International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:8, No:6, 2014

Systems Versioning: A Features-Based Meta-Modeling Approach

Ola A. Younis, Said Ghoul

***Abstract*—**Systems running these days are huge, complex and exist in many versions. Controlling these versions and tracking their changes became a very hard process as some versions are created using meaningless names or specifications. Many versions of a system are created with no clear difference between them. This leads to mismatching between a user's request and the version he gets. In this paper, we present a system versions meta-modeling approach that produces versions based on system's features. This model reduced the number of steps needed to configure a release and gave each version its unique specifications. This approach is applicable for systems that use features in its specification.

***Keywords*—**Features, Meta-modeling, Semantic Modeling, SPL, VCS, Versioning.

I. Introduction

NO legacy system may be used without modifying, changing, updating or replacing some part of it. Tracking these changes requires system to control them and enable storing and retrieving past versions [1]. Version control systems (VCSs) [1]-[5] is the best approach to guarantee system versioning consistency; since it provides user feedbacks and work history for any development or change. Without VCSs, the control of systems versioning and development is very hard process and errors prone. From the very beginning of version control systems the developers determine the relations between current versions and the new ones created [6].

VCSs main goals are to support system developers to work concurrently, insuring that their changes are consistent with each other and finally, to store any changes in a history archive [7]. These systems can be classified based on the working techniques [1]-[4], [7]-[9] into two types: Concurrent Versioning Control Systems (CVCSs) and Centralized Versioning Control Systems (CVCSs).

Nowadays, systems became more complicated and forked. Several versions with several descriptions and details are included in each one. For firms, in order to present a new version for their product, they build the new name based on the latest version name for the product [9]. Like V1.3.1, V1.3.2. For end user, having a lot of versions names and numbers makes the selection for a suitable version a very hard and confused process[10]. Which version to choose? Which features does this version include? What are the relations between the versions? and many other questions.

Ola A. Younis is a lecturer of computer science working with the Bio-inspired Systems Research Laboratory, Philadelphia University, Amman, Jordan, (e-mail: oyounis@philadelphia.edu.jo).

Prof. Said Ghoul is the leader of the Bio-inspired Systems Research Laboratory, Philadelphia University, Amman, Jordan, (e-mail: sghoul@philadelphia.edu.jo).

Laskey in his work [11] reported that version identifier (number or name) should be interpretable to reflect some of version's main features and structure. Version's identifier interpretation include understanding the functionality and use for each version in order to increase its usability [11]. Another challenge, reported in [10], is the vibration effect of change. To approve any new change or development for the product, the developer should schedule the change process to guarantee system consistency during its development [10]. Several other challenges were reported in [1], [2], [4], [7], [10], [11].

This paper deals with the following challenges: (1) Several version identifiers for each product without a clear methodology for identifier building, (2) lack of version's identifier interpretations, and (3) lack of change schedule to guarantee system consistency during the version configuration.

Enterprise systems in complex organizations support large number of different components and are composed of multiple units and variant areas of interests; hence, these systems have to control all processes, reports, and versions configuration. Thus, each unit may have several possible values to cover. The sources for these values are different: domain analysis, stockholders' needs, system evolution and so many other sources [12]. The ability of a system to be generalized, specialized or customized to perform special needs is the base for new versions to be created based on system's main features [12].

One way to deal with system's units and components and identifying them is the use of features that are defined for the whole system [13]. System features are defined in a feature diagram showing the parent feature and its children using relations like OR, AND, INCLUDE, EXCLUDE, and many other relations [8], [12], [13].

Researchers presented feature modeling in three approaches: Graph notations based, Text notations based, and Mixing graph and text based approaches [12]. These approaches are classified based on the technique used to capture system's main features and functionalities. Each approach has its strength and weakness points.

