On the Selection of Concurrency Control Policies for Configuration Management

João Gustavo Prudêncio, Cláudia Werner
COPPE, System Eng. and Computer Science
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil
\{gustavo,werner\}@cos.ufrj.br

Leonardo Murta
Instituto de Computação
Universidade Federal Fluminense
Rio de Janeiro, Brazil
leomurta@ic.uff.br

Abstract—Currently, the number of developers involved in a software development project is increasing because of the need to deliver systems with higher complexity and quality and to reduce time-to-market. In order to have the software development process executing in an organized way, we must provide mechanisms to control concurrent access over the project artifacts. These mechanisms are implemented by concurrency control policies in version control systems, which may allow (optimistic policy) or inhibit (pessimistic policy) parallel development. This work presents the Orion approach, which analyzes the project historical changes and selects the most appropriate concurrency control policy for each software element. In addition, it identifies critical elements, which are candidates to refactoring. This selection aims at minimizing conflict situations, and thus improving the development team productivity. A prototype was built to enable the application of the proposed approach and two experimental studies were performed as a preliminary evaluation.

Concurrency Control; Software Configuration Management; Version Control

I. INTRODUCTION

The software development process is an activity that involves, most of the time, more than one person. These people, who are part of a team, collaborate together in order to achieve the same goal: build software. Moreover, the number of developers who participate in this team is increasing. This occurs because organizations need to deliver systems with higher complexity and quality and need to reduce time-to-market. Other important reasons for adopting larger teams are market pressure and innovation needs [1]. Reasons like those contribute for the increasing of software development industry and stimulate organizations to hire more developers, thus, culminating in more concurrent work [2].

Configuration Management (CM) [2-4] systems, in special version control systems offer an infrastructure that supports coordination of the development team members [1]. These systems provide functionalities such as: a central repository for the development project artifacts storage; access and modification control of these artifacts by developers; and a mechanism to lock, compare and merge different versions of the same artifact. In this way, using some version control system, developers can concurrently work in the same project, manipulating the same artifacts.

Sometimes, developer’s tasks are not independent from others, which implies that modifications made in an artifact can impact in related artifacts. Moreover, occasionally, it is necessary that more than one developer needs to work over the same artifact at the same time. Due to that, conflicts may raise during merge. In general, these conflicts are solved by the involved developers, comparing the artifacts versions and modifying these versions, in order to build the final versions of the artifacts. Some of these conflicts are simple and quick to be solved, but depending on the modifications size, the complexity of the modified artifact, and the developer knowledge about the artifact, a lot of time and a great effort can be necessary to solve them. Time and effort can be considered to be some sort of additional task or rework, which would not be necessary if the conflicts had not occurred [5, 6]. Thus, conflicts can negativity impact on development team productivity. Mechanisms for concurrency control are implemented by concurrency control policies used in version control systems, which may allow (optimistic policy) or inhibit (pessimistic policy) parallel development.

This work presents the Orion approach, which analyzes historical changes of a project and selects the most appropriate concurrency control policy for each project element. In addition, it identifies critical elements that must receive some kind of refactoring. The approach aims at allowing a better utilization of developer’s time and effort, minimizing conflicts occurrence and impact, and thus improving the development team productivity.

The rest of this paper is organized as follows. Section II discusses about some important concurrency control concepts for this work. Section III presents the proposed approach. Section IV details the studies that were made in order to evaluate the proposed approach. Section V describes a prototype which implements the approach. Section VI presents some related work, and we conclude the paper in Section VII with an outlook at our future work.

II. BACKGROUND

Concurrency control is the activity of coordinating actions of processes that operate in parallel, access shared data, and thus potentially interfere with each other [7]. In computer science, many areas have already dealt with the concurrency control issue, as for instance, operational systems, parallel computing, and databases [4].
In the software development context, scope of this paper, concurrency control is applied in order to manage software engineers that work at the same time over the same set of artifacts. In a Database Management Systems scenario, for example, the concurrency control issue is handled by means of transactions. However, in databases an ordinary transaction lasts a fraction of a second and involves few objects [8]. On the other hand, in the software development scenario, a set of modifications made by a developer can last hours, days or more, depending on the project complexity, development team work style, developer experience, and so on. Besides, the modification scope can be large, including many objects. Due to that, the serialization approach used in databases, in which modifications are performed in sequence, is not always possible in the software development context [8]. It is necessary to provide a different solution for the concurrency control issue in the software development area, when compared to other computer science areas.

