Component-based Runtime Variability Model for Safe Dynamic Reconfiguration

***Abstract*—Design and Implementation**

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

I. Design and Transformation

In this section, we show how we used the input feature model to construct a component-based run-time variability model. We walk you through the construction of a model transformation tool step by step. The feature model will be sent into the model transformation tool, which will build a component-based run-time variability model. The semantics of the feature model, which represents the similarities and variabilities in the configuration of a certain domain, will be added to the created model. The model transformation tool and the integration of the resulting model with the application are shown in Figure 1.

Recent work uses feature-oriented analysis to allow the automatic selection of a valid configuration for a cloud provider to facilitate the deployment of an application [1], [2]. In contrast to these approaches that calculate the new configuration at each time configuration must change to keep application complaint to user and context requirements, our approach is to transfer the semantics of the feature models into a component-based runtime variability model. The generated model will be running alongside the application and aims to control the application while guaranteeing safe behavior in executing reconfigurations at runtime. Our approach avoids the overhead of computing a new valid configuration at both design and runtime for each time reconfiguration is required.

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 presented in Section I. The states of the component generated corresponding to feature `X` will be generated based on the nature of the feature `X`. After then, as we will see in the next section, the transition between states will be established in a static manner. Finally, after the creation of the components and their internal behaviour, the coordination layer will be generated based on the propositional logic constraints and hierarchical constraints contained in the feature model. The transformation design overview will be presented by first describing component creation, then explaining component internal behavior creation, and lastly explaining the coordination layer generation.

A. Creation of components

A component is a software object, encapsulating certain behaviors of a software element. On the other hand, features are elements in a feature model that represent a certain system functionality that can be chosen by the user to be part of the configuration for a certain domain. Thus a mapping between features and components is introduced. Components are created depending on the nature of the feature. Creation of component for a feature depends on its type.

a) Not leaf and parent of alternative group: Consider the `Screen` feature. `Screen` feature is parent of an exclusive sub-features. Since only one feature will be selected from the `Screen's` sub-features then only one component will be created and all the sub-features will be states in the component created (internal behaviour including states and transitions creation is discussed in next part). Thus, forcing only one feature from the `Screen's` sub-features to be active at a time (only one state can be active in the component at a time), as not to allow the selection of more than one features for the `Screen's` sub-features.

b) Leaf and part of an or-group: Consider the `Camera` feature. `Camera` is part of and Or-group, which means that it can be selected based on the user's need. Thus, a component is created for `Camera` in which this component can be in a state where `Camera` is selected according to user's need.

c) Leaf and optional or mandatory: Consider the `GPS` Feature. Since `GPS` is an optional feature, it may be selected or deselected based on the user's need. As a result, a component for `GPS` is created, and this component can be in a state where `GPS` is selected according to the user's need.

Therefore, if feature `X`, satisfies one of the previous three conditions, then a component with `X` name will be created.

B. Internal behaviour of components

Following the description of component creation, the internal behavior of the component will be detailed. Component behavior will be represented by finite state machines (FSMs). A FSM has a finite set of states and a finite set of transitions between these states. The states for the generated component `X` are generated based on the feature `X` and it's sub-features if exist. Therefor, for a generated component `X` corresponds to the feature `X`, the generated states will be as follows, if X is:

a) Not leaf: As presented in Figure 2a, `Screen` feature is parent of an alternative-group which means only one feature from `Screen's` sub-features should be selected. Therefore, to force only one feature from the sub-features to be selected at a time all the sub-features (`Basic, Color,`

