OrionPlanning: Improving Modularization and Checking Consistency on Software Architecture

Gustavo Santos, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse
RMoD Team
INRIA, Lille, France
{firstname.lastname}@inria.fr

Marco Tulio Valente
Department of Computer Science
UFMG, Belo Horizonte, Brazil
mtov@dcc.ufmg.br

***Abstract*—Many techniques have been proposed in the literature to support architecture definition, conformance, and analysis. However, there is a lack of adoption of such techniques by the industry. Previous work have analyzed this poor support. Specifically, former approaches lack proper analysis techniques (e.g., detection of architectural inconsistencies), and they do not provide extension and addition of new features. In this paper, we present OrionPlanning, a prototype tool to assist refactorings at large scale. The tool provides support for model-based refactoring operations. These operations are performed in an interactive visualization. The contributions of the tool consist in: (i) providing iterative modifications in the architecture, and (ii) providing an environment for architecture inspection and definition of dependency rules. We evaluate OrionPlanning against practitioners' requirements on architecture definition listed in a previous survey. We also evaluate the tool in a concrete example of software remodularization.**

Demonstration video URL: http://youtu.be/TWWPgjRIljk

***Index Terms*—Software Architecture; Software Maintenance; Architecture Description Language; Architecture Conformance; Remodularization; Rearchitecting.**

I. Introduction

The definition of the software architecture[1] in early stages of development is of utmost importance as a communication mechanism between stakeholders. It does not only involve the decisions and guidelines that must be followed during the software evolution, but also provides discussion about the requirements (and conflicting parts of them) that the software might have. Inconsistent architectural decisions might deeply compromise the success of a software project [3].

Although many techniques on architecture definition have been proposed in the literature, there is still little adoption of such techniques in industry. Previous work analyzed this gap between research and practitioners' needs [4, 7, 15]. Recently, Malavolta *et al.,* [9] conducted a study on practitioners' needs in architectural languages (*i.e.,* any form of expression, informal or formal, for architecture description) and which features might be useful for industry projects. They identified seven main features which were useful in past projects:

- *Tool Support:* the availability of a mature architecture description tool for a given architectural language.
- *Iterative Architecting:* the ability to refine the architecture from a general description. The language should not require the stakeholders to fully describe the architecture.
- *Analysis:* the ability to extract and analyze information from the architecture for testing, simulation, and consistency checking, for example. One of the most clear results from the study is the need for proper analysis techniques to detect inconsistencies in the architecture before it would be applied.
- *Multiple Architectural Views:* the ability to provide different representations based on different architecture attributes. Some examples of views include structural, behavioural, data-flow, and semantic views.
- *Versioning:* the ability to store, keep track of changes, and share a given description.
- *Graphical Syntax:* the ability to provide non-textual representations focusing on a different properties of the architecture.
- *Well-defined Semantics:* the ability to provide the definition of extra-functional properties.

In this paper, we present OrionPlanning, a prototype tool to support architecture definition. The user can create an architecture definition from scratch (Section II-A) or iteratively modify a current modularization extracted automatically from source code (Section II-B). OrionPlanning provides a simple and interactive visualization to assist remodularization[2] (Section II-C) and an analysis environment to check whether the current architecture is consistent according to user defined restrictions (Section II-D). We evaluate this tool in a specific case of software evolution (Section III), and we compare the tool to previous work on practitioners' needs for architecture definition tools (Section IV). Section V presents related work and Section VI concludes this paper.

II. OrionPlanning in Action

Figure 1 depicts the main user interface of OrionPlanning. It is build on top of the Moose platform [2]. The panel in Figure 1.A shows the systems under analysis and their versions, followed by a panel for color captions (Figure 1.B), and the list of model changes in the selected version (Figure 1.C). On the

[1] In this paper, we consider the definition of Garlan and Perry: the structure of components of a program, their relationships, and principles and guidelines governing their design and evolution over time [6].

[2] We consider remodularization a sequence of transformations (not necessarily behavioral preserving) restricted to the architecture.

Accepted for publication by IEEE. © 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

right side of the window, ORIONPLANNING generates a simple visualization of model entities and dependencies (Figure 1.D) and a list of dependency constraints which will be evaluated when the model changes (Figure 1.E).

