Modeling Variability of Hierarchical Component-Based Systems

Nico Jansen[*], **Jérôme Pfeiffer**[†], **Bernhard Rumpe**[*], **David Schmalzing**[*], **and Andreas Wortmann**[†]
[*]Chair of Software Engineering, RWTH Aachen University, Germany
[†]Institute for Control Engineering of Machine Tools and Manufacturing Units, University of Stuttgart, Germany

ABSTRACT The engineering of hierarchically decomposed component-based systems emphasizes the separation of concerns to reduce development complexity through work distribution and component reuse. Variability further promotes reuse, as system variants may be used in different markets or contexts. However, variability must be adequately managed as it introduces another layer of complexity to system development. Consequently, modeling of hierarchical component-based systems should support composition of variable components while simultaneously facilitating their formal analysis. To address this, we formally define variable component types, propose a modeling language for specifying the variability of hierarchically composed systems, and present a method to check the component variants' well-formedness. We extend the semantically grounded architecture description language MontiArc to realize the modeling of variable component-based systems supporting the well-formedness of variable component types and late binding of variability. The resulting realization of variable component types enables the specification of reusable and flexible components while making customization options explicit in the component interface and maintaining the black-box view of components. This can ultimately reduce complexity in developing variable components and, thus, facilitate the engineering of component-based systems.

KEYWORDS Architecture Description Languages, Software Architecture, Reuse, Variability

1. Introduction

Component-based software engineering (CBSE) (Heineman & Councill 2001) is a software engineering methodology that leverages reusable, off-the-shelf building blocks for software construction. These software components hide their implementation details behind a stable interface that facilitates their composition. Therefore, the individual components are supposed to be easier to reuse and evaluate and hence more mature. The behavior, composition, and communication of such software components require implementing these in some general-purpose programming languages (GPLs), which creates a conceptual gap between the problem domain and the solution domain with the respective GPL (France & Rumpe 2007). Models that describe the complexity of a system at multiple levels of abstraction and from different viewpoints, combined with model analysis and transformation, can bridge this gap. With this, software components are notated as component models that conform to architecture description languages (ADLs) (Medvidovic & Taylor 2000a). They combine the benefits of CBSE and MDSE and have been developed for and applied to multiple challenging domains, including automotive (Debruyne et al. 2004), avionics (Feiler & Gluch 2012), and robotics (Adam, Butting, et al. 2017; Adam, Hölldobler, et al. 2017). Once implemented, component customization and hierarchical decomposition foster model reuse and evolution. Enabling flexible reuse and evolution of components requires an appropriate mechanism to model commonalities and variability. Software product line engineering (SPLE) (Apel et al. 2013) is a software development paradigm that aims to increase software development productivity and the quality of products by increasing the level of reuse. It investigates how to describe commonalities and differences

JOT reference format:
Nico Jansen, Jérôme Pfeiffer, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. *Modeling Variability of Hierarchical Component-Based Systems*. Journal of Object Technology. Vol. 23, No. 3, 2024. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2024.23.3.a9

in software and present these as a product line. Variability has already been explored in ADLs in previous publications (Feiler & Gluch 2012; Debruyne et al. 2004; Haber, Rendel, Rumpe, Schaefer, & van der Linden 2011). However, these approaches require additional external models, a white-box view of components, or aligning product features with the component hierarchy. Thus, these approaches lack flexibility and contradict CBSE, where components are reusable building blocks, and implementation details should remain hidden.

We present a method to define variable component types, extend the MontiArc ADL (Haber et al. 2012; Butting et al. 2017) with modeling elements to specify a component's variability and present a method to access their well-formedness. Our approach differentiates between component type declaration and usage, i.e., configuration of the component type. Declaring a variable component type includes defining the component's variability via explicit feature definitions. These features can be referenced throughout the type declaration using a 150% superset approach to include or exclude different elements of the component. When instantiating such a component type, the selected features are passed as parameters of the instance. By establishing analyses tailored to our method, we ensure the structural well-formedness of variable component models and architectures. This approach offers three major benefits:

- Variable component descriptions can be composed, enabling the reuse of variable components and thus the creation of variable component libraries
- Variable components are configured during component instantiation and can be used in different configurations in the same architecture while maintaining the integrity of defined feature constraints.
- Variable components can be configured across multiple hierarchy levels. Some of a subcomponent's features can be configured on one hierarchy level; others can become part of the feature model of the enclosing component.

In the following, section 3 presents our definition of variable component and connector architectures, section 2 explains the MontiArc ADL as background, and section 4 presents our method of variable component types and the language extensions of MontiArc. Then, section 5 applies the presented method and section 6 evaluates our implementation before section 7 discusses the approach and section 8 compares it to related work. Finally, section 9 concludes.

2. Component & Connector Architectures

Component & Connector (C&C) ADLs (Medvidovic & Taylor 2000b) aim to describe software system architectures using components as units for computations and connectors between their interfaces to exchange messages. The fundamental elements of architectural descriptions (Medvidovic & Taylor 2000a) are: components, connectors, and configurations. Components are the units of computation in an architectural model and yield well-defined interfaces. Connectors connect the interfaces of components to realize component communication. A configuration is a graph of components and connectors that describes component composition.

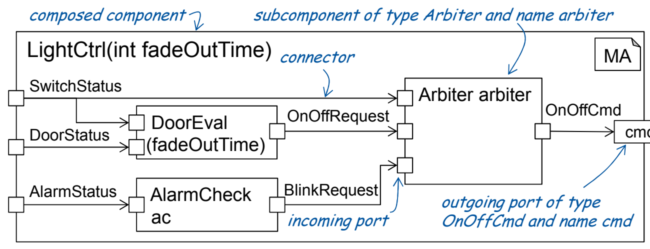


Figure 1 MontiArc architecture for a light control system with three subcomponents

MontiArc (Haber et al. 2012; Butting et al. 2017) is a textual C&C ADL and modeling infrastructure for the development of distributed systems. Consequently, MontiArc provides a small core of language features that are easy to learn yet powerful enough to model complex software architectures. The language's infrastructure comprises code generators translating models into arbitrary GPL realizations. Following the principles of C&C ADLs, MontiArc facilitates modeling C&C software architectures with hierarchically structured, interconnected components. The interface of a component is defined by a set of unidirectional, named, and typed ports. Components receive messages via their incoming ports and emit messages via their outgoing ports. Unidirectional connectors connect exactly one source port to one or more target ports.

MontiArc distinguishes between component types and component instances to facilitate component reuse. A component type (denoted "component" in the following) defines the interface of its instances by a set of ports and may comprise subcomponents and connectors defining a configuration. If a component contains subcomponents, it is called composed. Otherwise, it is called atomic. Atomic components perform the actual computations of a system. The behavior of a composite component results from the composition and behavior of its subcomponents. The behavior of an atomic component has to be implemented by hand, i.e., by providing GPL code implementations or embedded behavior descriptions.

MontiArc is a textual language. While research suggests that graphical representations improve design-knowledge transfer in software design (JSD+20 2020; Meliá et al. 2016) and feature modeling (Jakšić et al. 2014), textual modeling usually is more compact, promises to be more efficient(Meliá et al. 2016) and is platform independent (Grönniger et al. 2007). Therefore, we provide graphical representations alongside textual models for illustration purposes only.

Figure 1, for instance, depicts the graphical representation of the component type `LightCtrl`. The component controls the interior light of a car and provides behavior that depends on received messages as well as time events. Figure 2 shows its corresponding textual definition in MontiArc. The component consists of four ports (l. 2-3), three subcomponents (ll. 5–7), and seven connectors (ll. 9–15). Ports are defined after the keyword `port` and always have a type. Putting a name is optional and, if not given, is derived by uncapitalizing the type, e.g., the name of the port with type `SwitchStatus` is `switchStatus` (l. 2). Their direction can be either ingoing marked by the keyword

```
1   component LightCtrl(int fadeOutTime) {
2     port in SwitchStatus, in AlarmStatus, in DoorStatus,
3          out OnOffCmd cmd;
4
5     AlarmCheck ac;
6     DoorEval(fadeOutTime);
7     Arbiter arbiter;
8
9     arbiter.onOffCmd -> cmd;
10    switchStatus -> arbiter.switchStatus,
11                    doorEval.switchStatus;
12    doorStatus -> doorEval.doorStatus;
13    alarmStatus -> ac.alarmStatus;
14    ac.blinkRequest -> arbiter.blinkRequest;
15    doorEval.onOffRequest -> arbiter.onOffRequest;
16  }
```

Figure 2 The component type `LightControl` consists of three incoming ports, one outgoing port, three subcomponents, and multiple connectors. Furthermore, it defines a configuration parameter that is forwarded to one of its subcomponents.