Feature modeling used to describe system's common and variable components [12]. Common features present the constant behavior for system components. while variable features present the behaviors that change due to problem context and use the optional features to present it [8], [12], [13]. Using feature modeling increase system reusability and efficiency [12], since it shows the main components in a feature hierarchy and enable the developer to reuse them [13]. This will enhance the configuration process and make it easier than choosing components individually without specifying the relation between them [12], [13]. Others benefit for using feature modeling is its contribution in system specialization [13], synchronization between modeling and configuration [2], [5], [8], [12], its powerful semantics [12], [13] and many other benefits.

Several approaches presented the versioning concepts based on system's features. Some of these approaches focused on the development phase for the system, while others focused on the configuration and versioning concepts. CLAFER model [14] reported a new way for mixing the structural components (class model) with the conceptual components (feature model) of the system. This mixing was done based on constraints and inheritance concepts. Feature concepts were presented as a collection of type definitions and features. CLAFER had two main problems. The first one is that CLAFER did not define the connections between multiple features. These connections are very important to insure consistency during the mixing. The second problem is the weak representation of features' possible values that may be used in the feature model.

In the work presented by Gunther and Sunkle [15], a new creative programming language called RBFEATURES was presented. This language was built on top of dynamic programming language (ruby). RBFEATURES faced three main problems. Firstly, the classification for the used features was missing. Secondly, the relationships between the features were not specified. And finally, tracking version process was very hard, since the version is defined based on configuration only and not on features.

An Object-Oriented feature model that combines feature models' concepts with object-oriented concepts was presented by Sarinho and Apolinario in [16]. They proposed a new object-oriented feature model (OOFM) that captures both the feature model and feature modeling package. The problem with model reported in [16] is that it did not separate between feature and object model, which leads to a complex system.

Based on the weaknesses mentioned above, this paper proposes, some enhancements to the actual state of the research in this domain: (1) A features-based meta-model for versions configuration. (2) A Classification of the features used in building any system versioning model. (3) A combination of versioning and features concepts to build versions based on user-defined features. And (4) A semantics is given to each version based on the features it includes. The proposed meta-model supports the above two first versioning challenges are targeted by this paper.

This paper is organized as follows: Section II presents a versioning feature-based meta-model approach introducing some enhancements. These enhancements are evaluated in section III. Section IV presents a conclusion and perspectives of this work.

II. A Versioning Features-Based Meta-Model Approach

This section presents a version features-based meta-model, a configuration features-based model, and a versioning features-based approach with an example, the "Set" component, to simplify the idea. A Set is a variable class, having several model versions such as: Static stack, static queue, dynamic stack and dynamic queue. In the following, some significant parts of this example are presented. The complete case study is presented in [12].

A. A Version Features-Based Meta-Model

The proposed versioning model includes a features-based meta-model and an asset meta-model. The versioning meta-model is composed by four meta-features as shown in Fig. 1.

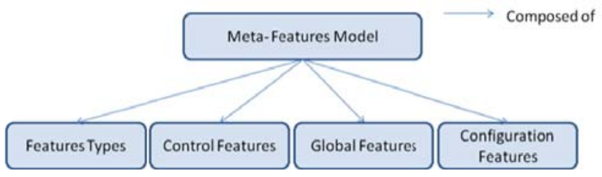

Fig. 1 Meta-features Model

These meta-features are:

- *Feature Types*: This meta-feature captures all features and their possible values in the system. These features include characteristics and relations.

Ex: Set.Behavior={static,dynamic};
Set.Scope ={shared, separated};

This example shows that the possible values for Behavior feature are either static or dynamic. And the possible values for Scope feature are either shared or separated.

- *Control Features*: This meta-feature captures the relationships between all system's features. These relations insure versions consistency during automatic system configuration generation from components versions.

Ex: Beh.static <excludes> Beh.dynamic;
Beh.dynamic <imply> datastr.dynamic

This example shows that the static behavior excludes the dynamic behavior. And the dynamic behavior implies a dynamic data structure. This means that for any version in "Set" example, you can't have dynamic and static behavior at the same time. And if you choose a dynamic behavior, your data structure must be dynamic.

- *Global Features*: This meta-feature captures the common (shared) features between all system components versions.