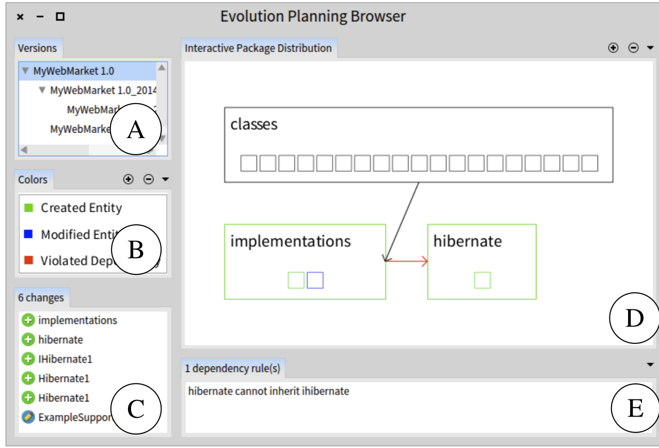


Fig. 1. ORIONPLANNING overview. The panel (D) shows three packages

A. Loading Code

To modify an existing project with ORIONPLANNING, the MOOSE platform already provides support to import code written in C++, Java, Smalltalk, Ada, Cobol, and other languages. The result is an instance of the FAMIX meta-model [5]. FAMIX is a family of meta-models that represent source code entities and relationships of multiple languages in a uniform way. We chose FAMIX because MOOSE already provides inspection and analysis tools which can be extensible for our work (see Section II-D). Details on how to import models in MOOSE are provided in The Moose Book [10]. After importing the code, a new model appears in the model selection panel (Figure 1.A).

B. Versioning

We use ORION [8] to perform changes on the FAMIX model extracted in the previous step. ORION is a reengineering tool that simulates changes in multiple versions of the same source code model. ORION efficiently handles the creation of *children* models. A *child* model has one reference to its parent version and a list of changes that were made in it. We use ORION because it manages modifications in multiple versions, including merging and resolving conflicts, without creating copies of the source code models.

Figure 2 shows the panel for model management. In practice, from a given model, the user can (i) inspect the changes in the current version, i.e., check the list of changes and modified entities (see also Section II-D); (ii) eventually create a new child version and make changes in it (see Section II-C); and/or (iii) discard the current version for different reasons. When the user is modifying a child model, the original one is not modified and no copies of this model are created. The list of changes is also displayed in OrionPlanning's main window (see Figure 1.C).

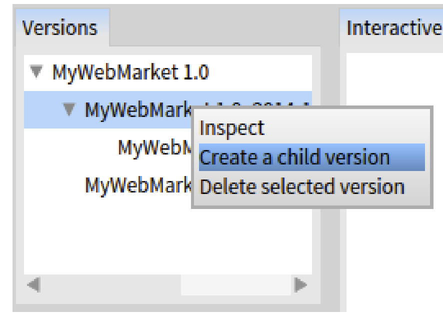


Fig. 2. ORIONPLANNING'S model selection menu

C. Remodularization and Visualization

In order to provide architecture visualization, we use a visualization engine called TELESCOPE. Figure 3 illustrates a visualization of a Java project. Packages are rectangles with classes represented as squares inside the package. Both packages and classes are expandable by mouse click. After expansion, a class shows its methods as small squares. In general, entities that were changed in the current model have their borders colored in blue, and the borders of entities created in the current model are colored in green.