`in` or outgoing indicated by the keyword `out`. Ingoing ports receive messages from a connected outgoing port of another component. Outgoing ports send messages and are connected to incoming ports of another component. Subcomponents are specified starting with a type and optionally a name. Similar to the port definition, if not given, the subcomponent name is inferred from the type name in lower case, e.g., the subcomponent `DoorEval` (l. 6) can be referred to as `doorEval` (ll. 11-12).

Connectors are unidirectional and connect one sending port with one or more receiving ports of compatible data types. The incoming port `switchStatus` of component `LightCtrl`, for instance, is connected to the same-named and type compatible incoming ports of the subcomponents `arbiter` and `doorEval` (l. 10 f.).

MontiArc already provides a point of variability, which are component parameters. Parameters have to be assigned when instantiating a component as a subcomponent. After that, the value must not change. Component `LightCtrl` has a parameter `fadeOutTime` (l. 1) that determines the fade-out duration of the light after closing a door. The parameter is passed to the subcomponent `DoorEval`, which then adheres to it. Parameters are a mean for open variability and provide capabilities for customizing component behavior on instantiation, especially for atomic components, by directly affecting values in behavior descriptions. While they also enable customization of composed component behavior through parameter forwarding and subcomponent composition, they so far did not support structural variability, that is, variabilities in the interface of components or in the structure of a component's topology.

Atomic components, i.e., components without subcomponents, can define automata to specify their behavior. Figure 3 shows the component `DoorEval` that, depending on the door and switch status, sends requests for turning the interior lights on or off. The contained automaton defines this behavior (ll. 7-19). An automaton specification always starts with specifying its states and its initial state. Afterward, the transitions between states state a guard, i.e., a boolean condition based on ingoing ports, that determines when the transition is fired, and an action statement determining what should happen in consequence of the transition firing. For instance, the automaton switches from state `off` to `on` whenever the switch is turned on or the door is opened (l. 11). In consequence, a request to turn the interior lights on is sent (l. 12).

```
1   component DoorEval(int fadeOutTime) {
2     port in SwitchStatus, in DoorStatus,
3          out OnOffRequest cmd;
4
5     int timer;
6
7     automaton {
8      initial state off;
9      state off, on, fading;
10
11     off    -> on [switchStatus == ON || doorStatus == open] /
12                 {OnOffRequest = ON};
13     on     -> fading [switchStatus == OFF
14                 || doorStatus == closed] /
15                 {timer = 0};
16     fading -> fading [timer <= fadeOutTime] / {timer++};
17     fading -> off [timer == fadeOutTime] /
18                 {OnOffRequest = OFF};
19    }
20  }
```

Figure 3 The atomic component type `DoorEval` consists of two incoming ports, one outgoing port. Furthermore, it defines a behavior automaton, that specifies when a request for turning on or off the lights should be sent.

3. Variable Component & Connector Architectures

3.1. Variability Composition

In composed-component hierarchies, components are composed of topologies of connected subcomponents. We differentiate between decomposed components that contain subcomponents and atomic components that are not further decomposed. Introducing feature modeling to introduce variability in hierarchical composition structures requires an extension of the component model. This includes 1) a mean to model variabilities and commonalities of the internal structure of components, 2) parameterization of components with a feature model, and 3) composition of public feature models across hierarchies.

Variability in decomposed components includes variability in the topological structure of subcomponents, e.g., subcomponents may only be included in certain variants, or connectors between these may vary. For such structural variabilities to be defined, a suitable means of expression must be provided. Step 1 is fairly obvious: for introducing variability in a component hierarchy, we must also support variabilities in the component structure, i.e., varying subcomponents and communication links. To provide an appropriate means of expression, a wide range of solutions are conceivable. However, their technical details have no impact on the fundamental idea behind compositional variability, which is the focus of this discussion. We, therefore, do not elaborate on this variability mechanism here; it is sufficient to assume that we have some technical concept at hand to express such variabilities in a component's internal structure. Steps 2 and 3 are more significant in our context because this is where variability management is designated to become compositional. As one of their key characteristics, all component models identify in detail the information that constitutes the public interface of a component. They all provide something like ports through which information is transmitted to or from the component; some provide means for specifying precise communication protocols using state machines, for example.

To deal with variability, we extend this interface specification by a feature model called the public feature model of the component. The purpose of this feature model is to specify the entire variability within its corresponding component in a form suitable for publication to the component's clients. Besides this extension to the public interface, we also introduce, in step 3, an extension to the internal structure: in each variably defined component, i.e., each component that provides a feature model in its public interface, a mapping is defined from this public feature model to the variants of its internal structure (step 1) and the public feature models of the directly contained subcomponents (step 2 w.r.t. the subcomponents). "Mapping" in this sense means that by applying this information, it is possible to derive a configuration of the component's internal structure and the public feature models of the contained lower-level components. Thus, the mapping states how to internally configure a component depending on the configuration of its public feature model. The precise internal realization of some variability presented in the public feature model (e.g., as an optional feature), in particular, whether it is realized by structural variability or by a variable subcomponent, is completely concealed from clients. In addition, how this internal variability is actually configured when an individual variant of the containing component is chosen is also hidden. This is perfectly in line with information hiding, the basic idea behind all public interface specifications. Therefore, the public feature model can be viewed as an equal constituent of a component's public interface, and the term configuration hiding can be used for this concealment of variability and configuration-related information.

In addition to information being hidden from the environment, another important analogy to interfaces and information hiding, in general, should be noted here. Just as the method signatures in a class in object-oriented programming abstract from the concrete behavior implementation, the public feature model of a component is independent of how the variability is internally structured. The variability of contained subcomponents as defined by their public feature models can be packaged and presented completely orthogonally in the containing component's public feature model.

3.2. Definition of Variable Component Types

Component models consist of hierarchies of connected components. To support the reuse of components, we distinguish between component definitions, i.e., the type of a component and its instantiation. Components of the same type can be instantiated multiple times in the same system. While reuse is an essential aspect in system development, effective utilization of component types also requires flexibility in their definition. An example is the definition of generic component types that provide a service, such as buffering or filtering of messages, for an arbitrary data type. A generic definition does not have to be developed anew for various data types but instead can be reused with little effort. Besides interface flexibility, however, one may also require flexibility in the behavior of a component. A common mechanism for behavior customization is parameterization, which enables setting various variables and thus customizing component behavior on initialization. In component and connector (C&C) architectures, flexibility in the behavior also requires flexibility in the structure, at least when considering composed components, as the interconnection of subcomponents defines a composed component's behavior. Structural flexibility is also required when considering interface flexibility apart from generic data types. With flexible component definitions, individual component types do not define a single computation function, but instead describe a set of behaviors from which one is selected on initialization of the component through parameterization. In the following, we introduce variable component type definitions that provide a template for instantiation and customization of respective component variants. The formalization shows the generalizability of our concept and is designed to apply to other C&C architecture description languages that have similar concepts.

Notation. In the remainder of this paper, we assume that mathematical objects beginning with capital letters are sets while others are individuals. Moreover, we use the"dot-notation", i.e., $p.x$ means that p is a tuple and $x \in Name$ identifies a field of the tuple, e.g., for Point $\subseteq \{x : \mathbb{N} \times y : \mathbb{N}\}$, $p.x$ identifies the first value of a $p \in$ Point.

- $Name$... a universe of names,
- $Type$... a universe of data types,
- $\mathbb{B}$... = {true, false},
- $\mathcal{P}$(S)... powerset of S,
- $S_1 \times \ldots \times S_n$... cartesian product over $S_1, \ldots, S_n$,
- $CTDefs$ a universe of component type definitions

Definition 1 ***Variable component types:*** *A variable component type definition is a 8-tuple* $Comp = (CType \times CPorts \times CParams \times CF \times CSubCmps \times CCons \times \Delta \times \Gamma) \in CTDefs$ *with* $CType \in Name$ *is the name of the component type.*

Definition 2 ***Port definition:*** $CPorts \subseteq \mathcal{P}(Ports)$ *with* $Ports = (dir : \{IN, OUT\} \times name : Name \times type : Type)$ *is the set of typed, directed ports of all variants of component type CType defining the component's interface. Each port* $p \in CPorts$ *has a name* $p.name$*, a type* $p.type$*, and a direction* $p.dir$*. For some variable component type* $comp$*, with* $CPorts_{IN} = \{p \in comp.CPorts \mid p.dir = IN\}$ *and* $CPorts_{OUT} = \{p \in comp.CPorts \mid p.dir = OUT\}$ *we denote the set of incoming and outgoing ports, respectively.*

Definition 3 ***Connector definition:*** $CCons \subseteq \mathcal{P}(Cons)$ *with* $Cons = (srcCmp : Name \times srcPort : Ports \times tgtCmp : Name \times tgtPort : Ports)$ *is the set of connectors of all variants of component type CType. Each connector* $con \in CCons$ *has a source component* $con.srcCmp$*, a source port* $con.srcPort$*, a target component* $con.tgtCmp$*, and a target port* $con.tgtPort$*.*

