Safe Dynamic Reconfiguration of Concurrent Component-based Applications

Salman Farhat, Simon Bliudze, Laurence Duchien

Safe Dynamic Reconfiguration of Concurrent Component-based Applications

Salman Farhat
Univ. Lille
Inria
CNRS
Centrale Lille
UMR 9189 CRIStAL
F-59000 Lille, France
salman.farhat@inria.fr

Simon Bliudze
Univ. Lille
Inria
CNRS
Centrale Lille
UMR 9189 CRIStAL
F-59000 Lille, France
simon.bliudze@inria.fr

Laurence Duchien
Univ. Lille
CNRS
Inria
Centrale Lille
UMR 9189 CRIStAL
F-59000 Lille, France
laurence.duchien@inria.fr

***Abstract*—Cloud computing and cyber-physical systems involve software capable of adapting at run time to remain compliant with user demands and environmental constraints. This calls for extending the life cycle of software systems with a reconfiguration step to go beyond analysis, design, development and deployment. Existing approaches compute a new valid configuration at design time, at run time, or both, inducing computational or validation overheads for each reconfiguration step. We propose an approach that relies on variability models to acquire a representation of the set of valid configurations of a system. We use feature models to automatically generate a JavaBIP run-time variability model. The generated model monitors and controls the application behaviour by intercepting reconfiguration requests and executing them in such a manner as to ensure that all reachable configurations are valid without the need of pre-computing the possible configurations neither at design time nor at run-time while only inducing a minimal run-time computational overhead.**

***Index Terms*—Distributed Systems, Concurrent Component-based Systems, Variability Models, Self-Configuration, Dynamic Reconfiguration**

I. Introduction

In order to cope with new user requirements, most run-time environments such as cloud computing or cyber-physical systems must adapt in response to a change in the requirements or the operating context [1].

In the context of cloud applications, we consider the system configuration to be sets of resources that host an application. These are constrained by the set of rules that define the coordination and dependencies among the components of the system. A reconfiguration [2] is a process that enables adapting the system configuration in response to a change in the platform or user needs.

There are several approaches for analyzing and planning reconfiguration. These approaches rely on different tools to calculate or validate the appropriate adaptation plans for applying the reconfiguration step. Existing approaches compute a new valid configuration at design time, at run time, or both, inducing computational or validation overheads for each reconfiguration step.

Feature modeling (FM) is a widely used approach to capture commonalities and variability across software systems that are part of a product line or system family [3], [4]. A feature model is usually depicted as a tree diagram whose nodes represent features that can be selected to build a product.

In this paper, we present an approach that leverages the feature model for acquiring a representation of a set of valid configurations of a system which will be used to generate a component-based run-time variability model. This model will be used to monitor and control the application at run-time. It will intercept reconfiguration requests and execute them while ensuring by construction that all intermediate configurations reached are valid without the need of computing the configurations neither at design time nor at run time.

By exploiting feature models and component-based run-time formal models such as JavaBIP model [5], we work on the means of enforcing safe behavior of concurrent component-based systems by construction through automatic derivation of executable models from feature models. A system is safe if any state reached is valid.

II. Motivation

Our main objective is to define a component-based run-time variability model encoding the semantics of a feature model to control reconfiguration operations while ensuring the following safety property: only valid configurations can be selected as a result of any reconfiguration request.

Several approaches [6]–[15] exist for analyzing and planning reconfiguration. These approaches either compute the set of all possible configurations in advance at the design time or a new valid configuration at run time. This induces computational or validation overheads for each new configuration, whenever reconfiguration is needed. Indeed, when all valid configurations are precomputed at design time, they must be stored explicitly in the runtime. This is problematic, since the set of configurations is exponential in the number of features, but particularly so for distributed systems, where a copy of the list has to be stored at every node. Alternatively, the new configuration must be computed and validated at run time, inducing a computational overhead.