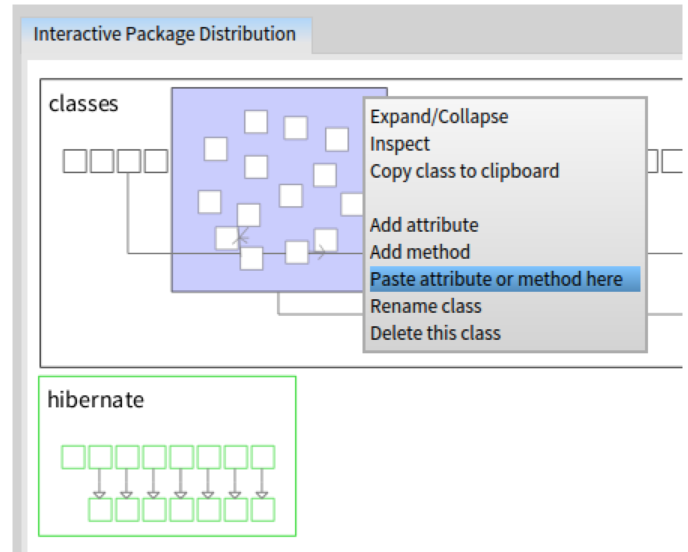


Fig. 3. ORIONPLANNING'S interactive visualization (right click on a class)

The visualization displays three types of dependencies: (i) the package dependency summarizes all dependencies (*i.e.*,, accesses, references, invocations, etc.) at a package level; (ii) the class dependency basically shows inheritance dependencies between classes inside one package; and (iii) the method dependency shows invocations between methods of different classes. We decided to show a fraction of all the dependencies to not overload the visualization with edges. Refactoring operators (*e.g.*,, add, paste attribute, and remove entities) are accessible by right click menu, as shown in Figure 3.

D. Model Analysis

According to previous survey with practitioners, the feature they missed the most in past projects was the support for architectural analyses [9]. Some of suggested analyses include dependency analysis between entities in an architecture, and consistency of architectural constraints [11]. In this section, we describe our work on extending ORIONPLANNING to provide dependency checking constraints defined by the user.

MOOSE provides a set of metrics for software, such as size, cohesion, coupling, and complexity metrics. From the visualization provided by ORIONPLANNING, any entity can be inspected and evaluated at any time (right click, Inspect). ORIONPLANNING allows the user to define rules based on these metrics. A possible example consists in restricting the number of classes in a package to less than 20. Due to space constraints, we do not show this feature in this paper.

ORIONPLANNING also supports the definition of dependency constraints. The definition uses the same syntax of DCL [12], a DSL for conformance checking originally proposed to Java systems. In ORIONPLANNING, the user first defined logical *modules* as a set of classes. These classes can be selected by matching a property (*e.g.,* a regular expression), or by manually selecting them into the module. Figure 4 depicts the module definition browser, in which the user selected (by regular expression) all classes which name ends with “Action”. Other properties can be easily extended to the model.

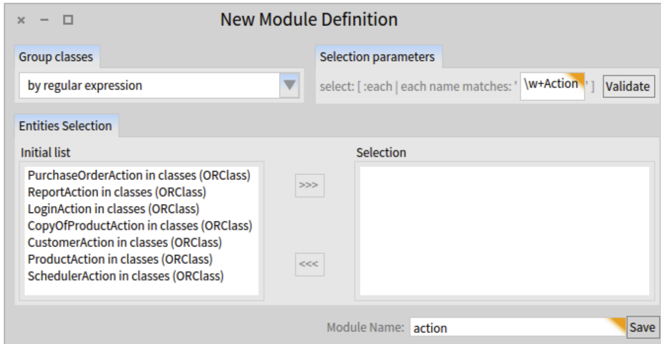


Fig. 4. ORIONPLANNING’s model definition browser

Finally, a dependency rule depends on two logical modules, which are defined in the previous step. Given two modules `A` and `B`, the user can define the following constraints:

- `only A can depend on B`;
- `A can only depend on B`;
- `A cannot depend on B`; or
- `A must depend on B`.

The types of constraints are predefined in the DCL language. Dependencies include access to variables, references to class, invocation to methods, and inheritance to classes. Figure 5 shows the rule definition panel, in which the user selects the source module, the type of constraint, the type of dependency to be analyzed, and the target module. In this example, the user defined that all `Action` classes cannot inherit from classes in the original monolithic package (named `classes`).

