9	102,180.4	64,742.3	NF	NA	0.67	0	169.3225	13.46				
SAP	327.7	311.8	1354.9	768.65	827,394.7	116,021.5	0.73	0.13	90.52	37.9				
SAPP	78.4	87.48	209.4	115	835.1	551.9	1	0	0.1018	0.097				
Approach	Cycle 1		Cycle 2		Cycle 3		Cycle 4		Recall		Time (s)			
	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$		
<i>Model 4</i>														
SA	NF	NA	NF	NA	NF	NA	NF	NA	0	NA	615.56	29.56		
SAP	879.2	972.4	3120.9	1409.5	14,164.7	7546.8	NF	NA	0.75	0	117.41	6.1		
SAPP	194.3	153.2	356.3	189.2	1001.8	497.76	84,004.3	68,552.3	1	0	9.35	7.60		
Approach	Cycle 1		Time (s)											
	$\mu$	$\sigma$	$\mu$	$\sigma$										
<i>Model 5</i>														
SA			132,721.3	121,958.9487	9.88	9								
SAP			133,656.4	88,715.591	10.982	7.51								
SAPP			132.3	136.04	0.0239	0.0302								
Approach	Number of iterations performed		Time (s)											
	$\mu$	$\sigma$	$\mu$	$\sigma$										
<i>Model 6</i>														
SA			1,000,000	0	580.47	26.65								
SAP			0	0	0	0								
SAPP			0	0	0	0								
Approach	Cycle 1		Cycle 2		Cycle 3		Recall		Time (s)					
	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$				
<i>Model 7</i>														
SA	NF	NA	NF	NA	NF	NA	0	NA	6519.873	0				
SAP	48.8	30.91	763.2	788.65	30,642.9	29,659.76	1	0	3.31	3.23				
SAPP	10.7	7.60	29.6	18.79	59.5	31.84	1	0	0.011	0.0076				

approach indicating that additional instances of dependency cycles are detected compared to the SA approach. In the SA approach, the search algorithm failed to detect any instance of the third cycle in all the 10 runs of the algorithm on model 3. In the SAP approach, the algorithm succeeded to detect two instances of the third cycle in model 3 in two runs—run # 3 and run # 8. In addition, the value of the recall metric associated with using the SAP approach on model 4 showed a great improvement compared to the results obtained using the SA approach due to the big reduction in the number of links associated with this model. The number of links in this model is reduced by 70.4% as shown in Table 2. Consequently, the recall jumped from 0 when the SA approach is used to 75% when the SAP approach is used.

With respect to the time needed to perform the pruning, it is included in the average time of running each approach 10 times in the last column in Table 1. In each run, the pruning time is added to the average time of running the approach 10 times. This provides the ability to compare the different approaches on the same basis (i.e., the total time needed to run each approach regardless of its components). As described earlier, the pruning mechanism is used in the SAP and SAPP approaches. The time needed to perform the pruning is added to the average time needed to run these approaches. The same thing can be said about the time needed to perform the pairing mechanism. This time is also added to the average time of running the SAPP approach.

In terms of the average number of iterations and the time needed to run the SAP approach, the improvement was more obvious. The average number of iterations needed to retrieve all the cycles in model 1 was reduced from 8341.2 in the SA approach to 2279.4 in the SAP approach (see  $\mu$  associated with cycle 5, i.e., the last cycle, in the SA and SAP rows in model 1) with a percentage of 72.67%. A similar conclusion can be drawn in model 2, the average number of iterations needed to retrieve all the cycles in this model was reduced from 21,560.2 in the SA approach to 3141.4 in the SAP approach with a percentage of 85.43%. With respect to model 3, the improvement was relatively small. The average number of iterations performed was reduced from 1,000,000 to 827,394.7, which represents a percentage of 17.26%. This is because 8 runs out of the 10 performed runs reached the maximum number of iterations without detecting the third cycle. The other two runs detected the third cycle in iteration # 252,995 and iteration # 20,952, respectively. These two runs were the reason behind the slight improvement in the number of iterations over the first approach. In model 4, despite the great improvement in the recall associated with using the SAP approach compared to the SA approach, the maximum number of iterations in each run is reached with both approaches. This is because the SA approach failed to detect any cycle and the SAP approach failed to detect any instance of the fourth cycle in all performed runs and this also

justifies why the average recall was exactly 0% and 75% (i.e.,  $\frac{3}{4}$ ), respectively.