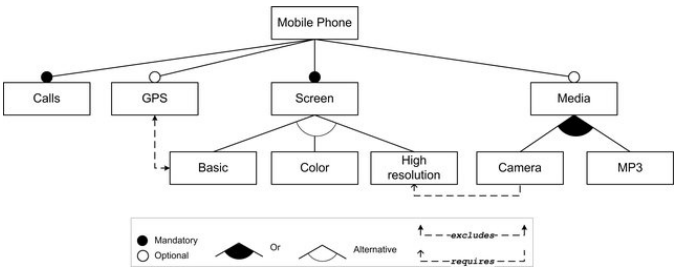

Fig. 1: Mobile feature model

The solution presented in this paper leverages feature models to generate a component-based run-time variability model encoding all valid configurations to drive the reconfiguration process. Furthermore, our solution removes from the user the burden of ensuring that the reconfiguration steps are carried out in the correct order.

III. Running example

We use the example of a software system used to control the hardware for a line of mobile phones. The structure of the software system reflects that of the mobile phones. All mobile phones in the product line must provide the basic functionality: the capability of making and receiving phone calls and a screen allowing the users to interact with the phone. In addition, the users can request additional optional features, namely, the GPS and Media modules. Naturally, the users can choose between cheaper, low-end models, equipped with a basic screen, mid-range models with a color screen, and more expensive models equipped with a high-resolution screen. The media capabilities of the phone can include a camera and an MP3 component to play music. The use of the camera requires the phone to be equipped with a high-resolution screen. Furthermore, low-end models do not have a GPS module.

IV. Foundations

A. Feature Models

Feature Models (FMs) [3] are regarded as the de-facto standard for managing variability and are commonly used for modeling the variability of software product lines. The possible configurations of a mobile phone are defined by the feature model shown in Figure 1. A FM defines variability in terms of features and their relationships. Features are arranged in a tree-like structure, where every node represents a mandatory or an optional feature. Furthermore, sub-features of a feature can form an Alternative-group or an OR-group. In addition to such structural constraints, FMs define two types of integrity constraints among features: a feature X can *exclude* or *require* another feature Y. As an example, the mobile phone variability model is shown in Figure 1.

A *valid configuration* is a set of features that satisfies all the constraints of a feature model. The 14 valid and complete configurations of the FM in Figure 1 are shown in Table I.

TABLE I: Valid configurations for the FM of Figure 1

```
C1:  {Mobile phone, Calls, Screen, Basic}
C2:  {Mobile phone, Calls, Screen, Color}
...  {...}
C14: {Mobile phone, Calls, Screen, High Resolution,
     Media, Camera, Media, MP3, GPS}
```

B. JavaBIP

JavaBIP [5] is an open-source Java implementation of the BIP (Behaviour-Interaction-Priority) [16] mechanism for the coordination of concurrent components. The behaviour of components is defined as Finite State Machines (FSMs). Transitions of these FSMs are labeled by *ports*, which are used to specify the possible interactions with other components: an *interaction* is a set of component ports that can be synchronised, i.e. executed together atomically. Graphically, allowed interactions are defined by *connectors*. The behaviour specification of each component along with the set of connectors are provided to the *BIP engine*. The engine orchestrates the overall execution of the system by deciding which component transitions must be executed at each cycle.

In JavaBIP, transitions of FSMs defining component behaviour specifications can be of three types: *enforceable*, *spontaneous* and *internal*. Only enforceable transitions are controlled by the engine. Spontaneous transitions are used to take into account changes in the environment and, therefore, they are not announced to the engine but executed upon notification from the environment of the component. Finally, internal transitions allow behaviour specifications to update their state based on internal information—when enabled, they are executed immediately. Spontaneous and internal transitions cannot be used for synchronisation with other components.

Figure 2 shows a JavaBIP model with three components: `GPS`, `Screen`, and `Camera`. Enforceable, spontaneous, and internal transitions are shown by solid black, dashed green and solid red lines, respectively. Ports are shown as grey boxes on the sides of the components. Two connectors—black lines connecting the ports—define the possible interactions. We do not show connectors for singleton interactions. Thus, unconnected ports, e.g. `High_Resolution_reset`, can fire alone, whereas the port `GPS` can only be fired together with the port `not_Basic`. Since there is no transition from the state `Basic` labeled `not_Basic`, this prevents the `GPS` component from entering the state `GPS` when the `Screen` component is in state `Basic`.