CM is a discipline to evolve software in an organized way [3]. Therefore, CM contributes to satisfy quality and delay constrains [2]. CM systems were designed to support concurrent software development that involves multiple software engineers. Moreover, these systems enable detailed management of the modifications that have been applied on software during its entire life cycle [9].

In this paper, we will focus on the version control system, a CM system that makes it possible to identify and control the distributed evolution of software artifacts. This enables the implementation of different modification requests in parallel by the development team.

Existent version control systems solutions provide an infrastructure for the definition of which artifacts will be under CM, named as Configuration Items (CIs). These items represent hardware or software aggregation, identified by CM as a unique element [10]. Besides, some others operations are provided by these solutions. One of them is check-out, which represents the process of copying CIs from the repository to the software engineer’s workspace [11]. Each software engineer has an isolated workspace where modifications can be made. After making modifications, engineers can save their CI versions in the repository using check-in operation. When a check-in is performed, conflicts may raise. It means that the CIs that are being checked-in were modified in parallel by other developers since the last synchronization with repository. These conflicts happen when there are overlapping modifications in the same CIs.

A. Concurrency control policies

Generally, version control system solutions support the adoption of different concurrency control policies for each CI, which may allow or inhibit parallel development. Basically, there are two classes of policy: pessimistic and optimistic [12].

Pessimistic policy adopts the use of reserved check-out, which locks the CI and inhibits concurrent development. By applying this policy, conflicts are avoided. Thus, developers will not spend time and effort in order to solve conflicts. Also, locks can assist developers regarding communication, because they know when another developer is modifying the same CI and it should instigate the communication among developers with the same interests.

In contrast, pessimistic policy has some disadvantages. First, locks can cause an unnecessary serialization over development process. Sometimes different developers can modify different independent parts of the same CI, but using a lock-based approach that is not possible. In addition, locks can cause a false security sensation [13]. Developers may think that their work will not be affected by other developers’ work due to the locks. However, sometimes a CI depends on other CIs, which can be modified by other developers. This situation is considered as an indirect conflict [12].

On the other hand, optimistic policy allows multiple developers working over the same CIs at the same time. However, if conflicts occur, they are solved by merging the contributions of the concurrent developers, generating another CI version.

Optimistic policy has also some disadvantages. First, developers will have to dedicate some of their time in order to solve conflicts, because sometimes these conflicts can only be solved manually. Besides, in order to know how to solve conflicts in a correct way, developers need to be in touch with each other. They need to decide how to combine the contributions and build a new version of the CIs. In order to do that, developers must be available at the moment of solving conflicts.

According to Redmiles et al. [14], the higher the concurrent work, the lower the code quality. Besides, developers consider conflicts as a bad thing because of the need of additional efforts to solve them. Due to that, developers will try to perform their modifications quickly and check CIs in, in order to avoid conflicts and consequently avoid reviewing and testing CIs after merge activity.

On the other hand, we discussed some reasons to adopt concurrent work in the beginning of this paper, for instance, to reduce time-to-market. Therefore, it is important to carefully analyze if parallelism should be allowed or inhibited on the development process by selecting suitable concurrency control policies. There are situations that one policy is more indicated than another. For example, in cases where merging CI versions is complex, as for example, for non-textual CIs, it is advised to use lock-based approach. On the other hand, in cases where the merge activity is not so complex, the merge-based approach is recommended [8].

The conflicts frequency can also be used to select concurrency control policies. It depends on the number of developers who used to work in parallel over the same CI. Estublier [8] presented a real development scenario of a large project that adopts the optimistic approach. The analyzed project has 4 million lines of code, developed by around 800 software engineers. In this context, 3 engineers used to modify the same artifact concurrently, and there were situations that 30 engineers concurrently modified the same CI. In cases like that, surely the number of conflicts increases, and consequently more developer time and effort will be necessary in order to solve them.