It is also important to investigate how the average number of iterations changes between the different evaluation models using the SAP approach as it gives a glimpse of how the SA algorithm and pruning mechanism work in various situations. The average number of iterations needed to retrieve all the cycles in models 1 and 2 increased from 2279.4 to 3141.4, which represents an increase of 37.82%. This increase is reasonable given the increase in the number of links between these models. On the other hand, the average number of iterations increased from 3,141.4 to 827,394.7 with a percentage increase of 26,238.41% in model 3. To understand the reason for this increase, we investigated the average number of iterations needed to retrieve each cycle in model 3. We found that the average number of iterations to detect each cycle in model 3 using the SAP approach was 327.7, 1354.9, and 827,394.70, respectively. Given these numbers, we can see that the majority of the increase is associated with the detection of cycle 3. The reason for this increase is the length of the third cycle (i.e., 9), as shown in Table 4. The same thing applies to model 4. Most of the increase in the number of iterations was associated with cycle 4. The reason for that increase is the length of that cycle (i.e., 13). This can also justify why the SAP approach failed to detect any instance of the fourth cycle in model 4 in all of the 10 runs.

In model 5, all the links of the model participate in a single cycle. That is, the pruning mechanism is useless as no link can be removed to reduce the search space (0% in Table 2). Therefore, both SA and SAP approaches performed similarly. They performed a comparable number of iterations (The SA approach performed 132,721.3 iterations on average and the SAP approach performed 133,656.4 iterations on average). This is because the pruning mechanism which differentiates the SAP approach from the SA approach is useless.

In model 6, the SA approach performed the maximum number of iterations even though it does not contain any cycle. This is because the SA approach lacks the capability of differentiating cyclic from non-cyclic models. This is not the case with the SAP and SAPP approaches. SAP and SAPP approaches are augmented with the pruning mechanism. The pruning mechanism prunes the model until only the links that participate in cycles are left. Since there are no cycles in model 6, all links in model 6 are pruned (100% in Table 2). Consequently, these two approaches (i.e., SAP and SAPP) do not need to perform any iteration at all as no links are left to search for cycles.

In model 7, the reduction in the number of links due to the application of the pruning mechanism was very significant (i.e., 93.9%), as shown in Table 6. This reduction led to significant improvement in the obtained results. With the SA approach, none of the cycles was detected, but with the SAP



approach, all cycles were detected in an average of 30,642.9 iterations and 3.31 s.

This approach alleviates the problem of model size. However, it shows weakness when it comes to long cycles, as shown in models 3 and 4. This problem is alleviated by augmenting the simulated annealing search algorithm with the pairing mechanism, as described in Algorithm 3 (see Sect. 0), resulting in developing a new approach (i.e., SAPP).

### 7.3 Third approach: SAPP

This approach is augmented with the pairing mechanism to help the simulated annealing search algorithm cope with lengthy cycles. The results of evaluating the SAPP approach are presented in Table 5. We can see that the obtained results improved substantially compared to the previous approaches. In terms of the recall metric, the SAPP approach was able to achieve 100% with all the evaluation models (i.e., all the cycles in each evaluation model were detected in each run). In terms of the average number of iterations and average time, the SAPP approach performed much better than the SAP approach. The average number of iterations performed to detect all the cycles in model 1 is reduced from 2279.4 using the SAP approach to 16.9 using the SAPP approach with a reduction rate of 99.26%. This huge reduction proves the effectiveness of the pairing mechanism. The same results are obtained with the other models as well. In model 2, the average number of iterations is reduced from 3141.4 using the SAP approach to 19.4 using the SAPP approach with a reduction rate of 99.38%. In model 3, the average number of iterations is reduced from 116,021.5 using the SAP approach to 551.9 using the SAPP approach with a reduction rate of 99.52%. In model 4, the SAP approach was unable to detect any instance of the fourth cycle in all the runs. This is not the case with the SAPP approach, the SAPP approach was able to detect all the cycles in model 4 by performing an average of 84,004.3 iterations. In model 5, the SA and SAP approaches performed a large number of iterations to be able to detect the only cycle in that model. This is different from the obtained results using the SAPP approach in which the pairing mechanism is utilized to improve the search process. Consequently, the average number of iterations is reduced significantly from 133,656.4 using the SAP approach to 132.3 using the SAPP approach with a reduction rate of 99.9%. Finally, in model 7, the SAPP approach was able to perform better than the SAP approach in terms of the average number of iterations, and, consequently, in terms of the average time needed to detect all the cycles. The average number of iterations needed to detect all the cycles was reduced from 30,642.9 using the SAP approach to 59.5 using the SAPP approach. This is also reflected in the reduction in the average time needed to detect

all the cycles from 3.31 s using the SAP approach to 0.011 s using the SAPP approach.