V. Design and Transformation

The creation of the component-based variability model elements will be based on the feature model. A feature will be analyzed and transformed into a component depending on its nature. As an example, in Alternative-group only one feature can be selected at a time. Thus, if the feature `X` is parent of an Alternative-group (e.g. `Screen`) then a component with name `X` is created and all sub-features are states in the FSM of the component `X`. To this end, a FSM with sub-features for component `X` is created which allows only one state to

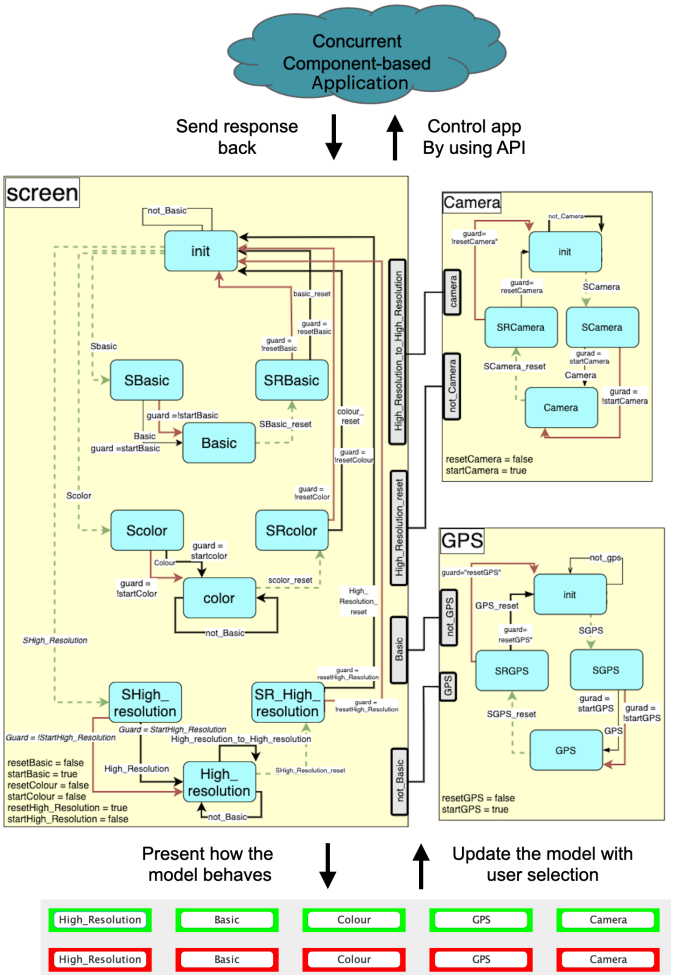

Fig. 2: Component-based runtime variability integration

be active, thus, an alternative choice is enforced (FSM has only one active state at a time). In addition, two intermediate states for each sub-feature are created that are used to support requesting the selection (e.g. `SBasic`)/deselection (e.g. `SRBasic`) of feature `Y` and avoid direct selection/deselection of feature `Y`.

The internal specification of the component generated corresponding to feature `X` will be generated based on the nature of the feature `X`. After then, the transitions between states will be established. Finally, after the creation of the components and their internal behaviour, the generation of the coordination layer will be based on the integrity constraints and structural constraints that exist in the feature model. Constraints will be translated into connectors to set up the coordination layer in between the components so that the component-based runtime variability model proceeds in a safe manner while applying reconfigurations at runtime. As an example for integrity constraints such as exclude constraint, `Basic` feature excludes `GPS` feature this means that only one feature in between these two features can be in a valid configuration. Thus, component `Screen`, should be synchronized with component `GPS` so that `Basic` feature can be selected only if `GPS` component is