156
The pessimistic and optimistic policies support varies for different version control systems solutions. Some version control systems like RCS [15] and Visual SourceSafe [16], for instance, preferentially adopt the pessimistic policy. On the other hand, solutions like CVS [17], for example, basically adopt the optimistic policy. However, modern version control systems, such as Subversion [13] and Rational ClearCase [18], support both kinds of concurrency control policies and, also, allow selecting different policies for different CIs.

III. ORION APPROACH

The Orion approach, proposed in this paper, aims at selecting suitable concurrency control policies for each CI. That selection is based on modification history stored in the version control system. Also, Orion approach intends to identify the CI that should get more attention regarding concurrency control, called critical elements.

Fig. 1 shows the Orion approach overview. Version control systems repositories are used as data sources. So, historical information about modifications are extracted from these repositories and filtered. This information includes all project CIs versions and data about these versions, like, for instance, check-in time and author name. After this, some proposed concurrency metrics are calculated for each CI and are stored in a different repository. Metrics results are used to select concurrency control policy for each CI and to identify critical elements. Finally, the output can be visualized in appropriate tools.

A. Concurrency metrics

IEEE Std 1540 [19] defines risk as the probability of an event, threat or situation to occur, related with undesirable consequences, thus, it is a potential problem. According to the risk management discipline [20], risks that should receive more attention in a software development project are those which have higher probability to occur and those which can cause a large impact in the project. Thus, one way to measure risks is using the risk exposition, which is calculated by means of the product risk probability and risk impact.

In the context of this paper, we consider losing productivity, caused by wrong selection of concurrency policies, as a risk. Based on the risk management discipline, we defined two concurrency control metrics: CI concurrency and CI merge effort.

Concurrency metric measures the concurrency level by developers over a CI. Similar to risk probability in risk management, the higher the concurrency level, the higher the conflicts occurrence probability [21], and, therefore, the lower the developer team productivity.

Merge effort metric measures the effort needed to solve conflicts in a CI. Similar to risk impact in risk management, the higher the difficulty to merge, the higher the impact in the development team productivity.

In order to calculate concurrency level for a CI, we analyze all check-out and check-in operations that were related to the CI in the past. We consider that the time needed to a developer to perform a modification can be represented by the time interval between check-out and check-in-operations. In this way, we use the historical information in the version control systems to identify how much time a CI has been in possession of developers for modifications, and how much this time was shared by different developers.

Fig. 2 represents a concurrency scenario over a CI. This is a visual representation of a modification history. Fig. 2 shows the modification activity of three developers over a CI from time 0 to time 10. A black line represents a modification activity that begins with a check-out (black circle) and ends in a check-in (white circles). For instance, developer B did a modification between time 4 and time 8. By analyzing the figure, we can identify three different situations: moments that nobody was modifying the CI; moments that only one developer was modifying the CI; and moments that more than one developer was modifying the CI.

Equation (1) is used to get the concurrency metric result for a specific time interval. We consider the time interval that more than one developer was modifying the CI (concurrency time - CT) and the time interval that at least one developer was modifying the CI (possession time - PT).

\[
\text{concurrency(CI)} = \frac{\text{CT(CI)}}{\text{PT(CI)}}
\]

In this way, the concurrency metric will always vary from 0 to 1, since concurrency time cannot be higher than possession time. For instance, in the example presented in Fig. 2, the concurrency metric is bound to 75% (6/8).

The merge effort metric measures how difficult it is to merge two CI versions in order to solve conflicts. This merge involves two versions of the same CI: the version modified by the developer who is performing the check-in and the most recent version in the repository at the time of analysis. Also, the value of this metric varies from 0 to 1. A high value means that conflicts in this CI are historically difficult to be solved.

![Figure 1. Orion approach overview.](image)
In order to get the merge effort result for a CI in a specified time interval, we need to analyze which CI parts were modified in each check-in operation. We consider that CIs are composed by smaller units, called comparison units [22], which are used to compare two versions of the same CI and to compute conflicts. For instance, if we consider a Java class as CI, attributes and methods can represent comparison units.