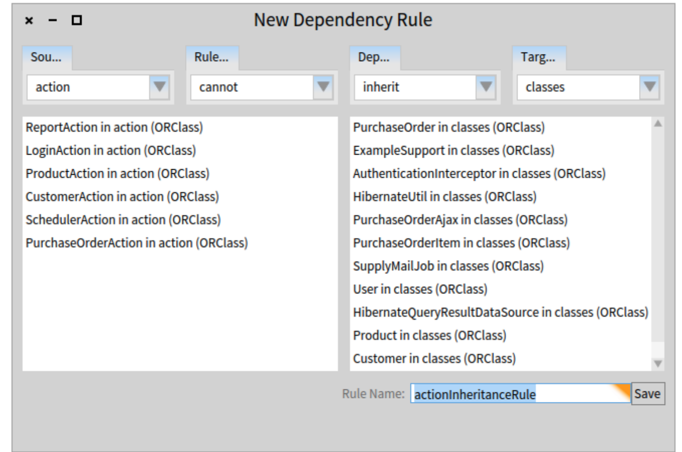


Fig. 5. ORIONPLANNING’s dependency constraint browser

In order to check the conformance between a model and the defined constraints, ORIONPLANNING queries all the dependencies in the current model. The dependency checker analyzes each dependency with each constraint. Violations to the constraint are highlighted in the visualization with a red color (see Section III in our concrete example). Finally, the dependency checker listens to changes in the current model and checks all the dependencies when a change is performed.

III. EVALUATION

To illustrate the use of ORIONPLANNING, we use a simple e-commerce system, called MYWEBMARKET. This system was created independently by another research group to illustrate a case of architectural erosion and the analysis of architectural violations [12]. This system was developed in a sequence of versions. The first one follows a very naive implementation and successive versions improved the modularization to correct specific dependency constraints.

In our evaluation, we imported the first version of MYWEBMARKET, consisting of only one package and performed in ORIONPLANNING the changes that had been made on the actual system (*i.e.,* directly in the code, without any form of model-based support). We observed a sequence of refactorings steps repeatedly applied on MYWEBMARKET that would therefore be interesting to perform using ORIONPLANNING. Listing I presents an informal description of these refactorings. The goal was to isolate the dependencies to a framework (*e.g.,* HIBERNATE) into a new package. Furthermore, it was decided to use the Factory design pattern.

During the replication study, ORIONPLANNING had the limitation of not supporting the Extract Method refactoring (line 6). This limitation is due to the fact that ORION does not yet handle model information at the level of statements (*i.e.,* using an Abstract Syntax Tree). Such support is work in progress [14]. We intend to include finer granularity refactorings in both ORION and ORIONPLANNING. Despite this limitation, all of the other refactorings were performed successfully.

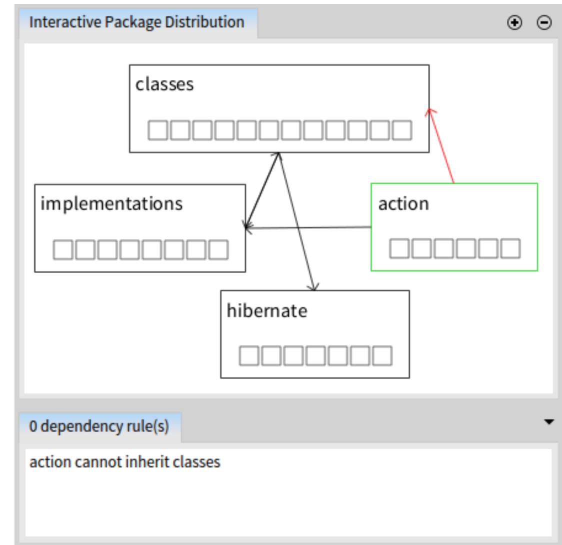
LISTING I
SYSTEMATIC REFACTORINGS IN MYWEBMARKET'S REMODULARIZATION (A PACKAGE *PHib* WAS CREATED TO HOLD DEPENDENCIES TO HIBERNATE, A FACTORY CLASS *FHib* WAS CREATED IN *PHib*)