Definition 4 ***Parameter definition:*** *$CParams \subseteq Params$ with $Params = (name : Name \times type : Type)$ is the set of parameters of component type $CType$. Each parameter $param \in CParams$ has a name $param.name$ and a type $param.type$.*

Definition 5 ***Feature definition:*** *$CF \subseteq Name$ is the set of features of the variable component type $CType$.*

Definition 6 ***Subcomponent definition:*** *$CSubCmps \subseteq \mathcal{P}(SubCmps)$ with $SubCmps = (name : Name \times type : CTDefs \times \Phi)$ is the set of subcomponent instances of all variants of component type $CType$. Each subcomponent instance $sub \in CSubCmps$ has a name $sub.name$, a type $sub.type$, and a component configuration Φ*

Definition 7 ***Component configuration:*** *We call $\Phi = \Pi \times \Psi$ a component configuration and by P_Φ denote an element of a component type definition under the application of the component configuration Φ.*

Definition 8 ***Parameter assignment:***
$\Pi = \times_{param \in type.CParams}\{CParams \rightarrow param.type\}$

Definition 9 ***Feature configuration:***
$\Psi = \times_{f \in type.CF}\{CF \rightarrow \mathbb{B}\}$

Definition 10 ***Configuration application:*** *$\Delta : \Phi \rightarrow \mathcal{P}(CCons) \times \mathcal{P}(CPorts) \times \mathcal{P}(CSubCmps)$ is the mapping of component configurations, denoting the elements of the component type under the application of the component configuration.*

Definition 11 ***Constraining variability:*** *$\Gamma : CParams \times CF \rightarrow \mathbb{B}$ is a constraint predicate over parameters and features that restricts the set of valid variants of the component type.*

In contrast to component definitions, variable component definitions do not define a single component type with fixed wiring, but a component with various characteristics or configurations. However, the definition is subject to further constraints on well-formedness. Instead of being based on a fixed configuration, the well-formedness rules here are extended so that the rules of a single component definition must apply to all configurations of a variable component type. That is, a variable component type is well-formed, if all of its valid configurations are well-formed. Thus, the following well-formedness rules apply with respect to all valid parameter assignments and feature configurations Φ of the component type:

Well-formedness rule 1 *Ports and variables within a component type definition have unique names, that is $\forall CPorts_\Phi \subseteq CPorts : \forall e_1, e_2 \in CPorts_\Phi \cup CParams : e_1.name = e_2.name \Leftrightarrow e_1 = e_2$*

Well-formedness rule 2 *Each port has at most one incoming connector, that is $\forall con_1 \in CCons_\phi \neg\exists con_2 \in CCons_\phi : con_1 \neq con_2 \wedge con_1.tgtCmp = con_2.tgtCmp \wedge con_1.tgtPort = con_2.tgtPort$*

Well-formedness rule 3 *Each subcomponent has a unique name different from CType, that is $\forall sub \in CSubCmps_\phi : sub.name \neq CType \wedge \forall sub_1, sub_2 \in CSubCmps_\phi : sub_1.name = sub_2.name \implies sub_1 = sub_2$.*

Well-formedness rule 4 *Each connector $con \in CCon_\phi$ satisfies one of the following four cases:*

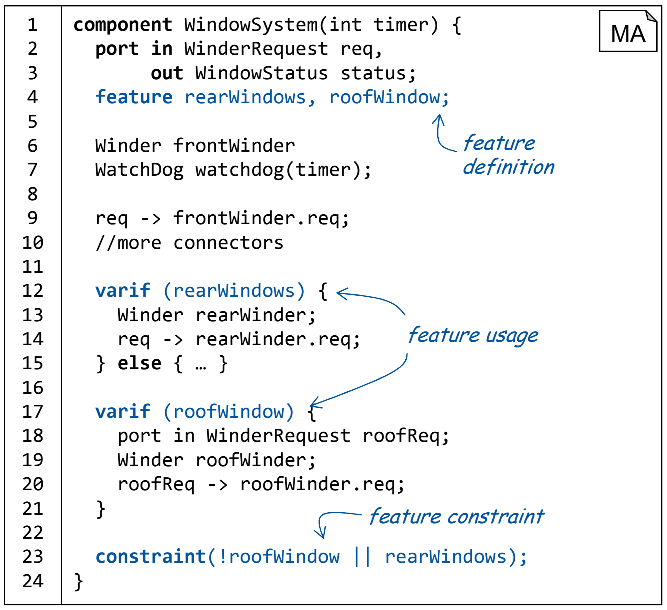
- *The component directly forwards input as output, that is $con.srcCmp = con.tgtCmp \wedge con.srcPort.dir = IN \wedge con.tgtPort.dir = OUT$.*
- *The component forwards input to a subcomponent, that is $con.srcCmp = CType \neq con.tgtCmp \wedge \exists t \in Name : (con.tgtCmp, t) \in CSubCmps_\phi \wedge con.srcPort.dir = IN \wedge con.tgtPort.dir = IN$.*
- *The component forwards output of a subcomponent, that is $con.srcCmp \neq CType = con.tgtCmp \wedge \exists t \in Name : (con.srcCmp, t) \in CSubCmps_\phi \wedge con.srcPort.dir = OUT \wedge con.srcPort.dir = OUT$.*
- *Two subcomponents are connected, that is $\exists t_1, t_2 \in Name : (con.srcCmp, t_1), (con.tgtCmp, t_2) \in CSubCmps_\phi \wedge con.srcPort.dir = OUT \wedge con.tgtPort.dir = IN$.*

Based on the parameter assignment, the configuration of the component is selected. Well-formedness is required of the component type, and thus, all component instances received are ensured to be well-formed. Component parameterization introduces structural variability, and especially variability in component interfaces. The well-formedness of variable component types must be considered in the context of component parameterization, thus is not solely dependent on the component types used. The definition requires that connectors between subcomponents only exist with respect to their (current) configuration. Furthermore, all configurations of a variable component type must be well-formed, ensuring that no invalid configurations can be selected on component initialization. Not covered by the definition is the differentiation between atomic and composed components. The latter required at least one subcomponent and otherwise well-formedness, whereas all components without subcomponents are considered to be atomic components providing some other form of behavior definition not covered here.

4. Realization of Hierarchical Variability Modeling in Montiarc

4.1. Variable Component Types in MontiArc

For defining the set of a component's features in MontiArc, a component type definition can contain a set of explicit features. Each feature has a name through which it is referenceable when it is used. A feature definition starts with the keyword `feature` followed by one or more feature names. Figure 4 shows a component `WindowSystem` that controls the winders of a car's windows. It defines two features, namely `rearWindows` and `roofWindow` (l. 4), which provide control capabilities for rear and roof window, respectively.



```
1   component WindowSystem(int timer) {
2     port in WinderRequest req,
3          out WindowStatus status;
4     feature rearWindows, roofWindow;
5
6     Winder frontWinder
7     WatchDog watchdog(timer);
8
9     req -> frontWinder.req;
10    //more connectors
11
12    varif (rearWindows) {
13      Winder rearWinder;
14      req -> rearWinder.req;
15    } else { … }
16
17    varif (roofWindow) {
18      port in WinderRequest roofReq;
19      Winder roofWinder;
20      roofReq -> roofWinder.req;
21    }
22
23    constraint(!roofWindow || rearWindows);
24  }
```

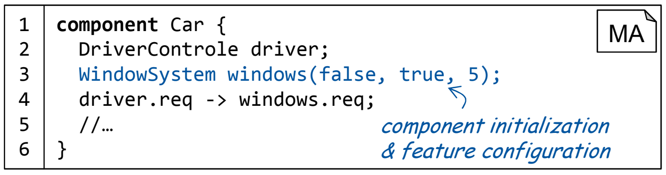
Figure 4 The `WindowSystem` defines two features `rearWindows` and `roofWindow`, which are used in this component type to activate certain subcomponents, ports, and connectors.

Constraints in MontiArc are formulated via expressions over a set of available features and component parameters. Constraints in MontiArc start with the keyword `constraint` followed by an expression. The expression is a boolean formula over features and component parameters. For instance, the component `WindowSystem` constrains that selecting the feature `roofWindow` implies that also feature `rearWindows` has be configured (l. 23).

Defining the mapping of features to component constituents, MontiArc introduces if-statements. For better distinguishability, they are identified by the keyword `varif` in component definitions and the keyword `if` in automata. Such a statement then references one or more features and component parameters of a component and combines them as a boolean expression. The body of the statement comprises the component constituents depending on the fulfillment of the expression's boolean formula. In composed components, constituents that can depend on features are ports, connectors, subcomponents, and inner components. For instance, in Figure 4, if the feature `rearWindows` is selected (l. 12), a new subcomponent `rearWinder` is added to the component (l. 13), and the `WinderRequest` is forwarded to its port `req` (l. 14). In the case of the `roofWindow`, a new component and connector are also created, as well as an additional port that is dedicated only to the roof winder (ll. 17-21).

