A toolchain for delta-oriented modeling of software product lines

A Toolchain for Delta-Oriented Modeling of Software Product Lines*

Cristina Chesta[1], Ferruccio Damiani[2], Liudmila Dobriakova[1], Marco Guernieri[1], Simone Martini[3], Michael Nieke[4], Vítor Rodrigues[2], and Sven Schuster[4]

[1] Santer Reply S.p.A.
[2] Universitá degli Studi di Torino
[3] Magneti Marelli S.p.A.
[4] Technische Universität Braunschweig

Abstract. Software is increasingly individualized to the needs of customers and may have to be adapted to changing contexts and environments after deployment. Therefore, individualized software adaptations may have to be performed. As a large number of variants for affected systems and domains may exist, the creation and deployment of the individualized software should be performed automatically based on the software's configuration and context. In this paper, we present a toolchain to develop and deploy individualized software adaptations based on Software Product Line (SPL) engineering. In particular, we contribute a description and technical realization of a toolchain ranging from variability modeling over variability realization to variant derivation for the automated deployment of individualized software adaptations. To capture the variability within realization artifacts, we employ delta modeling, a transformational SPL implementation approach. As we aim to fulfill requirements of industrial practice, we employ model-driven engineering using statecharts as realization artifacts. Particular statechart variants are further processed by generating C/C++ code, linking to external code artifacts, compiling and deploying to the target device. To allow for flexible and parallel execution the toolchain is provided within a cloud environment. This way, required variants can automatically be created and deployed to target devices. We show the feasibility of our toolchain by developing the industry-related case of emergency response systems.

Keywords: Software Product Lines, Delta Modeling, Model-Driven Engineering, Statecharts

1 Introduction

Software is increasingly individualized and adapted to the needs of specific customers. Moreover, after development and deployment, it may often have to be

* The authors of this paper are listed in alphabetical order.

adapted to changing needs and environments. To this end, the software may be replaced by a different variant of the software having different functionality. As also these software adaptations may be individualized depending on the software's configuration and the host device's environment (e.g., sensor data), such software is often developed as a *Software Product Line (SPL)* capturing its commonalities and variabilities across different variants [15,18]. Moreover, such software variants often run distributed on remote devices in heterogeneous environments, e.g., on a car's *Electronic Control Unit (ECU)*. To allow for individualized adaptations of these distributed variants based on the software's configuration and the device's environment, an individual reconfiguration and deployment is required. Furthermore, a development environment is necessary to describe the particular domain variability as well as to develop variability-aware realization artifacts.

In this paper, we present an integrated toolchain for the development of SPLs as well as the automated derivation and deployment of their respective software variants. We employ *Feature Models (FMs)* [10] to capture the commonalities and variabilities of the different variants. Furthermore, we employ *delta modeling* [17], a transformational variability realization mechanism, to express variability within realization artifacts. As statecharts are common industrial practice to model the behavior of a system, we follow a model-driven engineering process employing statecharts as realization artifacts to integrate the toolchain in existing industrial processes. To deploy the variants, we generate C code from the variant statecharts and link this code, e.g., to device-specific code artifacts. As also these external code artifacts may be customizable, we combine the coarse-grained delta-oriented variability on statecharts with fine-grained preprocessor-based variability within the code artifacts. Finally, we deploy the generated, linked and compiled variant to the respective target device.

The variant derivation for a software adaptation is triggered based on the deployed variant's configuration and the device's environment (e.g., changing context information such as GPS coordinates) [12,13]. Therefore, environmental information has to be received from the target device and a flexible and scalable infrastructure must be available to perform multiple parallel reconfiguration processes for a potentially immense number of variants. To this end, the variant derivation, code generation, linking and compilation are performed in a cloud environment.

In particular, we make the following contributions:

- We present a toolchain for modeling an SPL using statecharts and C code as realization artifacts. We provide automatic variant derivation of a statechart variant as well as C code generation, linking to external code artifacts, compilation and automatic deployment of the variant to a target device.
- We employ a combination of two different variability mechanisms: delta-oriented modeling for coarse-grained variability on the level of statecharts, as well as preprocessors for fine-grained variability on the code level.
- To employ delta modeling, a language to express transformations must exist, which is specific to the respective target language. To this end, we present a

delta language for statecharts and a delta language for adding external code artifacts to the variant, which we call metadata.

The paper is structured as follows. Section 2 describes relevant foundations that are used throughout the toolchain. Section 3 elaborates on the running example used throughout the paper. Section 4 introduces the general workflow and the components of our toolchain. Section 5 describes how to specify an SPL using the tool suite *DeltaEcore* and our provided extensions to it. Section 6 describes the process of variant generation, code generation, linking, compilation and deployment of a particular variant. Section 7 contrasts our work to related work. Finally, Section 8 gives a conclusion and and an outlook on future work.

2 Background

In this section, we present technology and concepts of the toolchain. In particular, we present the technologies to realize variability on metamodels, to create models defining variants, to compile and link source code created from our models to existing source code.