The average number of iterations needed to detect all the cycles using the SAPP approach in models 1 and 2 is 16.9 and 19.4, respectively, with an increase rate of 14.79% only. This increase is expected given the increase in the size of model 2 over model 1, as shown in Table 4. On the other hand, the average number of iterations needed to detect all the cycles in model 3 is 835.1 with an increase rate of 4204.64% compared to model 2. This increase, in part, is due to the increase in the size of model 3 over model 2; however, the major reason behind this big increase comes from the length of cycles. The length of cycles in model 3 is 4, 4, and 9, as shown in Table 4. This is not the case with model 2, in which the longest cycle is of length 6. To confirm this finding, we examined the average number of iterations performed to detect each cycle in model 3. As shown in Table 5, the average number of iterations performed to detect each of the three cycles in model 3 is 78.4, 209.4, and 835.1 respectively. We can see that most of the increase is associated with detecting the third cycle (i.e., the longest cycle). The same behavior was observed in the results associated with model 4. The average number of iterations performed to detect all the cycles is 84,004.3 with an increase rate of 9959.19% compared to model 3. This increase, in most, is due to the length of cycles in model 4 in which the length of the longest cycle is 13, as shown in Table 5. To confirm this finding, we examined the average number of iterations performed to detect each cycle in model 4. As shown in Table 5, the average number of iterations performed to detect each of the four cycles in model 4 is 194.3, 356.3, 1,001.8, and 84,004.3, respectively. We can see that most of the increase in the number of iterations is associated with detecting the fourth cycle (i.e., the longest cycle).

## 8 Discussion

In the previous sections, the circular dependency bad smell is introduced, and the detection technique is described and evaluated. In this section, we discuss two main concerns related to this research. First, in this work, the developed approaches are evaluated on experimental models in which the number of cycles is known. However, this is not the case in real-world settings. In real-world settings, usually, the number of cycles is unknown in advance. Second, we discuss the threats to the validity of this work that might affect the obtained results and conclusion.

### 8.1 Application guide

The simulated annealing algorithm augmented with the pruning and pairing mechanisms (i.e., SAPP) is proved to

be the most effective approach among the three proposed approaches, as shown in Sect. 2. However, to be able to apply the SAPP approach in real-world settings, the number of iterations parameter needs to be set. This number is set to 1000,000 in the conducted experiments, just to allow us to study the behavior of the different approaches under the different situations. This is different from real-world settings. In real-world settings, this parameter needs to be set such that it helps guide the search process in finding all the cycles in the addressed model. Hence, how can this parameter be set? There are several ways that are different from each other in their effectiveness. Firstly, the number of iterations can be set by the trial-and-error method as it is a common method in setting parameters in search-based software engineering [76, 77]. To improve the effectiveness of this method, the trial-and-error method can be combined with the model's size. However, it turns out that this technique is ineffective due to the high variability in the number of iterations conducted to retrieve the different cycles in the different models as shown by the obtained standard deviations ( $\sigma$ ), as shown in Table 5. This high variability makes this technique less effective in setting the number of iterations parameter in our case.

Secondly, another way of setting the number of iterations parameter is to link it to the number of cycles in the addressed model. That is, SAPP will keep looking for cycles until finding all cycles in the model. The problem is that the number of cycles in the model is usually unknown in advance. Therefore, the modeler needs to assume the number of cycles using the trial-and-error method. If the number of existing cycles in the model is less than the assumed number, the SAPP approach will keep searching forever. If the number of exiting cycles in the model is more than the assumed number, the SAPP approach will falsely declare the absence of cycles in the addressed model. We can see that this technique is ineffective, as well, as the actual number of the cycles in a model is missing.

Since the techniques presented above are not effective in setting a stopping condition, how can we specify the number of iterations that can be performed until finding all the cycles in a model, if any? We found that we can circumvent this issue using the pruning mechanism. We observed that if the model is non-cyclic, it can be fully pruned (see the outcomes of pruning model 6, as shown in Table 6). Based on that we established the following lemma.

**Lemma 1.** *A non-cyclic GRL model is a fully prunable model.*

This lemma can be utilized in various ways to help identify a stopping condition to run the needed number of iterations until detecting all the exiting cycles in a model. One possible way is shown in Algorithm 4. It starts by running the pruning algorithm on the subject model. If the subject model is not cyclic, it will be fully pruned. Otherwise, if the model is not fully pruned, it indicates the presence of cycles. In

**Table 6** The percentage of reduction in each evaluation model after applying the pruning mechanism

Model	No. of links before pruning	No. of links after pruning	Reduction (%)
Model 1	15	12	20%
Model 2	18	13	27.7%
Model 3	28	18	35.7%
Model 4	71	21	70.4%
Model 5	9	9	0%
Model 6	67	0	100%
Model 7	248	15	93.9%

this case, the detection approach (i.e., SAPP) will search for cycles until finding a cycle. Resolving this cycle will allow for more links to be pruned. This procedure will be repeated until detecting all the cycles in the subject model and the model is fully pruned.