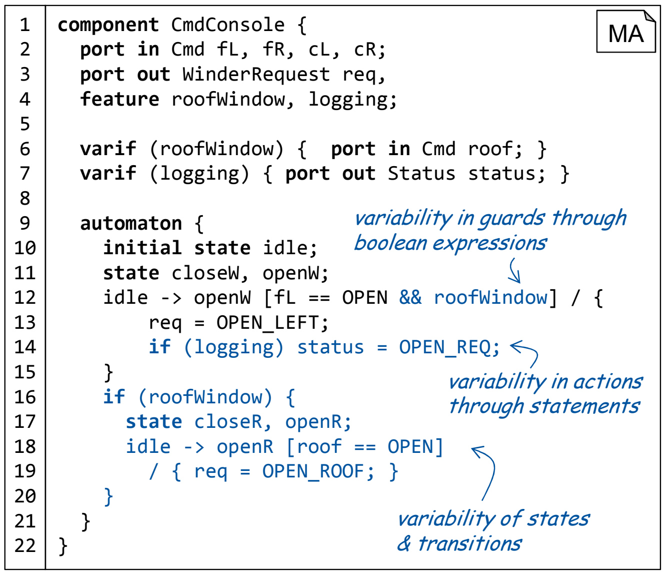
The formalism below defines the component `WindowSystem` (cf. Figure 4) using the notation from subsection 3.2.

- $cType = WindowSystem$
- $CPorts = \{(IN, req, WinderRequest), (IN, roofReq, WinderRequest), (OUT, status, WindowStatus)\}$
- $CParams = \{(timer, int)\}$
- $CF = \{rearWindows, roofWindow\}$
- $CSubCmps = \{(frontWinder, Winder), (rearWinder, Winder), (roofWinder, Winder), (watchdog, WatchDog, timer -> Timer)\}$
- $CConns = \{(req, frontwinder.req), (req, rearWinder.req), (roofreq, roofWinder.req)\}$
- $\Delta = \{(rearWindows \rightarrow \{\{(req, rearWinder.req)\}, \emptyset, \{(winder, Winder)\}\}), (roofWindow \rightarrow \{\{(roofReq, roofWinder.req)\}, \{(IN, roofReq, WinderReq)\}, \{(roofWinder, Winder)\}\})\}$
- $\Gamma = \{(!roofWindow || rearWindows)\}$



```
1  component Car {
2    DriverControle driver;
3    WindowSystem windows(false, true, 5);
4    driver.req -> windows.req;
5    //…
6  }
```

Figure 5 The configuration of the component `WindowSystem`. Feature `rearWindows` is deselected and feature `roofWindow` is selected. Furthermore, the parameter `timer` is set to 5.



```
1   component CmdConsole {
2     port in Cmd fL, fR, cL, cR;
3     port out WinderRequest req,
4     feature roofWindow, logging;
5
6     varif (roofWindow) {  port in Cmd roof; }
7     varif (logging) { port out Status status; }
8
9     automaton {
10      initial state idle;
11      state closeW, openW;
12      idle -> openW [fL == OPEN && roofWindow] / {
13          req = OPEN_LEFT;
14          if (logging) status = OPEN_REQ;
15      }
16      if (roofWindow) {
17        state closeR, openR;
18        idle -> openR [roof == OPEN]
19          / { req = OPEN_ROOF; }
20      }
21    }
22  }
```

Figure 6 The atomic component `CmdConsole` defines two features, which modify its behavior.

In atomic components, constituents that can be activated through feature configurations are ports, variables, automata, states, transitions, guards, and actions. In contrast to all other constituents, guards of transitions can reference a feature or parameter they depend on directly because guards are already conditions.

Figure 6 shows an atomic component `CmdConsole`, which defines a feature `roofWindow` and a feature `logging` (l. 4). If the feature `roofWindow` is selected, an additional port is added to the component (l. 6). In the automaton, the transition from state `idle` to state `openWindow` is only possible if the feature `roofWindow` is selected (l. 16), which evaluates to the boolean value `true`. Besides, the feature selection of `roofWindow` also adds two new states to the automaton and adds a new transition (ll. 16-19). In case, that the feature `logging` is configured, a new port `status` is added to the component (l. 7), and in the action of the transition from `idle` to `openWindow` the request is assigned to the port `status` (l. 14).

Features are configured at the time of instantiation of the component definition as subcomponent. The configuration is passed as an argument to the subcomponent similar to component parameter values. Configurations start with the list of features followed by the list of values of the parameters. A feature can either be selected (`true`) or not (`false`), that is, features are readable as boolean variables in constraints. Depending on a component's feature configuration, a component's constituents may be enabled and disabled at component instantiation. Figure 5 shows the configuration of the component `WindowSystem` (l. 3). The configuration activates feature `rearWindows` and disables the feature `roofWindow`. This is valid, because the components constraint defines that the `roofWindow` implies the `rearWindow`. In addition, the value `5` is passed for the parameter `timer`. Besides directly initializing all features of a variable component, features can instead be propagated along the hierarchy of components. This propagates the possibility for feature selection to the next level of the component hierarchy. However, we do not allow passing values other than parameters and features. Values of parameters and features are configured at design time only.

4.2. Implementation of Variable Component Descriptions

We realized a prototypical implementation of a modeling language for conceiving, developing, and maintaining the variability in the architecture of distributed systems on top of the architecture description language MontiArc (Haber et al. 2012; Butting et al. 2017) while reusing its modeling elements of components, ports, and connectors. Consequently, like MontiArc, this modeling language is realizing in the same technological space (Ivanov et al. 2002), i.e., with the language workbench MontiCore (Hölldobler & Rumpe 2017; Hölldobler et al. 2018). Its implementation entails the language's concrete syntax, abstract syntax, and well-formedness rules (context conditions). The latter are implemented the conformity checks using MontiCore's Java-based language context condition framework and Java-SMT (Karpenkov et al. 2016; Baier et al. 2021), an API for accessing various SMT Solvers in Java. Hence, the resulting toolchain consists of a MontiCore grammar that extends MontiArc with novel modeling elements to express variability, a parsing infrastructure (parser, lexer, symbol table) generated from this grammar, and more than 70 context conditions rules checking the well-formedness of models conforming to this grammar. With this in place, the well-formedness of models can be checked whenever they have been parsed deliberately and completely, i.e., MontiCore does not parse models partially.

Syntax We achieve the reuse of existing modeling elements by leveraging MontiCore's language composition mechanism, explicitly language extension. In MontiCore, language extension begins with extending a language's concrete and abstract syntax, which are defined via a Context-Free Grammar (CFG). The extension entails conceiving new grammar productions for structural variability and feature models (constraints) and embedding these into the body of component definitions, which we achieve by implementing a provided grammar production interface. The grammar productions for constraint and the condition of the *varif*-statement each embed an expression grammar production from a provided library. This enables constraints and conditions to contain extensive expressions consisting of conditional operators, assignments, and method calls. In addition, the block spanned by a *varif*-statement corresponds to a component body, whereby it can contain any architectural elements.

From the CFG extension, MontiCore generates a basic language infrastructure in Java, including an ANTLR-based parser. The parser can already parse textual descriptions of variable component types into an object structure of the typed abstract syntax tree in accordance to the defined grammar productions. In addition to the parser, the provided language infrastructure consists of symbol table management, visitors with traversal of the AST, interfaces for context-condition checks and a code generator engine. All further functionality is implemented on top of this infrastructure.

Well-formedness Conditions Because the generated parser is conceived from a CFG, it processes models unaware of their context. We implement context-condition checks on top of the parser to check the validity of models with regard to their context. For the variability extension, there are two kinds of checks to consider. Namely, those that are independent of the variability context and those that depend on it. The former means conceiving context conditions for newly introduced language features. and the latter means introducing variable context to all context-condition checks that depend on it.

Constraints and conditions must be boolean expressions. As MontiArc is static-typed and already provides type-checking capabilities, we can simply reuse the existing type-check and reapply it to the new language features. However, while most expressions in MontiArc are afterward translated to program code and executed there, constraints and conditions should already be evaluated during compile time. By evaluating constraints, we would determine whether the configuration of a component is valid to the given feature model.