For each class $C \notin$ package *PHib* that depends on Hibernate

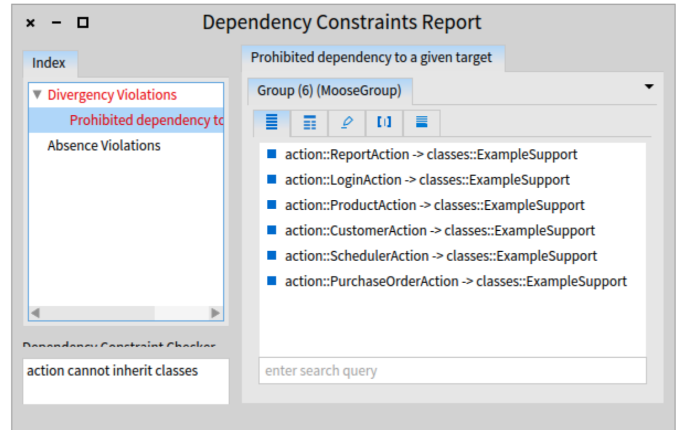
1. create an interface *IC'* in *PHib*
2. create a class *C'* in *PHib* implementing *IC'*
3. Create a method "*public C' getC'()*" in the factory *FHib*
4. $\exists$ method M in C
5. and $\exists$ S statements $\in M$ creating the dependence on Hibernate
6. extract statements S to a new method *M'* in *C'*
7. replace statements S by a call *FHib.getC'().M'()*

Additional to the replication study, we defined a dependency constraint on the resulting model. We performed the constraint definition discussed in Section II-D, *i.e.,* prohibit `Action` classes to inherit non-`Action` classes. In order to simplify the visualization, we previously moved all the `Action` classes to a new package, named `action`.

Figure 6 shows the dependency analysis in ORIONPLANNING in two views. The first view (Figure 6.a) shows the package visualization in which the edge between the packages `action` and `classes` is red. This property means that this dependency is a constraint violation. The second view (Figure 6.b) shows the list of constraint violations in the version under analysis. In this case, all of the `Action` classes extend a common class, named `ExampleSupport`, which does not follow the name convention. In order to fix this violation, the user shall move `ExampleSupport` to the `action` package, and optionally change the class name to the `*Action` name convention.



(a) Constraint violation in package visualization



(b) Violations List

Fig. 6. ORIONPLANNING's dependency rules visualization

IV. DISCUSSION

In this section, we evaluate ORIONPLANNING according to previous work on industrial needs on architectural languages (see Section I) as follows:

- *Tool Support:* our tool is currently a prototype and it provides support to (i) generate a new architecture from scratch; and (ii) modify an existing architecture by refactorings;
- *Iterative Architecting:* our tool accepts that the architecture definition might be incomplete. It provides an environment for incremental changes;
- *Analysis:* our tool provides support for inspection of any model entity at any time. It also provides support to define modules, dependency constraints between them and also constraints based on metrics. Constraints can be evaluated on any model after a change;
- *Multiple Architectural Views:* currently, ORIONPLANNING relies on the visualization of structural dependencies in an explorative view through packages, classes, and methods. There is work on progress to visualize module definitions (see Section II-D) and conceptual relationships between model entities based on their vocabulary;
- *Versioning:* the tool allows the user to create child versions and modify them separately, without creating copies of the working model;
- *Graphical Syntax:* both architecture definition and refactoring operations are performed in a graphical and interactive visualization;
- *Well defined Semantics:* Such semantics are difficult to achieve because they depend on specific project needs. In ORIONPLANNING, we focus on the structural (modular) quality of the code. We recommend that extra-functional aspects of the architecture shall be put into discussion with the stakeholders.

V. RELATED WORK

A considerable amount of tools for architectural description, software visualization, and architectural conformance have been proposed in the literature in the past decades. The survey to which this work is inspired cites architectural languages such as UML and AADL [9]. However, most of the proposed

tools lack proper analysis and they do not provide extension in order to add features specific to practitioners' needs.

In this section, we selected work on architectural languages that applies to two main conditions: (i) these work must provide a tool to support architecture description, and (ii) they also must provide analysis on the proposed architecture.

That et al. [13] use a model-based approach to document architectural decisions as architectural patterns. An architectural pattern defines architectural entities, properties of these entities, and rules that these properties must conform. The approach provides analysis by checking the conformance between an existing architecture definition and a set of user-defined architectural patterns. In ORIONPLANNING, the dependency constraints are similar to the definition of an architectural pattern. However, the architecture definition in our tool is not limited to conform to an architectural decision, *i.e.,* our approach is extensible enough to provide other types of analysis and model transformation.