Algorithm 4: The application procedure

```

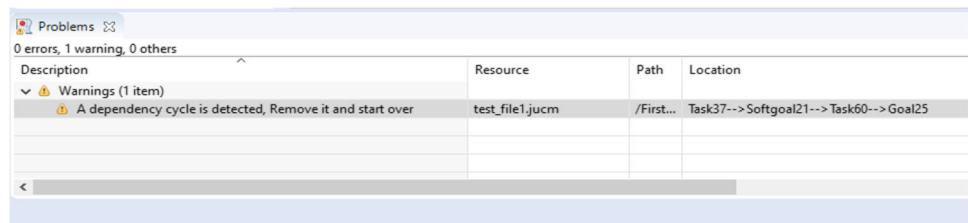
1: Input: PM: Pruned model
2: Process:
3: While the model is not fully pruned Do
4:   Find a cycle in PM
5:   Resolve the cycle
6:   Prune PM
7: End While
8: Output: All cycles in the model

```

This method can guarantee the detection of all cycles in the subject model. It allows for more links to be pruned each time a cycle is detected and resolved. This continues until all links are pruned. It is worth pointing out that this method does neither need to specify the number of cycles in a model nor specify a maximum number of iterations, sparing whoever wants to apply the SAPP approach from setting parameters.

To clarify how this procedure works, this approach is applied to model 4 (validation data, see Table 4). The application of this procedure on model 4 resulted in pruning the model links from 71 to 21 and detecting the first cycle in the first round. Once the first cycle is resolved, this method is applied for the second round and resulted in pruning the links from 21 to 18 and detecting the second cycle. Once the second cycle is resolved, the procedure is applied for the third round and resulted in pruning the links from 18 to 15 and detecting the third cycle. Once the third cycle is resolved, the method is applied for the fourth round and resulted in pruning the links from 15 to 13 and detecting the fourth cycle. This cycle is resolved, and this method is applied for the fifth round. This time the application of this method resulted in pruning all the remaining links in the model indicating that all the cycles in the model are detected.

**Fig. 13** The results as seen by end-users of the developed tool



```

Started
15:23:20
Recall: 100.0
Detected cycles::[
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 58, 69, 80, 82],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 68, 69, 71, 72],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 54, 55, 56, 58],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 49, 54, 58, 68, 69, 82],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 48, 58, 68, 69, 82]]
15:23:21
Finished
    
```

**Fig. 12** The results at the time of evaluation

Algorithm 4 is what the developed Eclipse plugin is designed to do. For experimental purposes (i.e., to evaluate the developed techniques), though, we designed the developed approaches to return a list of all the identified cycles, as shown in Fig. 12. We believe this is not effective for final users. Therefore, when we designed the tool (see Sect. 0) that is intended to be used by the end users, we detect one cycle each time and ask the user to resolve it, as shown in Fig. 13. This is done because on many occasions, resolving one cycle might lead to resolving another cycle, and on other occasions, resolving a cycle might bring another cycle. Therefore, following lemma 1, we believe it will be better to see the outcomes of resolving cycles one by one until the model is free of cycles.

This is different from the experiments used to evaluate the proposed approaches, as shown in Table 5. In these experiments, when a cycle is detected, it does not get resolved. The developed approaches are designed to proceed to find the next cycle until all cycles are detected, or the maximum number of iterations is reached (i.e., the stopping conditions). This strategy is adopted to provide an objective homogeneous evaluation. The gradual detection and resolution cycle by cycle strategy has two issues that might result in less objective evaluation. Firstly, there are several ways to resolve the detected cycle. Each way might have a different impact on the rest of the model; it might lead to an increase or decrease in the number of links, it might lead to removing or adding new elements, it might lead to merging elements, etc. Secondly, resolving a cycle might result in resolving more than one cycle. It might also result in resolving one cycle and adding a new cycle.

## 8.2 Threats to validity

The proposed approaches and the empirical validation are subject to several threats to validity that have been identified and categorized according to the classification provided in [78], as follows:

**Construct threats:** There is a possible threat related to some of the goal models used to evaluate the proposed approaches in this study. Some of the validation goal models are created for experimental purposes. To overcome this issue, we made sure that these experimental models cover several configurations of cycles and models. Besides, we augmented our evaluation with a real-world case study.

**Internal threats:** Internal threats to validity concern the degree to which a certain outcome depends on the intended experimental variables. In this paper, the developed approaches are implemented and tested to the best of our knowledge. However, there might be some implementation and code tweaks that work in favor or against the developed technique. Other implementations might also result in differences in the obtained results. To overcome this situation, the code used for implementing this technique is posted online (see Sect. 0) so that others can use it for comparison with our work.

**External threats:** External threats concern the ability to generalize the obtained results. The experiments in this study are conducted on GRL goal models built using the jUCM-Nav Eclipse plugin [13]. To generalize the obtained results to other goal models, such as i\*, the developed approaches may require some adaptations. However, we believe that such adaptations would be minimal, if any. Circular dependencies are structurally the same in the different goal modeling languages and frameworks (i.e., a propagation path that starts and ends at the element). Moreover, goal models created by the different goal modeling languages and frameworks can be represented alike (see Sect. 0). Hence, the generalization of our approach to other goal-oriented languages would not require significant efforts.