We evaluate constraints during compile time by integrating an SMT solver. We chose Z3 (De Moura & Bjørner 2008) as it is widely used and one of the most powerful SMT solvers. Furthermore, we utilized Z3-TurnKey [1] to be able to ship our application with Z3 native libraries. The integration was implemented by transforming the constraint and symbols, i.e., the available variables, into Z3 Expressions and Z3 Context, respectively. We apply the transformation and check for satisfiability after the type check and while executing the context condition checks. Constraints for which the SMT solver cannot find an assignment are reported as errors.

There are also some limitations for expressions that can be used in constraints and conditions of *varif*-statements. Both are part of the static architecture. Evaluation of these expressions should therefore be side-effect-free. However, the expressions used offer extensive constructs. In the static part, we therefore prohibit all potentially side-effect-prone expressions, such as

[1] https://github.com/tudo-aqua/z3-turnkey

assignments and method calls. These restrictions are also implemented as context-condition checks implementing the provided interface.

Analyzing Variable Systems While variability increases the flexibility of a system description it also introduces additional complexities. Instead of a fixed configured system, elements of the definition must be considered concerning all variants. Analysis of variable systems may need to consider all possible manifestations of a system. As our modeling language introduces structural variability, this implies that well-formedness checks need to consider the availability of modeling elements with respect to feature configurations.

Analysis of variable definitions is a common problem in managing software product lines. Multiple approaches exist for lifting analysis of single systems to the analysis of a variable system. A classification of product line analysis strategies differentiates roughly between product-based, feature-based, and family-based analysis and combinations thereof (Thüm et al. 2014), all having their advantages and disadvantages. Product-based analyses apply to the product, deriving each product to ensure full coverage, resulting in computation overhead through redundant computations. Feature-based analyses instead apply to domain artifacts that implement a certain feature in isolation and are, therefore, unable to analyze properties of interacting features. Finally, family-based analyses apply directly to domain artifacts and variability models, promising the most efficient computations.

We decided to implement a combination of family-based and product-based analyses to realize well-formedness checks for variable component definitions. We use product-based analyses to reuse a multitude of context-condition checks, ranging from simple design convention rules to complex type checks. Implementations for these context conditions do not consider variability. We thus leverage product-based analyses to reuse their implementation. We do so by calculating the variants defined by a variable component model at design time, virtually creating their corresponding component models, and employing the already existing context-condition checks on these. We can therefore reuse all existing context conditions, with the drawback of redundant computations.

Any context condition that is independent of variability is directly applied to variable component definitions instead, preventing redundant computations. This also includes condition checks that we have newly implemented. For example, checking the feature constraint of each variable component model.

By focusing on each component definition in isolation, we can streamline our analysis process, avoiding the need to calculate the entire component model. Variable structures of subcomponents are encapsulated. That is, context-condition checks are unaffected by the variability contained in any of the component's subcomponents. Only the feature model and variable interface descriptions are part of a component's signature and must be checked during component instantiation. These are checked with respect to the enclosing component's feature model. The enclosing component's feature model is composed of the locally defined feature constraints and the feature model of its subcomponents. Once the feature model of the enclosing component is composed, it is converted back to SMT and checked. This step ensures that the feature constraints of the subcomponents remain intact and are not violated.

5. Case Study

We evaluate the applicability of our method for modeling variability of C&C architectures in a case study using a flight control system based on an existing AADL (Feiler & Gluch 2012) example[2]. The goal is to engineer a family of flight systems that feature different variants representing different configurations of the system. Each has a different maturity level for their respective scenario. We accommodated this case study to MontiArc by reconstructing the vital components. Thus, we also provide the respective translated artifacts [3] for a complete overview of the implemented models. In the following, we discuss the most significant parts of the overall system, to evaluate the use of variability in MontiArc.

The overall architecture is represented in Figure 7. The flight system receives a satellite signal and input from the pilot and computes the observable failure of the route. In its base configuration, the system's behavior is based on a GPS subsystem, evaluating the received satellite signal, an internal power supply, an automated flight guidance for adjusting the pilot's command to the current coordinates, and a flight control system calculating the necessary measures. Furthermore, the modeled architecture contains two additional features, `advanced` (highlighted in red) and `dualGPS` (highlighted in blue). The advanced system uses improved GPS components, enabling even more precise localization, and a more complex auto flight guidance system. The latter is further subdivided into a basic flight guidance and an additional auto pilot. The dual GPS feature further introduces a second GPS system and a corresponding voter, processing both GPS signals to increase the accuracy. The features can be used separately or jointly, ultimately resulting in four valid configurations. The combined use can be observed in the second GPS. Here, a new component, `gps2`, is introduced, which is additionally required in its advanced configuration.

From an engineering perspective, all variants should be developed together in a single product line to avoid redundant implementations. In addition, analyses and well-formedness checks should support component developers in detecting errors early on. Figure 8 presents the corresponding textual model in MontiArc, introducing the two features (l. 2). Using the `dualGPS` configuration introduces the additional subcomponents `gps2` and `voter` (ll. 15-19). Additionally, new connections between the components are established (ll. 27-34). While the default configuration connects `gps1` (the only GPS in this variant) directly to the `autoFlightGuidance` (l. 33), the `dualGPS` modification interposes the `voter`, for processing multiple GPS signals first (ll. 27-31). Furthermore, the `advanced` configuration is forwarded (via constraints) to the selected subcomponents `autoFlightGuidance` (l. 36), `gps1` (l. 37), and `gps2` (l. 18) if existent. This enables instantiating these in the desired configu-

[2] github.com/osate/examples/tree/master/SafetyTutorial/packages
[3] github.com/MontiCore/montiarc/tree/develop/applications/avionics

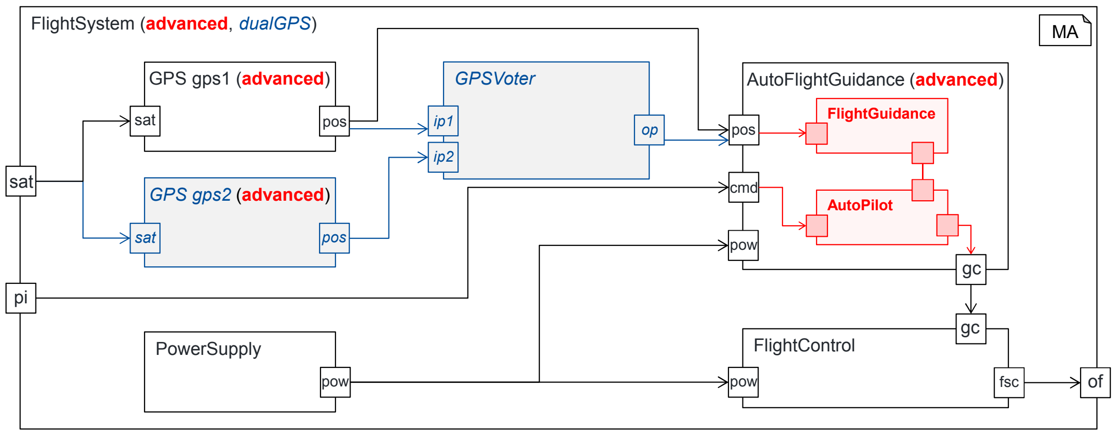


Figure 7 MontiArc architecture of a variable flight control system with two features, `advanced` and `dualGPS` resulting in four valid configurations.

ration, given, that the corresponding feature is available in the respective subcomponent.

The auto flight guidance supports two different configurations based on the `advanced` feature. We developed the variable component definition, computing the appropriate guidance command based on the current position and the operators steering command. In its base version (Figure 9, top), the component features a behavior definition via an automaton. Generally, the auto flight guidance component has three states. In its `Operational` mode, it continuously calculates the required guidance command. If the input data is assumed to have low fidelity, i.e., due to low precision, the system turns into a `NonCriticalModeFailure`, where the operation is still performed. The system can also recover from this state back into its `Operational` mode. If the service in unavailable, no pilot input present, or the power supply cut, we switch into a `CriticalFailureMode`, where no guidance command is computed.

Choosing the `advanced` configuration of the auto flight guidance (Figure 9, bottom) further decomposes the component into two subcomponents, a general flight guidance and an auto pilot. Thus, variability in MontiArc can influence the decomposability of a system. In this case, the position data is first processed and then, together with the pilot's steering command, forwarded to the auto pilot, which then, in turn, performs a more sophisticated computation of the resulting guidance command.

The corresponding textual model is presented in Figure 10. Similar to the overall flight system, the component has the `advanced` feature definition (l. 2). As the auto flight guidance is unrelated to the GPS configuration, the `dualGPS` feature does not apply. The structural variability is expressed in the presented `varif`-statement. While the base configuration contains an automaton specification (ll. 17-35), the `advanced` case (ll. 9-15) defines the discussed subcomponents. In the latter case, the composition of these subcomponents' behavior defines the behavior of the auto flight guidance itself.