For each check-in operation, we can highlight four CI versions related to this operation (Fig. 3): base version, the one that the developer checked-out; user version, which was modified by the developer and is subject to the check-in under analysis; current version, which was in the repository when the check-in was performed. It may be different from the base version if other developer modified this CI since check-out; and final version, version successfully checked-in. In cases that conflicts were raised, the final version represents the merge of the user version and the current version; otherwise, final version will be equal to user version.

Having identified these four versions, they should be compared in order to identify the actions performed for solving conflicts. Actions are represented by add, delete, or modify operations over CI comparison units. These actions are divided in two groups: applied actions and committed actions.

Applied actions set represents actions over CI comparison units performed in the current version (CV) and in the user version (UV). In order to identify these actions, diff3 algorithm [4] is used (2), where BV represent the base version:

\[
\text{applied actions(CI)} = \text{diff3}(BV, UV, CV)
\]

(2)

Committed actions set represents actions over CI comparison units approved during merge and actually committed in the final version (FV). In order to identify these actions, diff algorithm [4] is used (3):

\[
\text{committed actions(CI)} = \text{diff}(BV, FV)
\]

(3)

The intersection of these two sets of actions is shown in Fig. 4. Actions that are in both sets represent applied actions that were committed in FV and did not demand effort during merge. Committed actions set can include different actions if compared to applied action set. Actions that are only in the applied actions set represent discarded actions, which mean wasted effort. Actions that are only in the committed actions set represent actions that were performed in order to solve conflicts, which mean additional effort. In this case, the merge effort was higher due to the need of additional actions to conciliate both CV and UV. In this way, equation (4) shows how check-in merge effort is calculated.

\[
\text{merge effort(check-in, CI)} = \frac{|\text{CA} - \text{AA}|}{|\text{CA}|}
\]

(4)

where \(|X|\) means the number of elements in the X set, AA represents the applied actions set, CA represents the committed actions set and \(\ldots\) represents sets difference operation.

In check-ins that did not exist conflicts or there was no need for additional actions, check-in merge effort is 0. On the other hand, when all committed actions represent additional effort, merge effort is 1, the higher value.

It is important to emphasize that the merge effort metric is calculated for each check-in operation that occurred in the past. After that, we need to calculate the average in order to get the merge effort of the CI for a specified time interval (5):

\[
\text{merge effort(CI)} = \frac{1}{n} \sum \text{merge effort(check-in, CI)}
\]

(5)
merge effort(IC) = \((\sum \text{merge effort(check-in, CI)}) / \text{ (quantity of check-ins)}\)  \hspace{1cm} (5)

Once both metric results were calculated, a third metric, called critical level, is defined by multiplying the concurrency and merge effort metrics for a specific time interval (6). It also varies from 0 to 1 and indicates how critical a CI is.

\[
\text{critical level(CI)} = \text{concurrency(CI)} \times \text{merge effort(CI)}
\]  \hspace{1cm} (6)

B. Concurrency control policies selection and critical elements indication

After calculating the metrics results, a suitable concurrency control policy is indicated according to Fig. 5. Depending on the threshold defined by users in a configuration phase, a metric value can be considered high or low. For instance, in a possible configuration, a value greater than 0.5 can represent a high value, otherwise it is low.

In cases where a CI has low values for both metrics, the two policies can be chosen unrestrictedly. In cases where a CI presents low concurrency value and high merge effort value, Orion approach indicates pessimistic policy because historically a merge is difficult to be solved and in most cases, there was low concurrency over this CI (i.e., work serialization is not a problem). In cases that a CI has a high concurrency value and low merge effort value, optimistic policy is indicated because if conflicts arise, they would not probably be difficult to solve, and the parallel development is important since most of the time there is concurrency over this CI. Finally, in cases that a CI presents high values for both metrics, this CI is identified as a critical element regarding concurrency control. In this situation, both policies could bring problems to the development process. Therefore, Orion approach suggests that a refactoring should be performed in these elements. Probably, these elements are responsible for multiple functionalities, leading to low cohesion [23]. Dividing them in more than one element could positively impact in the metric values and the element would not be considered as a critical element anymore, possibly reducing the productivity loss risk.