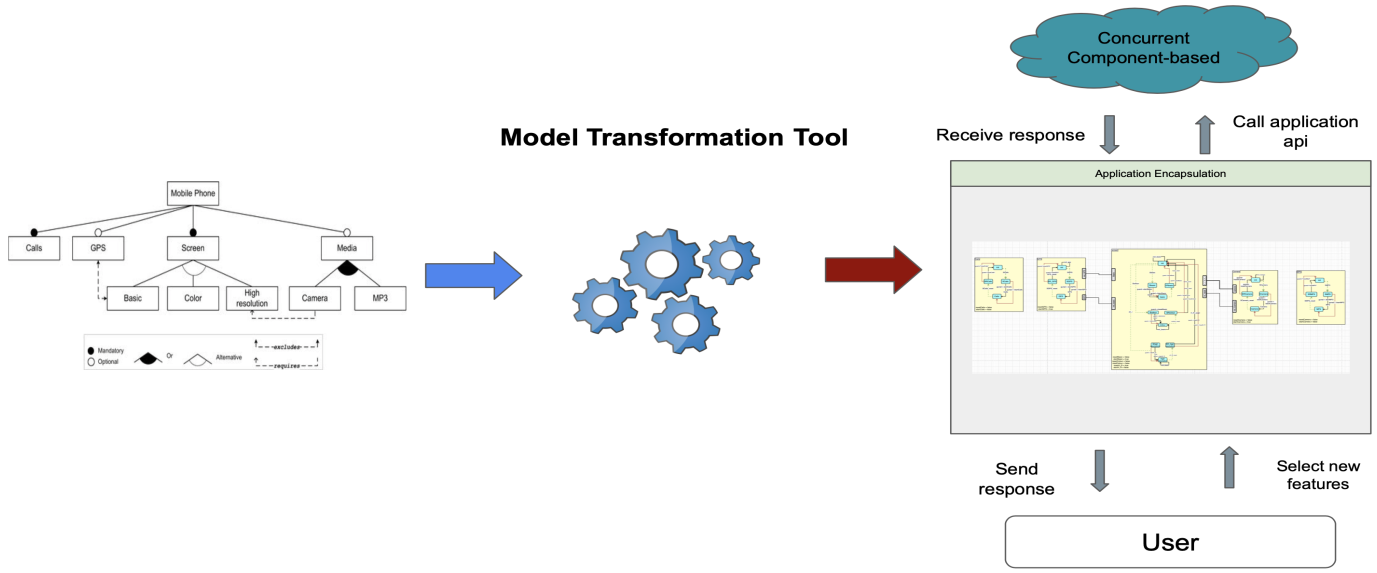
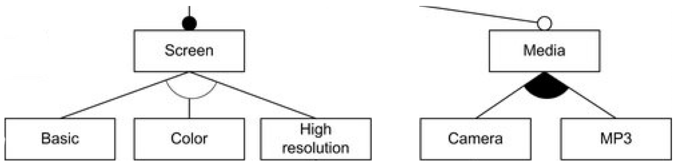


Fig. 1: Component-based runtime variability integration



(a) The `Screen` feature is the parent of an alternative group

(b) The `Media` feature is the parent of an OR-group

Fig. 2: Alternative and OR-groups of an FM

`High Resolution`) will be states in the generated `Screen` component. This design allows only one feature to be selected from the sub-features since in a finite state machine only one state will be a active at a time. For each sub-feature, three state will be created (e.g. for `Basic` feature three states are created: `Basic`, `SBasic`, `SRBasic`).

The intermediate states (`SX`, `SRX`) for feature `X` are used to prevent immediate activation or deactivation of feature `X` after receiving the user requests to activate or deactivate feature `X`. Thus, if the user requests to activate feature `X` the component that receives the request will move to the intermediate state `SX` and then examine if it is possible to proceed to activate feature `X` based on the coordination layer constructed using the constraints enhanced with dependencies. The same is applicable for turning off feature `X`.

b) Leaf: As shown in Figure 2b, `Camera` feature is a leaf and corresponds to the `Camera` component created (as discussed previously). In this case, since camera is a leaf then it is the only option in the component `Camera` (`Camera` feature has no sub-features), then the states created are the `Camera`, `SCamera`, and `SRCamera`. The Intermediate states are used for the same purpose as discussed for a non leaf feature. In general the states created for a leaf feature `X` corresponds to component `X` is presented in the following pseudo code.