We first developed the general architecture and then implemented the behavior of the subcomponents such as auto flight guidance, which constantly triggers the calculation of the corresponding command for the flight control. Thus, calculation depends on the selected feature. Here, static analysis comes into effect as some ports are only available for specific features. Our prototypical realization, for example, detects potential usage of the second GPS system as well as the GPS voter, resulting in different connection configurations for the first GPS system and the auto flight guidance. In contrast, the developed GPS component type only features open variability in the form of parameters. While these do not introduce structural variability but instead are used to customize internal behavior, they effect analysis of the component definition. In general, we cannot decide the satisfiability of component configuration but instead forward checks to component instantiation instead.

The developed GPS component is instantiated in the definition of the flight system and configured with respect to the selected feature (i.e., `advanced`). Here, concrete values are provided for the component's parameter, thus enabling the analysis of potential constraints. With the provided values, we can now determine the satisfiability of said constraints, at least for these specific configurations. The static analysis of the flight system detects that, e.g., the ports of the gps voter component are only connected in configurations where the corresponding component definition exists and checks the validity of the parameter assignments against the defined constraint. Other components, such as the power supply, are unaffected by feature configurations. Similarly, the components of the gps voter or the autopilot do not provide any structural variability. However, they are included for specific variants for their enclosing components and connected respectivly. Therefore, the analyzes detect for each variant and component references in connectors that the respective component instances exist.

While features of the auto flight guidance or GPS components are not explicitly assigned, they are forwarded to the signature of the flight system. As the latter also defines features of the same name, these are mapped to each other. That is, selecting feature `advanced` for a flight system also implies selecting that feature for its subcomponents.

```
component FlightSystem {                                    MA
  feature advanced, dualGPS;

  port in SatelliteSignal sat;
  port in CMD pi;
  port out boolean of;

  PowerSupply powersupply;
  FlightControl flightControl;
  AutoFlightGuidance autoFlightGuidance

  GPS gps1;
  sat -> gps1.satSignal;

  varif (dualGPS) {
    GPS gps2;
    GPSVoter voter;
    constraint(gps2.advanced == advanced);
  }

  pi -> autoFlightGuidance.cmd;
  powersupply.pow -> autoFlightGuidance.pow;
  powersupply.pow -> flightControl.pow;
  autoFlightGuidance.gc -> flightControl.gc;
  flightControl.fsc -> of;

  varif (dualGPS) {
    sat -> gps2.sat;
    gps1.pos -> voter.ip1;
    gps2.pos -> voter.ip2;
    voter.op -> autoFlightGuidance.pos;
  } else {
    gps1.pos -> autoFlightGuidance.pos;
  }

  constraint(autoFlightGuidance.advanced == advanced
    && gps1.advanced == advanced);
}
```

Figure 8 MontiArc architecture of a cruise control system featuring two additional variants.

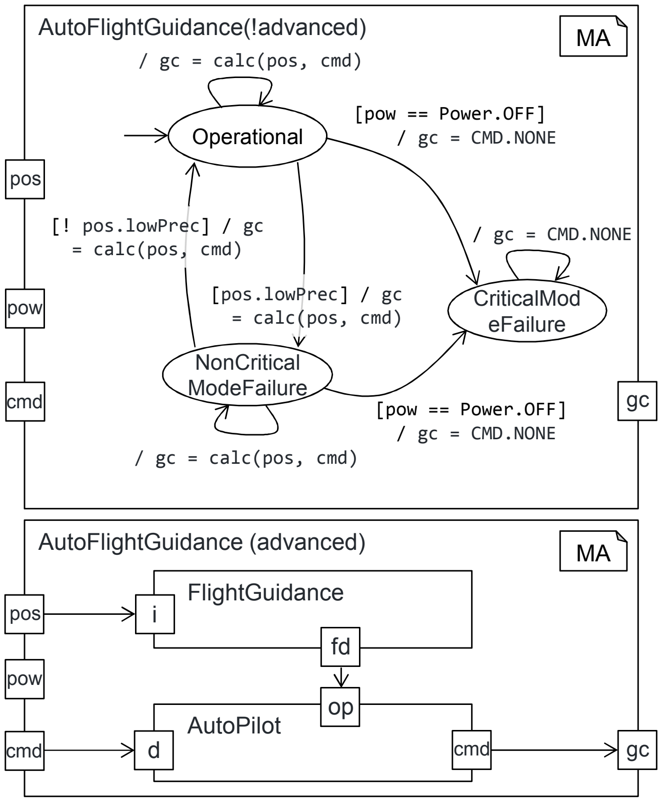


Figure 9 The two configurations of the auto flight guidance component: One configuration defines the behavior via an automaton, and the other configuration introduces further subcomponents.

6. Evaluation

We evaluated the MontiArc modeling infrastructure for statically analyzing variable component descriptions using multiple experiments. In this section, we present and discuss our experimentation setup and results. First, we examine the efficiency and scalability of our approach via experiments with an increasing number of variants. Then, we examine the effect of feature constraints on the execution time of the static analysis. Lastly, we present experiments showcasing the effect of encapsulating variable structures in subcomponents.

The experiments presented here all follow a general setup. Each experiment consists of multiple runs with input models of varying sizes. A run is processing the input models, that is, parsing the models to create their ASTs, performing inter-model transformations on the ASTs, creating the symbol table, and finally performing various static-analysis checks. We executed each run a hundred times and tracked execution times. All runs were executed on a computer with a 3.5GHz quad-core processor, 16 GB DDR4-RAM, and Windows 10 operating system.

In our first experiment, we analyze the effect of the number of variants on the execution time. To this end, we created multiple artificial models with varying numbers of features and variation points. Each feature defines two ports, a subcomponent, and connectors between the component's and the subcomponent's ports. For simplicity, we reuse the same component type for each subcomponent. As the component models in this experiment do not constrain feature combinations, the number of variants defined by a component is 2^F, where F is the number of features. We run the experiment for varying sizes of F.

The mean execution time of each run is shown in Figure 11. The depicted graph shows the number of features on the x-axis and the mean execution time on the y-axis. The smallest standard deviation is 0.0333 for the run with four features. The standard deviation for 14 features is 6.0588. The execution time is exponential to the number of features, i.e., linear to the number of variants.

The main factors for execution time for small values of F are model parsing, symbol-table creation, and variant-independent checks. Execution time for these activities grows linear with the input size, and there is a constant delay for tool startup, which puts execution times close together. With increasing values of F, the execution time of product-based analyses becomes more dominant. In an agile development environment, execution time for a small number of features may be justifiable. With a larger number of features, however, analyses of all variants can no longer be carried out with every change without hindering efficiency.

So far, each feature has added variation to a component, and feature combinations were not constrained. In our second experiment we reused the models from the first experiment to test the effect of adding features without variation. We repeated

```
component AutoFlightGuidance {
  feature advanced;

  port in Coordinates pos;
  port in CMD cmd;
  port in Power pow;
  port out CMD gc;

  varif (advanced) {
    FlightGuidance flightGuidance;
    AutoPilot autoPilot;
    pos               -> flightGuidance.i;
    cmd               -> autoPilot.op;
    flightGuidance.fd -> autoPilot.d;
    autoPilot.cmd     -> gc;
  } else {
    automaton {
      initial state Operational;
      state NonCriticalModeFailure;
      state CriticalModeFailure;

      Operational -> Operational /
                     {gc = cmd.calc(pos);};
      Operational -> NonCriticalModeFailure [pos.lowPrec] /
                     {gc = cmd.calc(pos);};
      Operational -> CriticalModeFailure [pow == Power.OFF] /
                     {gc = CMD.NONE;};
      NonCriticalModeFailure -> NonCriticalModeFailure /
                     {gc = cmd.calc(pos);};
      NonCriticalModeFailure -> Operational /
                     {gc = cmd.calc(pos);};
      NonCriticalModeFailure -> CriticalModeFailure [pow ==
                      Power.OFF] / {gc = CMD.NONE;};
      CriticalModeFailure -> CriticalModeFailure /
                     {gc = CMD.NONE;};
    }
  }
}
```

Figure 10 Textual MontiArc architecture of the variable auto flight guidance component featuring two variants.

each run but added ten unused features to the input model. The model processor reports all unused features. We observed an average increase of 0.48 seconds in the mean execution time of each run. We conclude that the execution time is mostly linear to the unique combinations of variation points. Feature combinations that build on the same set of variation points are grouped and evaluated together. Unused features have only a small effect on computation time. The observed increase in execution time may be due to an increase in the size of the SMT formal used to determine the unique variants and the reporting of unused features.