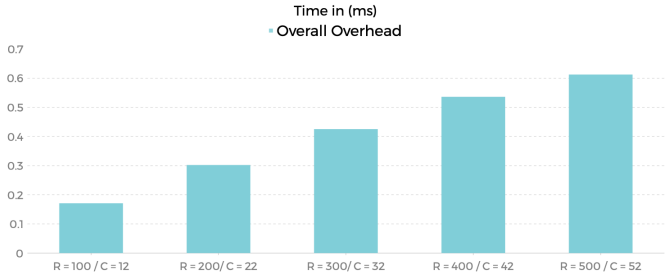

Fig. 3: JavaBIP overhead over the application

not in state `GPS` which means that `GPS` feature is not active. Thus, constraints in the feature model will be encoded in the generated component-based runtime variability model.

As shown in the bottom and top parts of Figure 2, the generated model intercepts reconfiguration requests using the UI and executes them by calling the APIs associated with the transitions in the components. Due to space constraints in the short paper, Figure 2 is a sub-part of the generated model for the feature model presented in Figure 1 and we did not present the design decisions underlying the transformation model. Check out [1] for additional details on the design.

VI. Implementation and Validation

We implemented our model transformation tool using ATLAS Transformation Language (ATL). ATL is a domain-specific language for specifying model-to-model transformations, it provides ways to produce a target model that conforms to a target metamodel from a source model conforms to a source metamodel [13]. The generated model specification will be conforming to the JavaBIP metamodel [17]. The generated XML file will be parsed using the DOM library in Java to generate the JavaBIP specification and the glue coordination. Our implementation can be found on zenodo platform [18].

For the experiments, we generate random feature models with 100, 200, 300, 400, and 500 features using BeTTy tool [19]. Then we measure the overhead of the component-based runtime variability model generated on the application. Figure 3 shows the overhead of the generated model in terms of time in milliseconds where `R` represents the number of features in the randomly generated feature model and `C` represents the number of components created for the random feature model.

Compared to approaches listed in section II, the overhead produced by the generated component-based run-time variability model on the application is proven to be minimal.

VII. Related Work

In this section, we present different engineering approaches that deal with reconfiguration and self-adaptation of systems.

a) Model-based approaches: The model describes system specifications that include information about system architecture, environment, and reasoning on the adaptation. These models are used to extract and calculate a reconfiguration

[1] salmanfarhat1.github.io/salman-farhat/#research

plan/self-adaptation plan to keep the system compliant to the user expectation and context constraints. Cetina et al. [6], [7] propose a model-based approach that is based on SPL. A feature model (FAMA [20]) is used to specify commonalities and variabilities of system functionality. A set of conditions and resolutions are defined at the design time.

b) Component-based approaches: Component-based techniques are a subset of model-based approaches in which the system description is made up of components. Connectors control the relationships between the components and establish interaction and dependencies between them. Concerto [21] is a component-based reconfiguration model focusing on modelling and coordinating the life-cycle of interacting parts of a system. Typically, each module of the system is modeled with a component type. A control component type is developed by the component developers and this component contains information about the life-cycle of the component and its dependencies. Internal parallelism is expressed inside a component in which multiple places may be active, and multiple transitions may be fired simultaneously. Concerto is equipped with a reconfiguration language that is used by system administrator to modify the architecture.

c) Service-oriented approach: Services are lightweight, encapsulated, and autonomous software units that execute a particular function. Services are used to support the creation of rapid, low-cost, and large distributed systems [22]. Cloud Integrator [23] is a service-based approach for managing services in multi-cloud environments. Cloud Integrator works as a mediator between the service providers and the clients, it is used for composing, executing, and managing services provided by different cloud computing platforms.

Our approach differs from the existing approaches by minimizing the computational overhead in-terms time at runtime. The generated model will ensure by construction that only valid configurations can be reached.

VIII. Conclusion and Future work

In this paper, we worked on the means of enforcing the safe behaviour of concurrent component-based systems by construction through automatic derivation of executable models from a variability model. The generated model will drive the reconfiguration process without the need of pre-computing the configurations neither at run time nor at design.