C. Orion approach application

The Orion approach results can support different roles in various ways in software development process. First, it can be used by developers, who can adopt a different CI concurrency control policy based on Orion suggestions, for instance, a developer can lock a CI with high merge effort before modifying it. Also, configuration managers can use the results to make sure that developers will adopt the chosen approach (lock-based or merged-based) by means of version control systems mechanisms. In this case, developers will focus on their job and worry less about parallel development issue and its consequences. Project managers can also use Orion results in order to better allocate human resources, they can avoid allocating many developer to concurrently work over a CI that presents a high merge effort metric value, trying to minimize the impact of possible conflicts. Finally, software architects can use Orion critical elements identification mechanism for performing refactoring.

Orion approach does not intend to act intrusively in the software engineer’s work. It makes suggestions and software engineers will decide if suggestions will be adopted. However, Orion approach could work together with the adopted version control systems in order to automatically make the policy selection. Also, the time interval analyzed by Orion is flexible. We must emphasize that it needs historical information, thus it can only be applied after the beginning of the project. Besides, Orion approach can be applied constantly, since new data is generated continuously. The integration with the version control system could be useful in this task as well: every time that a check-in is performed, metric values of CIs involved in this operation can be updated.

In order to achieve its objectives, Orion approach uses several pieces of historical information. Most of them are available in most version control systems. However, some are unavailable in some system, like, for instance, moments that check-out operations were performed and moments that conflicts were raised. Besides, most current version control systems, with few exceptions [24], lose information during parallel development. This occurs because users are forced to merge before they check-in. Due to that, there is no way to identify the original change intention of the user, but only the change already merged with changes performed by other users.

IV. Evaluation

Before building a prototype to automate the Orion approach, we wanted to verify if the proposed approach was really effective. It is important to notice that a tool only ensures the approach is being executed in an efficient way, but does not guarantee that it is being executed in a correct way. If the approach is not effective, the results of automatizing it would be catastrophic.

Two preliminary experimental studies have been executed in order to evaluate Orion approach. The studies results are restricted but they contributed to identify limitations and improvements of the proposed approach.
The first study aimed to characterize the feasibility of the concurrency control policies selection mechanism in coordination support of software development teams. The second study aimed to analyze its application.

The studies were organized in sessions that involved only one person (subject) each. There were three sessions, with three master students, for each study. Consequently, six different students were involved in these studies. The situation proposed to each subject was the following: he/she had just been hired by an organization as software configuration manager. His/her activity was to select concurrency control policies for each CI of a development project, aiming at increasing development team productivity. The development project used in both studies was intentionally very simple. There were only 6 CIs and about 30 check-in operations performed by 3 developers adopting an optimistic policy. Information about the project was provided (i.e., project description and class diagram). Also, two kinds of historical information were provided to the subjects, depending on the study stage: historical information usually stored by version control systems, and some additional historical information identified by Orion approach. Therefore, the subjects’ tasks can be summarized as: to analyze the provided information and to select the adequate concurrency control policy for each project CI based on the information that they had available.

All subjects involved in the studies were familiarized with using control version systems. Also, they had experience working with other developers in a development team either in the academy or in industry.

The metrics collected in these studies were: the selection of concurrency control policies, and the time for analyzing the provided information and selecting the policies. Besides, at the end of the section, subjects filled a form with some questions about the study.

A. Feasibility study

In this first study, our objectives were: to identify which kind of information is used in order to select concurrency control policies, and to verify if the additional information proposed by Orion is useful to select concurrency control policies.

This study was divided in two stages. In the first one, the project information and historical information usually stored by version control systems were provided to the subjects. Then, the subjects performed the policies selection for the 6 CIs. After that, in the second stage, the additional historical information identified by Orion approach was provided. Then, again, the subjects performed the policies selection for each 6 CIs.