After the creation of the states, transitions between the states are established. There are three types of transitions supported between the states enforceable, spontaneous, and internal as discussed in **??**. Transitions are associated with calls to methods, which allows the model to control the application as illustrated in Figure 3, such as:

c) Spontaneous transitions are generated as follows:

- From state *`init`* to *SX*: this spontaneous transition is used by external users to inject external commands to request the selection of new feature *X* to the configuration.
- From state *X* to state *SRX*: this spontaneous transition is used by external users to inject external commands to request deselection of feature *X* from the configuration.

d) Internal transitions are generated as follows:

- From state *SX* to state *X*: this internal transition is used to update the component behaviour according to internal information such as guards. Thus, if the guard evaluate to true, then the component behavior state will move to the state *X* and thus calls the method associated to the transition to select feature *X*. The internal transition will be executed if feature *X* has no integrity constraints nor hierarchical constraints which means that feature X has no dependencies. Thus, feature `X` can be selected and there is no need for the component containing state X to be synchronized with other components for the selection of `X`.
- From state *SRX* to state *`init`*: this internal transition is used to update the component behaviour according to internal information such as guards. Thus, if the guard evaluate to true, then the component behavior state will move to *`init`* state and thus calls the method associated

to the transition to deselect feature X. This internal transition will be used to update state by deselecting feature `X` if there is no dependencies on `X`.

e) Enforceable transitions are generated as follows:

- From state *SX* to state *X*: this enforceable transition is used to update the component behaviour to select feature *X* while being synchronized with other components using ports. The transition will be accompanied by a guard. The guard evaluates to true if the component must be synchronized with other components to select feature `X`. Thus, if feature `X`, has dependencies (e.g. `Basic` feature) then the enforceable transition will be the option to update the component behaviour.
- From state *SRX* to state *init*: this enforceable transition is used to update the component behaviour to deselect feature *X* while being synchronized with other components using ports. The transition will be accompanied by a guard and the guard evaluates to true if there is feature `Y` depends on feature `X`. Thus, feature `X`, must not be deselected unless feature `Y` is deselected first.

A Boolean guard will be associated to the enforceable and internal transition that leads to select/deselect a feature X. The transition is only enabled if the data provided by the components satisfy the guard. Guards are generated according to the integrity constraints and the hierarchical structure of the feature model. In which, if feature X need to be synchronized with other components in order to be selected/deselected then the guard associated to the enforceable transition will evaluate to true, while if feature X doesn't need to be synchronized with other components then the internal transitions guard will evaluate to true and the component behaviour will be updated according to the enabled internal transition. Thus the guard for each state in the component is created based on the information acquired from the input feature model.

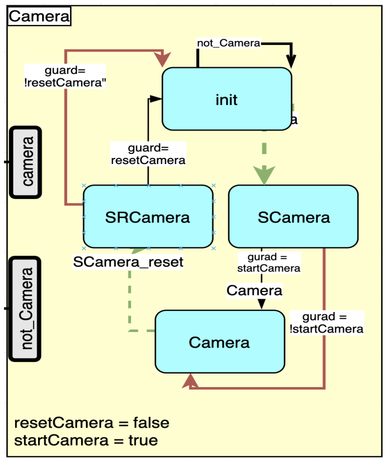


Fig. 3: Generated Camera component

Figures 3 and 4 show an overview of a component `Camera` that corresponds to the leaf feature and for the component `Screen` that corresponds to a non-leaf feature.