To go further, the feature models that are currently supported in our work cannot fully represent cloud-like platforms. These require taking into account complex attributes and constraints, such as time constraints and temporal constraints [24].

References

[1] M. Aksit and Z. Choukair, "Dynamic, adaptive and reconfigurable systems overview and prospective vision," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE, 2003, pp. 84–89.

[2] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, vol. 17, pp. 184–206, 2015.

[3] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1990.

[4] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela, "Software diversity: state of the art and perspectives," 2012.

[5] S. Bliudze, A. Mavridou, R. Szymanek, and A. Zolotukhina, "Exogenous coordination of concurrent software components with JavaBIP," *Software: Practice and Experience*, vol. 47, no. 11, pp. 1801–1836, 2017.

[6] C. Cetina, J. Fons, and V. Pelechano, "Applying software product lines to build autonomic pervasive systems," in *2008 12th International Software Product Line Conference*. IEEE, 2008, pp. 117–126.

[7] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Autonomic computing through reuse of variability models at runtime: The case of smart homes," *Computer*, vol. 42, no. 10, pp. 37–43, 2009.

[8] R. Abid, G. Salaün, and N. De Palma, "Formal design of dynamic reconfiguration protocol for cloud applications," *Science of Computer Programming*, vol. 117, pp. 1–16, 2016.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[10] T. A. Lascu, J. Mauro, and G. Zavattaro, "A planning tool supporting the deployment of cloud applications," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. IEEE, 2013, pp. 213–220.

[11] J. Zhang and B. H. Cheng, "Model-based development of dynamically adaptive software," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 371–380.

[12] M. Chardet, H. Coullon, D. Pertin, and C. Pérez, "Madeus: A formal deployment model," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2018, pp. 724–731.

[13] C. Quinton, D. Romero, and L. Duchien, "Saloon: a platform for selecting and configuring cloud environments," *Software: Practice and Experience*, vol. 46, no. 1, pp. 55–78, 2016.

[14] R. Di Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro, "Aeolus: A component model for the cloud," *Information and Computation*, vol. 239, pp. 100–121, 2014.

[15] C. Quinton, M. Vierhauser, R. Rabiser, L. Baresi, P. Grünbacher, and C. Schuhmayer, "Evolution in dynamic software product lines," *Journal of Software: Evolution and Process*, vol. 33, no. 2, p. e2293, 2021.

[16] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the BIP framework," *IEEE software*, vol. 28, no. 3, pp. 41–48, 2011.

[17] A. Mavridou, J. Sifakis, and J. Sztipanovits, "DesignBIP: A design studio for modeling and generating systems with BIP," *arXiv preprint arXiv:1805.09919*, 2018.

[18] Zenodo, "Component-based Runtime Variability Model for Safe Dynamic Reconfiguration," Nov. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5680389

[19] S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, and A. Ruiz-Cortés, "BeTTy: benchmarking and testing on the automated analysis of feature models," in *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, 2012, pp. 63–71.

[20] D. Benavides, P. Trinidad, A. Ruiz-Cortés, and S. Segura, "Fama," in *Systems and software variability management*. Springer, 2013, pp. 163–171.

[21] M. Chardet, H. Coullon, and S. Robillard, "Toward safe and efficient reconfiguration with concerto," *Science of Computer Programming*, vol. 203, p. 102582, 2021.

[22] M. Zhou, R. Zhang, D. Zeng, and W. Qian, "Services in the cloud computing era: A survey," in *2010 4th International Universal Communication Symposium*. IEEE, 2010, pp. 40–46.

[23] E. Cavalcante, T. Batista, F. Lopes, A. Almeida, A. L. de Moura, N. Rodriguez, G. Alves, F. Delicato, and P. Pires, "Autonomous adaptation of cloud applications," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2013, pp. 175–180.

[24] G. Sousa, W. Rudametkin, and L. Duchien, "Extending dynamic software product lines with temporal constraints," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2017, pp. 129–139.