**Conclusion threats:** Conclusion validity concerns the degree to which the conclusions made in the study are reasonable. Several conclusions are made with respect to

the performance of the three proposed approaches. Since metaheuristic search algorithms have a built-in randomness component, this randomness can result in one approach performing better than the other in one single run despite that it is not a better approach in general. For example, the SAP approach might perform better than SAPP approach in a single run although, in general, SAPP performs much better than SAP. Therefore, to mitigate this threat, the conclusions made in this study are based on the average of running each approach on each model for 10 runs. Another concern that may threaten the conclusion validity of this study is the time needed to run the different approaches. The time needed to run the different approaches is obtained based on our machine. The time is dependent on the machine running the developed approaches and the other applications running on the same machine. To mitigate this threat, this time is accompanied by the number of iterations. We believe that combing these two measures creates a better understanding of the time needed to run the developed approaches.

## 9 Conclusion

In this paper, we introduced and defined the circular dependency bad smell in GRL goal models. Circular dependencies occur in the context of the satisfaction analysis of these models. Satisfaction analysis starts by setting evaluation values to some elements of the model (i.e., a strategy). Then, these evaluation values are propagated throughout the model by a propagation algorithm to calculate the evaluation values of the other elements in the model. When the path of the propagation forms a cycle, it creates a circular dependency. The impact of circular dependencies on the propagation algorithm, and satisfaction analysis, depends on the design of the algorithm. In forward propagation algorithms (designed as suggested in the standard), circular dependencies block the propagation of evaluation values of cycle elements hindering satisfaction analysis. Beyond satisfaction analysis, circular dependencies can cause problems in reusing the model or in using the model in conjunction with the other models such as UCM and feature models. Thus, to help eliminate these problems, first, we need to detect these cycles.

The manual detection of dependency cycles is difficult, if not impossible. The complexity of detecting these cycles is exponential with the number of cycles and the number of cycles grows exponentially with the number of elements in the model. Hence, to provide a scalable technique, the simulated annealing search algorithm is employed to detect dependency cycles. To enhance the performance of the simulated annealing search algorithm, the structural properties of GRL models are studied and two heuristics are developed. These heuristics are integrated into the developed approach

as a pruning mechanism and as a pairing mechanism. Based on that, three search approaches, namely simulated annealing algorithm (SA), simulated annealing augmented with pruning (SAP), and simulated annealing augmented with pruning and pairing (SAPP), were proposed. Besides, we provided an application procedure based on the pruning mechanism to help modelers apply the outcomes of this study effectively.

The three approaches were evaluated on a set of seven experimental models of different numbers of elements, number of links, number of cycles, and length of cycles. The obtained results show that the simulated annealing algorithm augmented with pruning and pairing mechanisms (SAPP) is the most effective in finding cycles as compared to SA and SAP.

As future work, we plan to generalize our approach to cover many goal-oriented modeling languages such as  $i^*$ . In addition, we plan to develop a refactoring recommendation technique to suggest resolutions to the detected cycles to help modelers resolve dependency cycles. Moreover, we are planning to add new features to the developed tool such as presenting all the detected cycles as a list.

**Acknowledgements** The authors acknowledge the support of King Fahd University of Petroleum and Minerals in the development of this work.

## References

1. Lamsweerde A.v.: Goal-oriented requirements engineering: a guided tour. In: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering pp. 249–262 (2001)
2. Lapouchnian A.: Goal-oriented requirements engineering: an overview of the current research. University of Toronto, vol. 32, (2005)
3. Mylopoulos, J., Chung, L., Nixon, B.: Representing and using non-functional requirements: a process-oriented approach. *IEEE Trans. Software Eng.* **18**(6), 483–497 (1992)
4. Chung, L., Nixon B.A., Yu, E.J: Mylopoulos: Non-functional requirements in software engineering. Springer Science & Business Media (2012)
5. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. *Sci. Comput. Program.* **20**(1–2), 3–50 (1993)
6. Van Lamsweerde, A., Letier: From object orientation to goal orientation: A paradigm shift for requirements engineering," in *Radical Innovations of Software and Systems Engineering in the Future*: Springer, pp. 325–340 (2004)
7. Yu, E.: Towards modelling and reasoning support for early-phase requirements engineering. In: Proceedings of the Third IEEE International Symposium on Requirements Engineering, pp. 226–235: IEEE (1997)
8. ITU-T, Z: 151 User requirements notation (URN)—Language definition. ITU-T, (2018)
9. Horkoff J. et al: Goal-oriented requirements engineering: a systematic literature Map. In: IEEE 24th International Requirements Engineering Conference (RE), pp. 106–115 (2016)