In our third experiment we evaluate the effect of encapsulating features into subcomponents. Features that have no interaction can be encapsulated independently from one another. That is, instead of having one component that holds the whole variability, we can decompose that component into multiple subcomponents with variable parts. The feature model of the composed component is unchanged. If there is no variability in the interface of these subcomponents, then we can effectively reduce the complexity of each component. Furthermore, each component can be checked in isolation with a reduced number of variation points. We thus would expect to see a decrease in the average computation time. To evaluate this, we reused the models from our first experiment but distributed the variation points evenly across two subcomponents. That is, for an even number of features, each subcomponent holds half of the variation points of the examples from the first experiment. Each

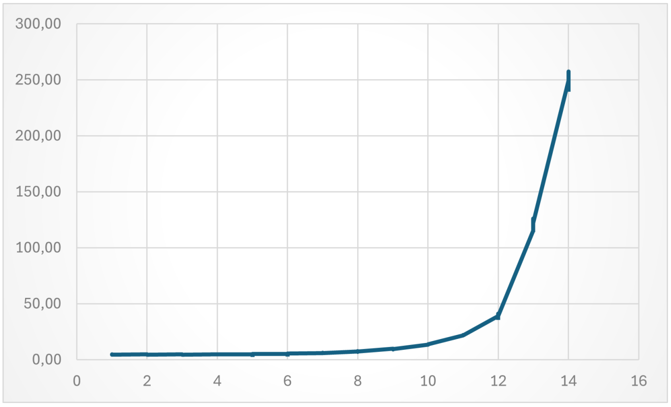


Figure 11 The graph showcases the mean execution time of processing a model with the given feature size. The x-axis is the number of features; the y-axis is the mean execution time in seconds.

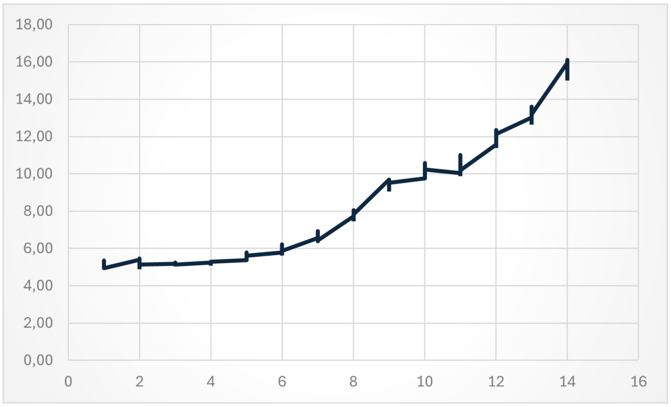


Figure 12 The graph showcases the mean execution time of processing a model with the given feature size where features are evenly distributed across two subcomponents. The x-axis is the number of features; the y-axis is the mean execution time in seconds.

subcomponent defines its own feature model, and the composition of these feature models is exactly the feature model from the first experiment. The mean execution time of each run of this experiment is shown in Figure 12.

Compared to the first experiment, the execution time increases significantly less with the number of features. However, there is a computation overhead for models with a small number of features. The experiment shows that decomposition can be effectively used to reduce the complexity of variable component descriptions. Instead of analyzing one big model, variable parts can be evaluated in isolation. The worst-case execution time is still exponential to the number of features. However, features without interaction can be effectively split across subcomponents to avoid redundant computations. Variable components from a library can be reused without reevaluating their internal structure. Reusing the same component type multiple times, i.e., having multiple subcomponents of the same type, does not result in redundant computations besides calculating the overall feature model. While this approach effectively reduces computational complexity, it is only viable where one can identify sets of variation points without feature interaction. Ultimately, the approach is still bottlenecked by feature explosion.

7. Discussion

We aimed to develop a methodology for modeling variable system architectures that maintains the composability of components. The developed methodology empowers component developers to create libraries of variable components and compose these as needed. Rather than having a static global feature model, feature models became part of components' signatures to maintain the black-box usability of components. A component's feature model can be composed alongside component usage to create the overall feature model of the modeled system. A particularly nice aspect of the methodology is that components can be configured directly on usage. That is, component users have the freedom to decide which part of a subcomponent's feature configuration is locked in place and how to compose the remaining features with the feature model of the enclosing component. Furthermore, as subcomponents are configured individually, component users can employ different variants of a component in the same architecture.

However, there are some limitations to consider when employing the presented methodology. Feature explosion, a common challenge in modeling variant-intensive systems, is also a significant concern in our methodology. Modeling 150% architectures has the drawback that models can become large fast. Despite that, we opted for the 150% modeling approach due to its seamless integration with other MontiArc concepts. MontiArc already supports parameterizing components for behavior configuration. Features and structural variability are just an extension of these concepts.

As presented in the evaluation, encapsulating variable structures in subcomponents is an option to battle complexity. Where this is not an option, variable component descriptions can become hard to maintain. However, the main aspects of our methodology are not subject to 150% modeling approaches. Making feature models part of a component's signature and supporting feature configuration on component usage could also be achieved with other variability modeling approaches.

With regard to the implementation, there is room for improvement. We decided to use a product-based analysis approach to some of the static analyses to reuse already existing implementations. Reimplementing these to make analyses variant-aware could show improvements in execution times. For example, variable structures could be encoded together with the well-formedness rules in SMT to employ SMT solving to check well-formedness across all variants.

The presented approach is applicable to components of component-and-connector ADLs that share similar properties (i.e., components with interfaces of named elements, connectors between these interfaces, . . .). Furthermore, extending its implementation most often demands implementing new context conditions (of ADL-specific complexity) only. For instance, applying it to ADLs featuring bidirectional ports does not require changing our method. Instead, only adding corresponding conditions is needed to conduct the structural checks.

8. Related Work

Variability has already been explored in the area of component and connector architectures (Suloglu et al. 2018). For AADL (Feiler & Gluch 2012), multiple approaches exist for modeling variability. In (Shiraishi 2010), features and their characteristics are directly defined in the body of the component, and (Adachi Barbosa et al. 2011) propose an aspect-oriented approach for modeling software product line architectures. The approach presented in (González-Huerta et al. 2014) builds on the common variability language (Haugen et al. 2012), representing variability to a base model separately, with a resolution model providing decisions for variability selection. However, in all these approaches, the feature model is not part of a component's signature. Instead, the feature model is defined globally. Furthermore, MontiArc configures components directly during component instantiation, enabling the composition of partially configured components and their feature models. Variability binding (Dolstra et al. 2003) during product configuration only severely limits the reuse.

In EAST-ADL (Debruyne et al. 2004), variability is also part of a component's signature (Blom et al. 2013). While structural variability of composed components is supported for black-box usage, modeling variability of behavior requires a more detailed view of the system (Leitner et al. 2012). Similar to MontiArc, EAST-ADL supports the composition of variable components, but this is done along the hierarchy of components only (Blom et al. 2016). Explicitly, each component in EAST-ADL is understood as a variable element, thus representing a feature in the feature tree. In addition, feature trees of individual components are composed alongside hierarchical component composition. In MontiArc, however, the structure of the underlying feature tree is independent of the hierarchy of components. Rather, features in MontiArc are defined over global namespaces and can be partially bound or hidden by other features at different hierarchy levels.

The approaches described in (Mann & Rock 2009; Loughran et al. 2008; Thiel & Hein 2002) focus on the description of system architectures in the automotive sector. The approaches are external to the architecture description language. Individual variable elements, like optional ports, are modeled as individual features. The analysis is only employed on the feature model. The approach we have presented makes it possible to check the well-formedness of the composition of variable components.

Variability has already been explored several times in MontiArc using different variability modeling techniques. Hierarchical variability modeling presented in (Haber, Rendel, Rumpe, Schaefer, & van der Linden 2011) likewise supports the composition of variable components and late substitution of variable parts. However, the approach does violate the black-box view of components. Furthermore, variability binding at the time of component instantiation is not supported. Instead, a single variant of the component type is derived, severally limiting the reuse of variable component definitions. Instead, the solution presented here is more flexible, as it supports variability binding on component usage, as well as partial configuration. In another approach (Haber, Rendel, Rumpe, & Schaefer 2011), we combined MontiArc with delta-oriented programming (Schaefer et al. 2010). There, variability is not part of the component but is described by so-called (delta) transformations (i.e., small transformations adding new structure or behavior to a component). Variants of a component are described by these transformations applied to a base component. However, this approach also only allows for changes to component types that are resolved before instantiation. Furthermore, although the transformations are powerful, permitting subsequent changes and deleting elements, they add another layer of complexity which manifests through additional rules and artifacts.

9. Conclusion

We presented a method for modeling the variability of hierarchically composed components. Here, global features define variation points in component descriptions and are part of the component's signature, maintaining the black-box view of components while enabling global feature configuration without the need for composing individual feature models of subcomponents. Variability of components is bound or partially bound during component instantiation with support for hiding global features along the hierarchy of components. Furthermore, we formally defined variable component types and their well-formedness rules. We prototypically implemented modeling the method as an extension of MontiArc and presented the concrete syntax and the realization of consistency check via an SMT solver. This paves the way for exploring formal analyses of component variants.