Ex: Set.Form={ch,con}; Set.View={ll,cl};

This example shows that the default values for the form feature are chain and continuas. And the default values for the view feature are linked list and closed list. This means that for each version of "Set" example, the version's view is either LL or CL. And the version's form is either Ch or Con.

- *Configuration Features*: This meta-feature captures each configuration (release) specification.

Ex: Features Configuration
{Name: S_stack
view.cl<require> state.correct;
<reject> scope.shared; }

This example shows the configuration process for static stack. In this configuration, we insure that the view must be cl and the state is correct. This configuration rejects the shared scope. Other features are automatically added according to the relations between "Set" features.

The connections between these meta-features are described in Fig. 2.

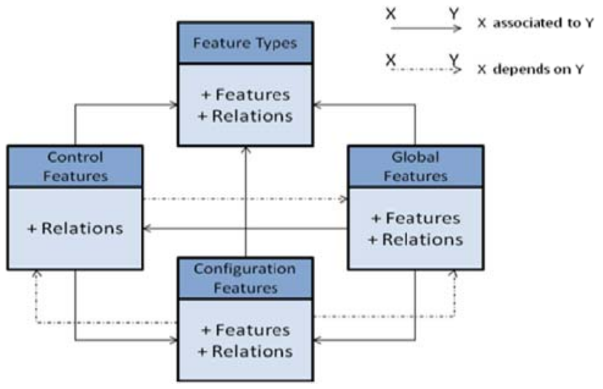

Fig. 2 Meta-features connections

The Asset meta-model is composed by class interfaces and their implemented attributes, methods and implementations that present the final and real component that will be provided to the end user as a new version of the system (Fig. 3).

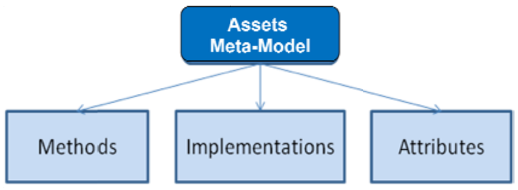

Fig. 3 Assets Meta-model

B. Configuration Features-Based Model

After instantiating, for a system, the two version features-based meta-models described above, the configuration process can be enabled to create software releases based on system's features. For best understanding of the configuration process, we present the configuration model in Fig. 4.

The configuration starts by defining features meta-model that was described in Fig. 1. Then, and instance of this meta-model is created to produce features model that specifies the Features types, Control, Global features and Configuration features. This step creates only the structure of these features and do not specify any real values.

Another model is created after the features meta-model. This model is the assets meta-model that was reported in Fig. 3. After defining all features and relations for the requested version, the real configuration will start. Before starting configuring a new version, a request sent to versions repository searching for a version with specifications defined by features model. If the requested version was available, the user can take a copy (release) of that version. If not, a new configuration process will start.

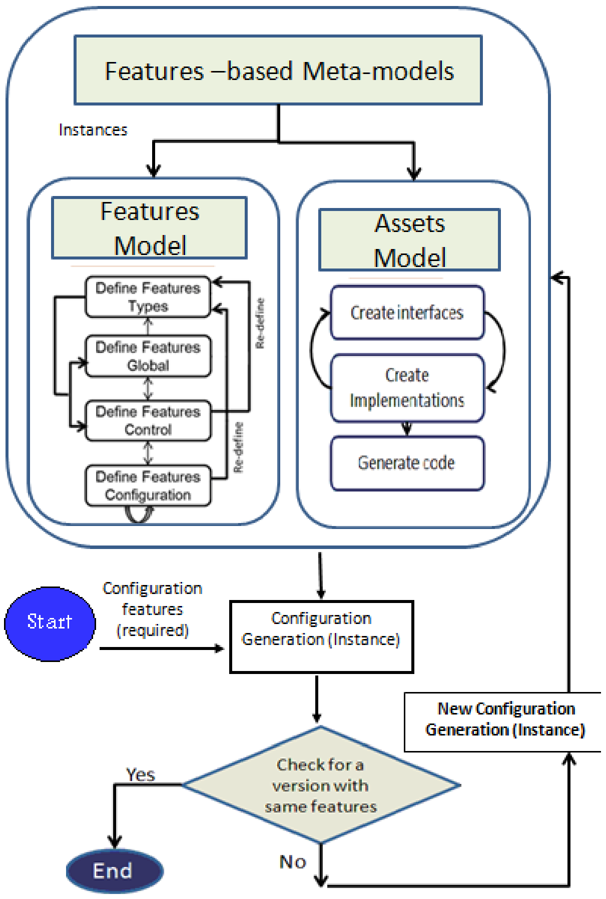
Features –based Meta-models
Instances
Features Model
Define Features Types
Define Features Global
Define Features Control
Define Features Configuration
Re-define
Re-define
Assets Model
Create interfaces
Create Implementations
Generate code
Start
Configuration features (required)
Configuration Generation (Instance)
New Configuration Generation (Instance)
Check for a version with same features
Yes
No
End