10. Pacheco, C., Garcia, I.: A systematic literature review of stakeholder identification methods in requirements elicitation. *J. Syst. Softw.* **85**(9), 2171–2181 (2012)
11. Amyot, D., Ghanavati, S., Horkoff, J., Mussbacher, G., Peyton, L., Yu, E.: Evaluating goal models within the goal-oriented requirement language. *Int. J. Intell. Syst.* **25**(8), 841–877 (2010)
12. Amyot, D., Mussbacher, G., Ghanavati, S., Kealey, J.: GRL Modeling and Analysis with jUCMNav. *iStar*, 766, pp. 160–162 (2011)
13. Fowler, M.: Refactoring: Improving the Design of Existing Code 2nd ed. Addison-Wesley Signature Series (Fowler), (2018)
14. Sjøberg, D.I., Yamashita, A., Anda, B.C., Mockus, A., Dybå, T.: Quantifying the effect of code smells on maintenance effort. *IEEE Trans. Softw. Eng.* **39**(8), 1144–1156 (2013)
15. Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A.: On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empir. Softw. Eng.* **23**(3), 1188–1221 (2018)
16. Zazworka, N., Shaw, M. A., Shull, F., Seaman C.: Investigating the impact of design debt on software quality. In: Proceedings of the 2nd Workshop on Managing Technical Debt, (2011), pp. 17–23: ACM.
17. Alshayeb, M.: Empirical investigation of refactoring effect on software quality. *Inf. Softw. Technol.* **51**(9), 1319–1326 (2009)
18. Arendt, T., Taentzer, G.: UML model smells and model refactorings in early software development phases. Universität Marburg, (2010)
19. Horkoff, J., Eric, S.: A Qualitative, Interactive Evaluation Procedure for Goal-and Agent-Oriented Models. In: CAiSE Forum, (2009)
20. Duran, M.B., Mussbacher, G.: Investigation of feature run-time conflicts on goal model-based reuse. *Inf. Syst. Front.* **18**(5), 855–875 (2016)
21. Tinnes, C., Biesdorf, A., Hohenstein, U., Matthes, F.: Ideas on improving software artifact reuse via traceability and self-awareness. In: IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST), pp. 13–16: IEEE (2019)
22. Duran, M.B., Mussbacher, G., Thimmegowda, N., Kienzle, J.: On the reuse of goal models. In: International SDL Forum, pp. 141–158: Springer (2015)
23. Sharma, T., Spinellis, D.: A survey on software smells. *J. Syst. Softw.* **138**, 158–173 (2018)
24. Misbhauddin, M., Alshayeb, M.: UML model refactoring: a systematic literature review. *Empir. Softw. Eng.* **20**(1), 206–251 (2015)
25. Azeem, M.I., Palomba, F., Shi, L., Wang, Q.: Machine learning techniques for code smell detection: a systematic literature review and meta-analysis. *Information and Software Technology*, (2019)
26. Mariani, T., Vergilio, S.R.: A systematic review on search-based refactoring. *Inf. Softw. Technol.* **83**, 14–34 (2017)
27. Alkharabsheh, K., Crespo, Y., Manso, E., Taboada, J. A.: Software design smell detection: a systematic mapping study. *Software Quality Journal*, pp. 1–80, (2018)
28. Bertran, I.M.: Detecting architecturally-relevant code smells in evolving software systems. In: 33rd International Conference on Software Engineering (ICSE), pp. 1090–1093: IEEE (2011)
29. Dexun, J., Peijun, M., Xiaohong, S., Tiantian, W.: Detecting bad smells with weight based distance metrics theory. In: Second International Conference on Instrumentation, Measurement, Computer, Communication and Control, pp. 299–304: IEEE (2012)
30. Nongpong K.: Feature envy factor: a metric for automatic feature envy detection. In: 7th International Conference on Knowledge and Smart Technology (KST), pp. 7–12: IEEE (2015)
31. Fourati, R., Bouassida, N., Abdallah, H. B.: A metric-based approach for anti-pattern detection in uml designs. In: Computer and Information Science: Springer, pp. 17–33 (2011)
32. Singh, S., Kahlon, K.: Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells. *ACM SIGSOFT Softw. Eng. Notes* **36**, 1–10 (2011)
33. Tahvildar, L., Kontogiannis, K.: A metric-based approach to enhance design quality through meta-pattern transformations," in Seventh European Conference on Software Maintenance and Reengineering, pp. 183–192: IEEE (2003)
34. Chen, Z., Chen, L., Ma, W., Zhou, X., Zhou, Y., Xu, B.: Understanding metric-based detectable smells in Python software: a comparative study. *Inf. Softw. Technol.* **94**, 14–29 (2018)
35. Velioglu, S., Selçuk, Y.E.: An automated code smell and anti-pattern detection approach: In: IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA), pp. 271–275: IEEE (2017)
36. Czibula, G., Marian, Z., Czibula, I.G.: Detecting software design defects using relational association rule mining. *Knowl. Inf. Syst.* **42**(3), 545–577 (2015)
37. Lee, S.-J., Lo, L.H., Chen, Y.-C., Shen, S.-M.: Co-changing code volume prediction through association rule mining and linear regression model. *Expert Syst. Appl.* **45**, 185–194 (2016)
38. Kessentini, M., Sahraoui, H., Boukadoum, M., Wimmer, M.: Design defect detection rules generation: a music metaphor. In: 15th European Conference on Software Maintenance and Reengineering, pp. 241–248: IEEE (2011)
39. Lee, K.S., Geem, Z.W.: A new meta-heuristic algorithm for continuous engineering optimization: harmony search theory and practice. *Comput. Methods Appl. Mech. Eng.* **194**(36–38), 3902–3933 (2005)
40. Maddeh, M., Ayouni, S.: Extracting and Modeling Design Defects Using Gradual Rules and UML Profile. In: IFIP International Conference on Computer Science and its Applications, pp. 574–583: Springer (2015)
41. Di-Jorio, L., Laurent, A., Teisseire, M.: Mining frequent gradual itemsets from large databases. In: International Symposium on Intelligent Data Analysis, pp. 297–308: Springer (2009)
42. Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A.: Lightweight detection of Android-specific code smells: The aDoctor project. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 487–491: IEEE (2017)
43. Fontana, F.A., Mäntylä, M.V., Zanoni, M., Marino, A.: Comparing and experimenting machine learning techniques for code smell detection. *Empir. Softw. Eng.* **21**(3), 1143–1191 (2016)
44. Hozano, M., Antunes, N., Fonseca, B., Costa, E.: Evaluating the accuracy of machine learning algorithms on detecting code smells for different developers. In: ICEIS (2), pp. 474–482 (2017)
45. Maneerat, N., Muenchaisri, P.: Bad-smell prediction from software design model using machine learning techniques. In: Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 331–336: IEEE (2011)
46. Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Gueheneuc, Y.-G., Aimeur, E.: SMURF: A SVM-based incremental anti-pattern detection approach. In: 19th Working Conference on Reverse Engineering, pp. 466–475: IEEE (2012)
47. Maiga, A. et al.: Support vector machines for anti-pattern detection. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 278–281: IEEE (2012)
48. Hassaine, S., Khomh, F., Guéhéneuc, Y.-G., Hamel, S.: IDS: An immune-inspired approach for the detection of software design smells," in Seventh International Conference on the Quality of Information and Communications Technology, pp. 343–348: IEEE (2010)
49. Luke, S.: Essentials of metaheuristics. ed: Springer, (2011)
50. Kessentini, W., Kessentini, M., Sahraoui, H., Bechikh, S., Ouni, A.: A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Trans. Softw. Eng.* **40**(9), 841–861 (2014)