For the sake of simplicity, we present a transformation of the `Screen` feature. `Screen` feature is a parent feature of an alternative-group, which means only one sub-feature (`Basic`, `Color`, `High Resolution`) can be chosen for a configuration. The component `Screen` corresponds to the `Screen` feature will be generated as presented in Figure 4. The generated component is the `Screen` component and the states are the exclusive sub-features (`Basic`, `Color`, `High_resolution`) with their intermediates states. Assume that the user wants to select the `Basic` feature, the user should request the `Basic` feature using the user interface provided and when the model receive the request, the `Screen` component will move to the `SBasic` state. At the `SBasic` state there are two choices either update the behaviour to activate the `Basic` feature according to the internal transition or the enforceable transition. The two transitions are associated with guard (g and not g) and only one of these two transitions will be used according to the internal data. Thus if the guard for starting the `Camera` is true, that means that the component must be synchronized with other components while moving to `Basic` state and if the guard evaluates to false the internal behaviour of the component will be updated by the internal transition. In the next part, we will discuss in details the generation of the coordination layer between the generated components.

C. Coordination layer

After the creation of the internal behaviour of the components, coordination between components will be established. JavaBIP clearly separates system-wide coordination policies from the component behavior. Coordination is applied in an exogenous manner, relying totally on component APIs. Components coordination is defined by means of interaction models, that is, sets of interactions. Interactions are sets of ports that define allowed synchronizations between components. An interaction model is defined in a structured manner by using connector motif. Due to separation of concerns between the behaviour and the coordination layer, we are able to automatically generate connectors that define possible interaction in between components based on the internal behaviour of the components.

Integrity constraints and hierarchical structural constraints exist in the feature model will be translated to connectors motifs to set up the possible interactions needed in between components, so that the component-based runtime variability model proceed in a safe manner while applying reconfigurations at runtime. As an example for integrity constraints such as exclude constraint, `Basic` Feature exclude `GPS` Feature this means that only one feature in between these two features can be in the current configuration. Thus, component `Screen`, should be synchronized with component `GPS` so that `Basic` can selected (state Basic is active) only if `GPS` component is not in state `GPS` which means that `GPS` is not active. To this end, we present the generation of the connector motifs according to the following constraints:

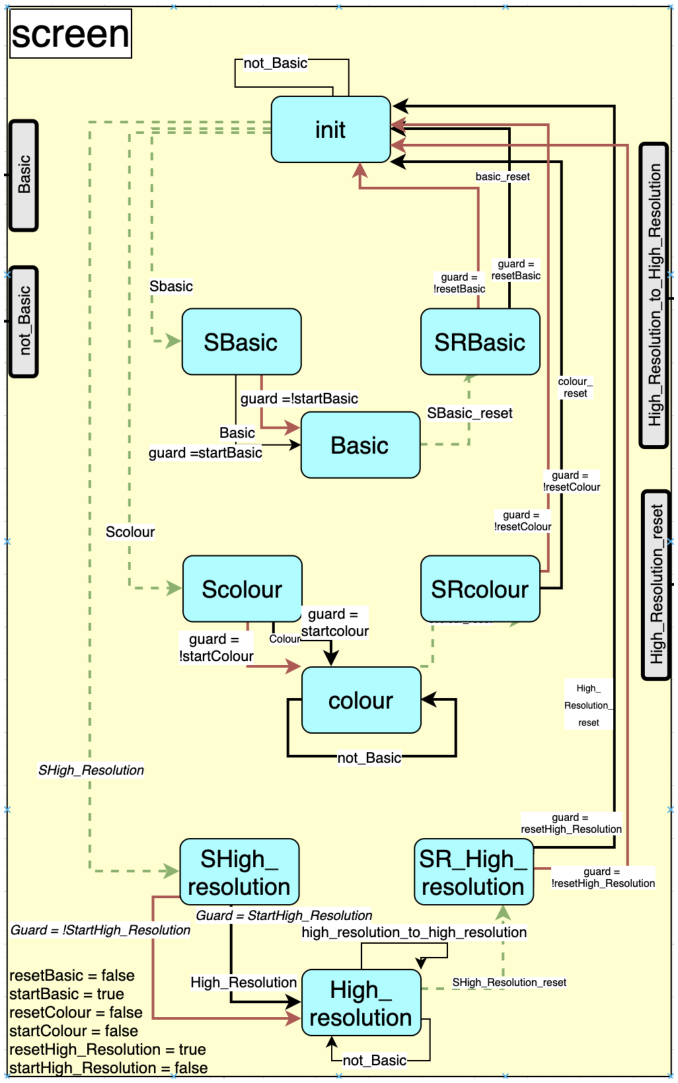


Fig. 4: Generated Screen component

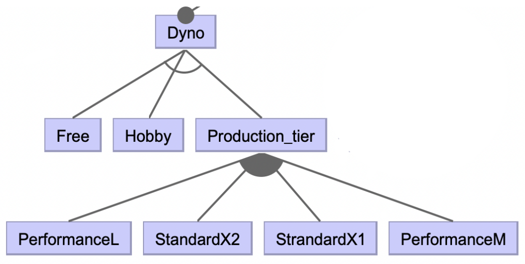


Fig. 5: Part of the Heroku cloud feature model

1) Hierarchical constraints: In feature model, features are arranged in a tree-like structure which induce hierarchical constrains. As an example of hierarchical constraints, Figure 5 is part of the Heroku cloud feature model, it presents `Dynos`, where dynos are isolated, virtualized unix containers, which provide the environment required to run an applications deployed in it. The Heroku cloud feature model will be translated into five components (`Dyno`, `PerformanceL`, `StandardX2`, `PerformanceM`, `StandardX1`), as discussed in previous sections. As indicated in Figure 5, a user can configure the system using one of three options: `Free Dyno`, `Hobby Dyno`, or any mix of `Production_tier Dyno`. The `Production_tier` features (`PerformanceL`, StandardX2, `PerformanceM`, StandardX1) cannot be selected in combination with `Free` or `Hobby` features, which means in a configuration if the user request `Free` or `Hobby` feature then it is not possible to request any feature from the `Production_tier` and vise versa. Thus, connector motifs will be created to manage the coordination between components so that it preserves the hierarchical constraints illustrated in the feature model. Five connector motifs will be created as shown in Figure 6. For selecting a feature from the `Production_tierr` it need to be synchronized with the the activation of the `Production_tierr` feature in the Dyno component (Dyno feature is a parent for an exclusive features thus a component Dyno will be created and all sub-features will be states in it). For deactivation of `Production_tier` feature in the Dyno component it need to be synchronized with all component of `Production_tier` (`PerformanceL`, `StandardX2`, `PerformanceM`, `StandardX1`) to make sure that all of these components are not in the states where features are selected, thus `Production_tier` feature can be deactivated and the user can request a new `Dyno` feature.

2) Exclude Constraints: Exclude constraints in the feature model prevent two features to be in the current configuration at the same time. For example, in mobile phone feature model, `GPS` feature excludes the `Basic` Feature. The transformation design consists of creating two connector motifs as shown in Figure **??**, such as to prevent the user from selecting the two features at the same time. The first connector motif will synchronize the activation (`GPS` transition) of the `GPS` feature with the transition (`not_Basic`) in the Screen component leading to force activation of `GPS` while `Basic` if off. The second connector motif have the same semantics as the first one in which the activation of `Basic` feature will be synchronized with the transition (`not_GPS`) leading to force the activation of `Basic` feature while `GPS` is off.

3) Require Constraints: Require constrainst such as `X requires Y` in feature model enforces the current configuration to have feature `Y` before the activation of feature `X`. For example, in mobile phone feature model, `Camera` feature require `High_resolution` Feature. In brief, the transformation consists of creating two connector motifs for the coordination between components. Since to select `Camera` feature, `High_resolution` feature should be first exist in the configuration, so the first connector motif synchronizes `Camera` activation (`Camera` transition) with the transition (`high_resolution_to_high_resolution` loops on the `High_resolution` state) in the Screen component to ensure that the `High_resolution` feature is already activated, hence, this enforces `High_resolution` feature to be in the configuration before the activation of