Baroni et al. [1] also use a model-based approach and extend it to provide semantic information. With assistance of a wiki environment, additional information is automatically synchronized and integrated with the working model. The analysis consists in checking which architectural entities are specified in the wiki. One critical point of this approach is that the information might be scattered in different documents, which can be difficult to maintain. In ORIONPLANNING, both the working model and the tool itself can be easily extended. In fact, the dependency constraint checking (see Section II-D) extends (i) the model to define logical modules, and (ii) it also extends the visualization to define and automatically check constraint violations.

VI. CONCLUSION

In this paper, we presented ORIONPLANNING, a prototype tool for iterative architecture description. ORIONPLANNING relies on FAMIX meta-model to describe the architecture as a set of packages and logical modules. Changes to the resulting model are performed by user-interface refactoring operators. We compared the tool with a survey on practitioners' needs in architecture languages. We showed the extensibility of the tool by implementing a conformance checking extension based on dependency and metric constraints.

We also evaluated the tool with a concrete example which is specific to software evolution. Our study showed the potential of ORIONPLANNING to perform model and analysis-based transformations. In this example, we were able to (i) replicate most of the evolution scenario, (ii) define and check dependency constraints, and (iii) observe refactoring patterns that can be replicated to other cases (the HIBERNATE case).

Future work include modifications to the usability of the tool. For example, we would like to make some actions more intuitive by using *drag-and-drop* instead of menus. Further work also include generating partial source code from an ORION model. The goal is to generate code snippets, following the assumption that the architecture might not be fully described. However, the snippets would have enough information for developers to further complete them. An important improvement would be to include Abstract Syntax Tree modeling to ORIONPLANNING, in order for it to handle more fine-grained operators (*e.g.,* Extract Method).

REFERENCES

[1] Alessandro Baroni, Henry Muccini, Ivano Malavolta, and Eoin Woods. Architecture description leveraging model driven engineering and semantic wikis. In *11th Conference on Software Architecture*, pages 251–254, 2014.

[2] Muhammad U. Bhatti, Nicolas Anquetil, and Stéphane Ducasse. An environment for dedicated software analysis tools. *ERCIM News*, 88:12–13, 2012.

[3] Jan Bosch. Architecture challenges for software ecosystems. In *4th European Conference on Software Architecture*, pages 93–95, 2010.

[4] Paul C. Clements. A survey of architecture description languages. In *8th International Workshop on Software Specification and Design*, pages 16–, 1996.

[5] Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Research report, 2011. URL https://hal.inria.fr/hal-00646884.

[6] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6): 17–26, 1995.

[7] Rich Hilliard and Tim Rice. Expressiveness in architecture description languages. In *3rd International Workshop on Software Architecture*, pages 65–68, 1998.

[8] Jannik Laval, Simon Denier, Stéphane Ducasse, and Jean-Rémy Falleri. Supporting simultaneous versions for software evolution assessment. *Journal of Science of Computer Programming*, 76 (12):1177–1193, 2011.

[9] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, 2013.

[10] Moose. Importing and Exporting MSE Files. http://www.themoosebook.org/book/externals/import-export/mse, 2010. Accessed: 2015-05-30.

[11] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17 (4):40–52, 1992.

[12] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. Recommending refactorings to reverse software architecture erosion. In *16th European Conference on Software Maintenance and Reengineering*, pages 335–340, 2012.

[13] Minh Tu Ton That, S. Sadou, and F. Oquendo. Using architectural patterns to define architectural decisions. In *Conference on Software Architecture and European Conference on Software Architecture*, pages 196–200, 2012.

[14] Yuriy Tymchuk. Extending FAMIX metamodel to generate ASTs for Java and Smalltalk applications. Master's thesis, National University of Lviv, Ukraine, 2013.

[15] Pengcheng Zhang, Henry Muccini, and Bixin Li. A classification and comparison of model checking software architecture techniques. *Journal of Systems and Software*, 83(5):723–744, 2010.