51. Ouni, A., Kessentini, M., Inoue, K., Cinnéide, M.O.: Search-based web service antipatterns detection. *IEEE Trans. Serv. Comput.* **10**(4), 603–617 (2017)
52. Boussaa, M., Kessentini, W., Kessentini, M., Bechikh, S., Chikha, S.B.: Competitive coevolutionary code-smells detection. In: *International Symposium on Search Based Software Engineering*, pp. 50–65: Springer (2013)
53. Ghannem, A., Kessentini, M., El Boussaidi, G.: Detecting model refactoring opportunities using heuristic search," In: *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 175–187: IBM Corp (2011)
54. Asano, K., Hayashi, S., Saeki, M.: Detecting bad smells of refinement in goal-oriented requirements analysis. In: *International Conference on Conceptual Modeling*, pp. 122–132: Springer (2017)
55. Yan, J.B.: *Static Semantics Checking Tool for jUCMNav*. ed: Master's project, SITE, University of Ottawa, (2008)
56. E. Knauss, El Boustani, C., Flohr, T.: Investigating the impact of software requirements specification quality on project success. In: *International Conference on Product-Focused Software Process Improvement*, pp. 28–42: Springer (2009)
57. Mussbacher, G., Amyot, D., Heymans, P.: Eight Deadly Sins of GRL. In: *iStar*, pp. 2–7 (2011)
58. Johnson, D.B.: Finding all the elementary circuits of a directed graph. *SIAM J. Comput.* **4**(1), 77–84 (1975)
59. Horkoff, J., Yu, E.: Comparison and evaluation of goal-oriented satisfaction analysis techniques. *Requirements Eng.* **18**(3), 199–222 (2013)
60. Yu, E.: *Modelling strategic relationships for process reengineering (Social Modeling for Requirements Engineering)*. p. 2011 (2011)
61. Jureta, I.J., Faulkner, S.: Clarifying goal models. In: *26th international conference on Conceptual modeling*, vol. 28, pp. 139–144 (2007)
62. Santander, V.F., Castro, J. F.: Deriving use cases from organizational modeling. In: *Proceedings IEEE joint international conference on requirements engineering*, pp. 32–39: IEEE (2002)
63. Jureta, I.J., Faulkner, S., Schobbens, P.-Y.: Clear justification of modeling decisions for goal-oriented requirements engineering. *Requirements Eng.* **13**(2), 87 (2008)
64. Rayasam, S.: *Transformational creativity in requirements goal models*. University of Cincinnati, (2016)
65. Omer, A.M., Schill, A.: Automatic management of cyclic dependency among web services. In: *14th IEEE International Conference on Computational Science and Engineering*, pp. 44–51: IEEE (2011)
66. Bitonti, T.F., Lei, Y.: *Methods, systems, and computer program products for using graphs to solve circular dependency in object persistence*. ed: Google Patents, (2009)
67. Zhong, E.: *Methods and systems for determining circular dependency*. ed: Google Patents, (2007)
68. Melton, H., Tempero, E.: An empirical study of cycles among classes in Java. *Empir. Softw. Eng.* **12**(4), 389–415 (2007)
69. Hassine, J., Alshayeb, M.: Measurement of actor external dependencies in GRL Models. In: *Proceedings of the Seventh International i\* Workshop co-located with the 26th International Conference on Advanced Information Systems Engineering (CAiSE 2014)*, Thessaloniki, Greece, (2014)
70. Yu, E., Giorgini, P., Maiden, N., Mylopoulos, J., Fickas, S.: *Strengths and weaknesses of the i\* framework: an empirical evaluation (Social Modeling for Requirements Engineering)*. MIT Press, (2011)
71. Lima, P., et al.: An extended systematic mapping study about the scalability of i\* Models. *CLEI Electron. J.* **19**(3), 7–7 (2016)
72. Yu, E., Giorgini, P., Maiden, N., Mylopoulos, J., Fickas, S.: *Strengths and Weaknesses of the i\* Framework: An Empirical Evaluation*. MIT Press, *Social Modeling for Requirements Engineering* (2011)
73. Abdel-Basset, M., Abdel-Fatah, L., Sangaiah, A.K.: *Metaheuristic algorithms: a comprehensive review. Computational intelligence for multimedia big data on the cloud with engineering applications*, pp. 185–231, (2018)
74. Almhana, R., Kessentini, M.: Considering dependencies between bug reports to improve bugs triage. *Autom. Softw. Eng.* **28**(1), 1–26 (2021)
75. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. *science*, vol. 220, no. 4598, pp. 671–680, (1983)
76. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: *33rd International Conference on Software Engineering (ICSE)*, pp. 1–10: IEEE (2011)
77. Eiben, A.E., Smit, S.K.: Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm Evol. Comput.* **1**(1), 19–31 (2011)
78. Campbell, D.T., Cook, T.D.: *Quasi-experimentation: Design & analysis issues for field settings*. Rand McNally College Publishing Company Chicago, (1979)