Acknowledgments

The authors of the RWTH Aachen University were supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) [grant number 499241390] and the authors of the University of Stuttgart were supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) [grant number 441207927].

References

Adachi Barbosa, E., Batista, T., Garcia, A., & Silva, E. (2011). Pl-aspectualacme: an aspect-oriented architectural description language for software product lines. In *Software architecture: 5th european conference, ecsa 2011, essen, germany, september 13-16, 2011. proceedings 5* (pp. 139–146).

Adam, K., Butting, A., Heim, R., Kautz, O., Pfeiffer, J., Rumpe, B., & Wortmann, A. (2017). *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse*. Aachen, Deutschland: Shaker Verlag.

Adam, K., Hölldobler, K., Rumpe, B., & Wortmann, A. (2017). Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, *8*(1), 3–16.

Apel, S., Batory, D., Kästner, C., & Saake, G. (2013). Software product lines. In *Feature-Oriented Software Product Lines* (pp. 3–15). Berlin, Heidelberg: Springer.

Baier, D., Beyer, D., & Friedberger, K. (2021). Javasmt 3: Interacting with SMT solvers in java. In *International Conference on Computer Aided Verification* (pp. 195–208). Springer.

Blom, H., Chen, D.-J., Kaijser, H., Lönn, H., Papadopoulos, Y., Reiser, M.-O., … Tucci-Piergiovanni, S. (2016, 07). EAST-ADL: An Architecture Description Language for Automotive Software-intensive Systems in the Light of Recent use and Research. *International Journal of System Dynamics Applications*, *5*, 1-20.

Blom, H., Lönn, H., Hagl, F., Papadopoulos, Y., Reiser, M.-O., Sjöstedt, C.-J., … others (2013). EAST-ADL: An architecture description language for automotive software-intensive systems. In *Embedded Computing Systems: Applications, Optimization, and Advanced Design* (pp. 456–470). IGI Global.

Butting, A., Haber, A., Hermerschmidt, L., Kautz, O., Rumpe, B., & Wortmann, A. (2017, July). Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference on Modelling Foundations and Applications (ECMFA'17)* (pp. 53–70). Springer.

Debruyne, V., Simonot-Lion, F., & Trinquet, Y. (2004). EAST-ADL—An architecture description language. In *IFIP World Computer Congress, TC 2* (pp. 181–195). Boston, MA: Springer US.

De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337–340). Springer.

Dolstra, E., Florijn, G., & Visser, E. (2003). *Timeline variability: The variability of binding time of variation points.* Utrecht University: Information and Computing Sciences.

Feiler, P. H., & Gluch, D. P. (2012). *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley.

France, R., & Rumpe, B. (2007, May). Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, *abs/1409.6620*, 37–54.

González-Huerta, J., Abrahão, S., & Insfran, E. (2014). Automatic derivation of AADL product architectures in software product line development. In *Proceedings of the 1st Architecture Centric Virtual Integration Workshop.* Valencia, Spain: CEUR-WS.org.

Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., & Völkel, S. (2007). Textbased Modeling. In *4th international workshop on software language engineering, nashville.* Johannes-Gutenberg-Universität Mainz.

Haber, A., Rendel, H., Rumpe, B., & Schaefer, I. (2011). Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII* (pp. 1 – 10). fortiss GmbH.

Haber, A., Rendel, H., Rumpe, B., Schaefer, I., & van der Linden, F. (2011). Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)* (pp. 150–159). IEEE.

Haber, A., Ringert, J. O., & Rumpe, B. (2012, February). *MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems* (Technical Report No. AIB-2012-

03). RWTH Aachen University.

Haugen, Ø., Wąsowski, A., & Czarnecki, K. (2012). CVL: common variability language. In *Proceedings of the 16th International Software Product Line Conference-Volume 2* (pp. 266–267). New York, NY, USA: Association for Computing Machinery.

Heineman, G. T., & Councill, W. T. (2001). Component-based software engineering. *Putting the pieces together, Addison-Westley*, *5*, 16.

Hölldobler, K., & Rumpe, B. (2017). *MontiCore 5 Language Workbench Edition 2017*. Aachen, Deutschland: Shaker Verlag.

Hölldobler, K., Rumpe, B., & Wortmann, A. (2018). Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, *54*, 386–405.

Ivanov, I., Bézivin, J., & Aksit, M. (2002). Technological spaces: An initial appraisal. In *4th International Symposium on Distributed Objects and Applications, DOA 2002.*

Jakšić, A., France, R. B., Collet, P., & Ghosh, S. (2014). Evaluating the usability of a visual feature modeling notation. In *Software language engineering: 7th international conference, sle 2014, västerås, sweden, september 15-16, 2014. proceedings 7* (pp. 122–140).

Jolak, R., Savary-Leblanc, M., Dalibor, M., Wortmann, A., Hebig, R., Vincur, J., ... Chaudron, M. R. V. (2020, November). Software engineering whispers : The effect of textual vs. graphical software design descriptions on software design communication. *Empirical software engineering*, *25*(6), 4427-4471.

Karpenkov, E. G., Friedberger, K., & Beyer, D. (2016). JavaSMT: A unified interface for SMT solvers in Java. In *Working Conference on Verified Software: Theories, Tools, and Experiments* (pp. 139–148). Springer.

Leitner, A., Kajtazovic, N., Mader, R., Kreiner, C., Steger, C., & Weiß, R. (2012). Lightweight introduction of EAST-ADL2 in an automotive software product line. In *2012 45th Hawaii International Conference on System Sciences* (pp. 5526–5535). IEEE Computer Society.

Loughran, N., Sánchez, P., Garcia, A., & Fuentes, L. (2008). Language support for managing variability in architectural models. In *International Conference on Software Composition* (pp. 36–51). Berlin, Heidelberg: Springer.

Mann, S., & Rock, G. (2009). Dealing with variability in architecture descriptions to support automotive product lines: Specification and analysis methods. In *Proceedings of Embedded World Conference 2009* (pp. 3–5). Citeseer.

Medvidovic, N., & Taylor, R. N. (2000a). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, *26*(1), 70–93.

Medvidovic, N., & Taylor, R. N. (2000b). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, *26*(1), 70–93.

Meliá, S., Cachero, C., Hermida, J. M., & Aparicio, E. (2016). Comparison of a textual versus a graphical notation for the maintainability of mde domain models: an empirical pilot study. *Software Quality Journal*, *24*, 709–735.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., & Tanzarella, N. (2010). Delta-oriented programming of software product lines. In *International Conference on Software Product Lines* (pp. 77–91). Berlin, Heidelberg: Springer.

Shiraishi, S. (2010). An AADL-based approach to variability modeling of automotive control systems. In *International Conference on Model Driven Engineering Languages and Systems* (pp. 346–360). Berlin, Heidelberg: Springer.

Suloglu, S., Kaya, M. C., Karamanlioglu, A., Entekhabi, S., Saeedi Nikoo, M., Tekinerdogan, B., & Dogru, A. H. (2018). Comparative analysis of variability modelling approaches in component models. *IET Software*, *12*(6), 437–445.

Thiel, S., & Hein, A. (2002). Systematic integration of variability into product line architecture design. In *International Conference on Software Product Lines* (pp. 130–153). Berlin, Heidelberg: Springer.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., & Saake, G. (2014, June). A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)*, *47*(1).

About the authors

Nico Jansen is a research assistant at the Chair of Software Engineering of the RWTH Aachen University. His research interests cover software language engineering, software architectures, and model-based software and systems engineering. You can contact the editor at jansen@se-rwth.de.

Jérôme Pfeiffer is a research assistant at the Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW) of the University of Stuttgart. His research interests include software language engineering techniques and applied model-driven engineering with a focus on digital twins and Industry 4.0. You can contact the author at jerome.pfeiffer@isw.uni-stuttgart.de or visit www.isw.uni-stuttgart.de/en/institute/team/Pfeiffer-00005/.

Bernhard Rumpe is a professor heading the Chair of Software Engineering of the RWTH Aachen University. His main interests are rigorous and practical software and system development methods based on adequate modeling techniques. This includes agile development methods as well as model-engineering based on UML/SysML-like notations and domain-specific languages. You can contact the author at rumpe@se.rwth-aachen.de.

David Schmalzing is a research assistant at the Chair of Software Engineering of the RWTH Aachen University. His research interests include software architectures, model-driven engineering, and software language engineering. You can contact the author at schmalzing@se.rwth-aachen.de.

Andreas Wortmann is a full professor at the Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW) of the University of Stuttgart. He conducts research on model-driven engineering, software language engineering, and systems engineering with a focus on Industry 4.0 and digital twins. You can contact the author at andreas.wortmann@isw.uni-stuttgart.de or visit www.wortmann.ac.