Fig. 4 Configuration features-based model

Configuration generation (version instantiation) takes the Configuration features (that were defined in the previous step) with the assets meta-model to create assets model. This model defines version's interfaces and implementations and creates code for these implementations. This means specifying a real values foe all features, relations interfaces and implementations, which produce (as a result) a new version for the system based on user's pre-defined features. Next, we add the new derivative version to versions repository for future use.

C. Features- Based Framework

In this section, we present an approach that captures the configuration steps based on pre-defined features. This approach can be applied to any type of feature-based systems. Firstly, let us define some used concepts:

- *Version specifications*: this term refers to the user pre-defined specifications for requested version. The user requests a version from the system, if there were a previous version with same specifications (the version already has been configured by the system), then the system replies with a copy of that version.
- *Requested Version*: the version that user is requesting from the system.
- *Version configuration*: the process of building new

version with user's specifications.

- *Versions repository*: a directory that contains versions that have been done previously by the system. A copy or pointer for each version (including all its features, relations and any other resources) is stored.

Fig. 5 presents a pseudo code model of the full approach for any request from the users.

```
V_Speci >> version specification
R_Vers >> Requested version
V_Config >> version configuration
V_Repo >> versions repository.

V_Speci← null;
R_Vers ← null;
V_Config ← null;

For each user_requirements for any version

    V_Speci← {feat1, feat2, ….,featn};
    Create Features Meta-Model;
    Create Assets Meta-Model;
    Create Features Model;

    Check V_Repo
    If( Version(V_Speci)) exists Then
            R_Vers←Copy of Version(V_Speci).Config;
    Else
       Check feat.control;
       V_Config←feat.control. derivative;
    End If
    For each feat in V_Speci
      Check feat.global;
     V_Config←feat.global. derivative;
   End
   Foreach feat in V_Speci
      Check feat.derivative.control;
      V_Config←feat.derivative.control. derivative;
  End
```

Fig. 5 Features-based framework

Based on this approach and features-based model presented in the previous section, each version holds a meaningful semantics based on the features it includes. This step will enhance version systems and simplify the process of change tracking and versions classifications.

III. Evaluation and Perspectives

In this section, we introduce the implementation issues of the proposed approach, its application areas, and its technical comparison with others relevant works.

A. Implementation Issues

Any environment that may use our model needs a strong object-oriented and feature oriented programming languages. All mentioned features have to be implemented in classes and objects environment to be used later in each configuration version.

Each configuration version is an object instantiation based of the classes and relations between them that are defined based on the features and their relations.

B. Application Area

The proposed approach supports software engineering, reverse engineering, and reengineering tasks by adding the features to its process and classifying them in a way that will enhance versioning process.

Big systems that require VCSs, like operating systems, enterprise systems, multi-agent systems and others may highly take advantages by using this approach.