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Mawal A. Mohammed** received his PhD and MSc from the department of Information and Computer science of King Fahd University of Petroleum and Minerals (KFUPM). Prior to this, Mr. Mawal worked as a full-time teaching assistant in the Software Engineering department of Taiz University, Yemen, where he received his BS degree. His research interests include software engineering, requirements engineering, software design patterns, and software quality.



**Mohammad Alshayeb** is a Professor at the Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Saudi Arabia. He received his MS and PhD in Computer Science and certificate of Software Engineering from the University of Alabama in Huntsville in 2000, 2002 and 1999 respectively. Dr. Alshayeb worked as a senior researcher and Software Engineer and managed software projects in the United States and the Middle

East. He published over 100 refereed conference and journal publications and holds 5 US patents. He received Khalifa award for education as "the distinguished University Professor in the Field of Teaching within Arab World", 2016. Dr. Alshayeb's research interests include software quality and quality improvements, software measurement and metrics, evidence-based software engineering and empirical studies in Software Engineering. More



**Jameleddine Hassine** is an Associate Professor at the department of Information and Computer Science of King Fahd University of Petroleum and Minerals (KFUPM). Dr. Hassine holds a Ph.D. from Concordia University, Canada (2008) and an M.Sc. from the University of Ottawa, Canada (2001). Dr. Hassine has several years of industrial experience within worldwide telecommunication companies; Nortel Networks (Canada) and Cisco Systems (Canada). His

main research interests include requirements engineering (languages and methods), software testing, formal methods, software evaluation, and maintenance. He is actively involved in several funded research projects and has over 50 publications on various research topics in his field. Dr. Hassine published his research in many high-impact journals like Requirements Engineering Journal (REJ), Journal of Systems and Software (JSS), Information and Software Technology (IST), and Software and Systems Modeling (SoSyM).