By analyzing these results, we could conclude that, in the first stage, the information used by the subjects was: the number of developers who modified the CI (cited by all 3 subjects) and the time interval between check-in operations (cited by one subject).

Also, we could conclude that, in the second stage, besides the information mentioned in the first stage, they also used: concurrency time periods (mentioned by all 3 subjects), number of concurrent developers (mentioned by 2 subjects), number of conflict occurrences (mentioned by 2 subjects), and solving conflict effort (mentioned by 2 subjects). Those new pieces of information were identified by Orion approach.

Comparing the selections of concurrent control policies performed by the subjects in both stages, we noticed that the additional historical information identified by Orion approach did impact the selections. In some cases, the selected policy for the same CI, in the same situation, changed from the first stage to the second. Moreover, all subjects stated that the additional historical information supported them in the concurrency control policies selection activity, leading to higher confidence in the policies selection.

In general, the study results were positive, however we faced some issues. For instance, on the second stage, one subject did not use the additional historical information provided to analyze the merge effort, as we expected.

B. Application study

In this second study, our objectives were: to verify if the concurrency control policy selection mechanism is coherent with subject’s judgment, and to verify if subjects agree with Orion approach suggestions about concurrency control policies and critical elements.

The application study was also divided in two stages. In the first stage, the project information, historical information usually stored by version control systems, and additional historical information identified by Orion approach were provided to the subjects. Then, the subject performed the policies selection for the 6 CIs. After that, in the second stage, the Orion approach was presented and the Orion results were provided for each of the 6 CIs. After that, the subjects analyzed the results and informed if they agree or not with them.

By analyzing these results, we could conclude that the concurrency control policy selection mechanism is coherent with the subject’s judgment. In most cases (56%), the policy selections matched. In cases where Orion suggested refactoring for CIs (33%), the answers differed. However, refactoring was not an explicit option for subjects. In the other cases that the answers differed (11%), subjects agreed in changing their answers after seeing the Orion suggestion based on how the approach works.

Generally, the study results were also positive. In most of the cases (94%), the subjects answered the same as Orion approach or they said they agree with the Orion result. In only one case (6%), the subject did not accept the Orion suggestion. This variance occurred because the subject used time to measure merge effort, instead of performed actions as used by Orion approach.

C. Evaluation conclusions

The presented studies represent an initial evaluation of Orion approach. These are simple studies, which involved few subjects, and, thus, the results cannot be generalized. Surely, additional studies are necessary in order to evaluate the proposed approach in a more complete way.
Also, both studies intentionally presented very simple situations. The project under analysis was very small, involving only 6 CIs and about 30 check-ins performed by just 3 developers. The subjects took about 30 minutes to analyze the information provided and select concurrency control policies for each CI. A real industry development project certainly presents higher numbers. For instance, the version control system repository of the Subversion project [13] shows modification history since 2001. This project has about 5,000 CIs, 33,000 check-in operations and involves more than 100 developers. If we compare our studies to a real development project like Subversion, the time needed in order to analyze and select policies for 33,000 CIs would be extremely high. Certainly, it would be very difficult and lengthy to do it manually.

Facing the positive results, we developed a prototype that implements Orion approach, which is presented in the next section.

V. PROTOTYPE

As presented before, in order to achieve its objectives, Orion approach uses historical information from version control systems. However, depending on the version control system, some pieces of information are not collected and are not available. Orion prototype was implemented using Odyssey-VCS [22, 25, 26] as data source.

Odyssey-VCS is a version control system tailored to fine-grained UML model elements, that was conceived to work in a similar way to popular file-based version control systems, such as Subversion. Models are checked-out from a central server, changed by users in private workspaces, and finally checked-in back to the repository.

Odyssey-VCS has received some new features in order to allow Orion approach application [26]. One of the new features is the auto-branching concept that consists on automatically creating a branch for every check-out/change/check-in cycle (Fig. 6). Due to that, it is possible to identify not only the change already merged with changes performed by other users, but also the original change intention of the user. With auto-branching, when conflicts arise, we can store both the intention (user version) and the merged changes (final version). New features also include: pessimistic concurrency control policy support, that prohibit other architects to change a specific part of the model for a period of time, by using locks; and the check-out transaction concept, that stores the moment that users performed check-outs.