C. Comparisons with Similar Works

Since the presented approaches is a modeling technique for the versioning process, its comparison with others relevant works [3], [5], [8], [14], [15] will be based on specific versioning criteria. The selected criteria are:

- *Covered steps in software process.* The proposed approach and the work presented in [14], cover the design and implementation phases. The works [3], [5] covered the implementation step, while the work [15] covered only the design phase.
- *Mixing feature and versioning concepts.* In the presented work, mixing feature and versioning concepts was achieved by extending versioning concepts with feature concepts to produce features-based model. This step was missed in [3], [5], [15]. Kacper et al. [14] presented two separated models and concepts for versioning and features models.
- *Supporting approach.* The proposed approach supports configuration's methodology and a design pattern that is applicable for any system to create versions based on pre-defined features. But this step was not covered by any of the presented researches. Configurations were carried out individually without any formal way.
- *Using reduced number of concepts and having a uniform semantics.* In the introduced approach, we reduced the number of concepts that may be used in each configuration process by classifying the features into global, control and configuration ones and defining their syntax and semantics. This step is very important in large systems where versions number is very huge and configuration's time is important. This step was missed in [5, 14] and partially applied in [3, 15].
- *Enhancing Software product Line (SPL) area.* This step has a nature relation with the previous one. SPLs will be enhanced by features-based configuration model that has been defined using strong syntax and semantics and using reduced number of concepts to produce system versions.

IV. Conclusions

In this paper, we presented a features-based model and approach for the configuration process. We classified the features that may be used in any system based on their functionalities into Global, Control, and configuration features. These features capture all possible versions based on the relations and derivative features that result from combining them together in system version or release. The process of configuring new versions, based on user-defined features (specifications), is automatically carried out.

References

[1] Ba, M.L., A. Talel, and S. Pierre, Uncertain version control in open collaborative editing of tree-structured documents, in Proceedings of the 2013 ACM symposium on Document engineering. 2013, ACM: Florence, Italy.

[2] Bauml, J. and P. Brada. Automated Versioning in OSGi: A Mechanism for Component Software Consistency Guarantee. in Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on. 2009.

[3] Gomez, V.U., S. Ducasse, and T. D'Hondt, Visually characterizing source code changes. Science of Computer Programming, 2013(0).

[4] Jiang, Z., How to give away software with successive versions. Decision Support Systems, 2010. 49(4): p. 430-441.

[5] Lindkvist, C., A. Stasis, and J. Whyte, Configuration Management in Complex Engineering Projects. Procedia CIRP, 2013. 11(0): p. 173-176.

[6] Rochkind, M.J., The source code control system. IEEE Transactions on Software Engineering, 1975. 1(4): p. 364–370.

[7] Sink, E., Version Control by Example, ed. B. Finney. July 2011: Pyrenean Gold Press.

[8] Buchmann, T., A. Dotor, and B. Westfechtel, MOD2-SCM: A model-driven product line for software configuration management systems. Information and Software Technology, 2013. 55(3): p. 630-650.

[9] Jannik, L., et al., Supporting simultaneous versions for software evolution assessment. Science of computer programming 2011. 76(12): p. 1177-1193.

[10] Marc Novakouski, G.L., William A,nderson and Jeff Davenpor, Best Practices for Artifact Versioning in Service-Oriented Systems in SEI Administrative Agent T. Research, and System Solutions Program, Editor. 2012, Carnegie Mellon University.

[11] Laskey, K. Considerations for SOA Versioning. in Enterprise Distributed Object Computing Conference Workshops, 2008 12th. 2008.

[12] Ola Younis, S. Ghoul, and M. Al Omari, Systems variability modeling: A Textual model mixing class and feature concepts. International Journal of Computer Science & Information Technology (IJCSIT), 2013. 5(5): p. 127-139.

[13] Don, B., Feature models, grammars, and propositional formulas, in Proceedings of the 9th international conference on Software Product Lines. 2005, Springer-Verlag: Rennes, France.

[14] Kacper, B., Clafer: a unifed language for class and feature modeling. 2010.

[15] Sunkle, S.G.S., rbFeatures: Feature-oriented programming with Ruby. Science of Computer Programming, 2012. 77: p. 152-173.

[16] V. T Sarinho, A.L.A.E.S.d.A., OOFM - A feature modeling approach to implement MPLs and DSPLs, in EEE 13th International Conference on Information Reuse and Integration (IRI). 2012, IEEE.