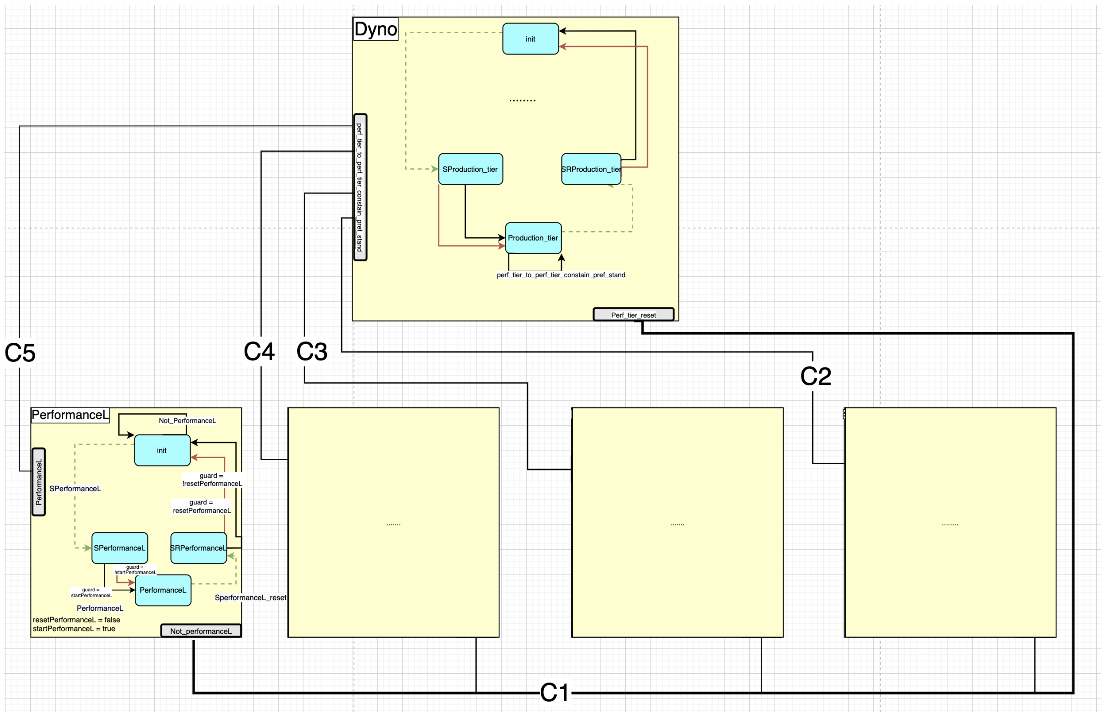


Fig. 6: The generated model of the Heroku cloud feature model

the `Camera` feature. On the other hand, second connector is used for the deactivation of the `High_resolution` feature. `High_resolution` feature should not be deactivated while `Camera` feature is still active. Thus, the deactivation of the `Camera` (transition: `Camera_reset`) will be synchronized with the transition (`not_Camera`) in the Camera component thus prevent the deactivation while `Camera` is active. An overview of the full design will be presented in the appendix.

Because we follow the semantics of the feature model while building and designing the model transformation tool, we will obtain a the component-based runtime variability model that models the variability of the configurations for a concurrent component-based application. Thus, we answer the first research question RQ1 about how to obtain the component-based runtime variability model.

Afterwards, The generated model intercepts reconfiguration requests executed by the user via the user interface and execute them while enforcing the constraints ensuring that all intermediate configurations are safe while avoiding the overhead of calculating a new configuration at each time reconfiguration must occur. The model executes them by calling the application API in the methods associated with transitions in components. Thus, RQ2 is answered and we showed how the model can control the application and how the user can request reconfigurations at runtime.

When a new requirement changes, the formal executable model will guarantee by construction the safety of applying reconfiguration. Thus, if a feature cannot be reached from the current state the model will will hold the execution until it is possible to reach the desired request in which RQ3 is answered.

References

[1] 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.

[2] E. Wittern, J. Kuhlenkamp, and M. Menzel, “Cloud service selection based on variability modeling,” in *International Conference on Service-Oriented Computing*. Springer, 2012, pp. 127–141.