Orion prototype was built as a plug-in for Odyssey-VCS. The first step in order to run the prototype is to configure the thresholds, discussed in Section III, which will be used to determine if a metric value is high or low. Then, the user can choose from two operation modes. In the first mode, the user runs the approach for only one UML element and sees the results in a pop-up window. In the second mode, the approach is applied for all CIs in a UML model. At the end, a new UML model identical to the original model is generated, but appended with tagged values and stereotypes (UML extension mechanisms) that describe all metric values, critical elements, and concurrency policy selection. This new model is exported as an XMI file, allowing the user to visualize the Orion results in any UML CASE tool.

A better way to visualize Orion results was built using the EvolTrack mechanism (Fig. 7). It provides a visualization infrastructure for the evolution of different versions of a development project from a version control system. This visualization mechanism was implemented as an Eclipse development environment plug-in [27].

EvolTrack can also interpret and represent metrics values attached to UML models. Users can select a specific metric and then CIs are colored based on the metric values for this CI. All Orion metrics vary from 0 to 1. Thus, we use a color scale from green (0) to red (1). Fig. 8 illustrates an Orion results visualization example of CIs from a simple model. In this case, UML classes are CIs, which are colored based on critical level metric value. The most critical element can be easily identified by reddish colors.

Another advantage of using EvolTrack to visualize Orion results is analyzing CIs metric values evolution. With all versions analyzed, EvolTrack can generate an animation that shows the software structure evolution, during the development life cycle, with CIs changing colors based on metric values in specific periods of time.
VI. RELATED WORK

Concurrency control plays an important role in the software development process. Here we briefly discuss about some related work regarding concurrency control in software development.

Some approaches, such as [8] and [28], propose a new SCM systems infrastructure in order to better provide scalability and efficiency issues for concurrency control. Thus, new kinds of concurrency control policies are proposed for this new infrastructure. They entail some new concepts like user groups with the same interest. Also, concurrency control policies definition languages are defined based on those new concepts in order to make the creation of new kinds of policies possible. In spite of allowing concurrency policies definition, these approaches do not support the selection of which policy is more appropriate for each CI.

Some other approaches, such as [12], support distributed software development by early conflict detection in order to mitigate its impacts. They make it possible to identify that another developer is working on the same artifacts and then try to avoid conflicts before check-in. In these approaches, there are some important contributions for development awareness, however, they introduce some new activities for software engineers because they have to observe other software engineer’s modifications and be aware of possible conflicts. We consider our approach complementary to these...
approaches because we do not focus on enforcing or supporting specific concurrency policies, but on identifying the most appropriate policy for a given scenario.

VII. CONCLUSIONS

We presented in this paper an approach for concurrency control policies selection and critical elements identification, supporting different development process roles. Also, we presented preliminary evaluation studies and a prototype that implements the proposed approach.

Orion approach has some limitations. The required data used to calculate the proposed metrics can be easily and automatically extracted from Odyssey-VCS, but this task can be difficult in some version control systems. This happens because Orion approach needs some historical information, such as check-out time and user’s version (initial check-in version), that are not available in some version control systems like. However, there are some alternative ways to collect and store this information using external tools, via branches and hooks, but this discussion is out of the scope of this paper. Besides, in some situations, the metrics may present unexpected results. For instance, concurrency metric do not consider the number of concurrent developers. We intend to refine them along the performance of additional studies. Furthermore, Orion approach does not work for recently created CI, that is, a CI that does not have available historical information about modifications. In these cases, we need to use a pessimistic policy and wait until some historical information is available to apply the proposed approach.

As future work, we intend to perform deeper studies in order to evaluate Orion approach and its prototype, and after that apply them in real software development projects. This will provide stronger improvement indications, such as, for instance, an evolution and adjustment of how the metrics are calculated. Also, other metrics can be defined in order to enforce Orion suggestions. Moreover, another future work is to suggest how the critical elements can be refactored.

REFERENCES