2.1 Model-Driven Software Development with the Eclipse Modeling Framework (EMF)

In Model-Driven Software Development, models represent an abstraction of the reality [7]. A metamodel is an abstraction of models and specifies types of models. The Object Management Group (OMG) specified a standard for metamodels, i.e., a meta-metamodel, in the Meta-Object Facility (MOF)[1] and also a reduced Essential MOF (EMOF) standard. With the Eclipse Modeling Framework (EMF), it is possible to specify ECORE metamodels, which are mostly based on the EMOF standard. ECORE models are specified by classes, attributes for the classes and references between them. EMF also provides a code generator, which generates Java code for metamodels defined in ECORE.

2.2 Yakindu Statecharts

As the explicit goal of our toolchain is to integrate into common industrial practice, we mainly use statecharts as realization artifacts which specify the behavior of software products. As a representative for tool support, we use the open source YAKINDU statechart tools[2]. YAKINDU statecharts are defined as an ECORE metamodel. With YAKINDU, it is possible to define statecharts with states and transitions between these states. Moreover, it is possible to define so-called *specifications* for the statechart. These specifications can consist of variables which are accessible from transactions or states. To this end, YAKINDU defines several primitive and platform independent types. Moreover, it is possible

[1] http://www.omg.org/mof
[2] https://www.itemis.com/en/yakindu/statechart-tools

to specify interfaces and external *operations*. Each operation has a signature and can also be used in transactions or states. YAKINDU provides a graphical editor to ease the creation of statecharts in a visual representation similar to UML state diagrams[3].

To generate an executable application out of a YAKINDU statechart, there are three different code generators for the `C/C++/Java` programming languages. As input, the code generators take a fully specified statechart and produce the source code for traversing the different states and transactions in the respective target language. Moreover, the generators create definitions for the variables, interfaces and external operations specified in the *specification* of a statechart in the respective programming language. Additionally, YAKINDU supports the specification of customized code generators that are able to generate source code for arbitrary languages.

2.3 Autotools

AUTOTOOLS, also known as The GNU Build System[4], is a set of tools designed to allow source code compilation for different Unix-based environments. Compilation of C/C++ source code can be very different from one Unix-based system to another, among others, because system headers and library functions can change decisively. To this end, AUTOTOOLS automatically generate *Makefiles* for certain platforms and environments. As input, AUTOTOOLS receive user-specified high-level configuration and makefiles, as Figure 1 illustrates. In the *Makefile.am*, the

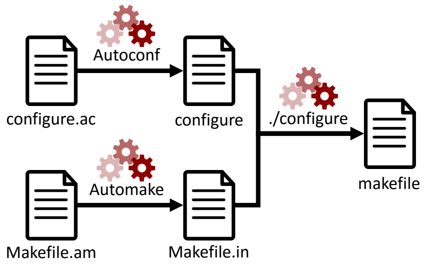


Fig. 1: Workflow to generate a makefile

user writes an abstracted makefile in a high-level specification, which is then translated to a template, the *Makefile.in*, for a concrete makefile. Users write a meta script, which abstracts over different execution environments for the script, which is called *configure.ac*. From this meta script, a portable concrete *configure* script is created, which is used to generate a concrete *makefile* out of the *Makefile.in*.

[3] http://www.omg.org/spec/UML
[4] https://www.gnu.org/software/software.en.html

2.4 Software Product Lines

Software Product Lines (SPLs) are a methodology for large-scale reuse for families of closely related software systems in terms of variabilities and commonalities [15]. Conceptually, in the *problem space*, the variabilities and commonalities of an SPL are captured in a variability model, e.g., a Feature Model (FM) [9]. Variability models define all possible configurations of an SPL. FMs consist of multiple features representing functionality of the SPL, independent of the implementation. Features are structured hierarchically and can be *optional* or *mandatory*. Moreover, features can be grouped into *alternative* or *or* groups. In *alternative* groups, exactly one feature has to be selected, whereas in *or* groups, at least one feature has to be selected. Moreover, cross-tree constraints (CTC) in terms of propositional formulas can be used to define dependencies between features independent of their hierarchical structures. Figure 2 illustrates an example of a visual representation of an FM.

The features in the *problem space* represent conceptual functionality, the realization artifacts, e.g., code or documentation, reside in the *solution space*. The realization artifacts are the artifacts, which are suitable for reuse in the SPL. The *configuration knowledge* defines the relations between the problem and the solution space, i.e., Boolean formulas defining under which feature selection certain realization artifacts are selected. Individual systems created from a configuration of an SPL are called variants or products.

Different variability realization mechanisms exist to describe variability in the realization artifacts [18]. In compositional approaches, e.g., Feature-Oriented Programming (FOP), new functionality, representing the selected features, is added to a base implementation [16,5]. In FOP, feature modules are specified, which define the new functionality and target for composition. In annotative approaches, for instance pre-processor directives (`#ifdefs`) in C [8], the variable code is annotated and removed if not needed. In transformational approaches, for instance delta modeling [6], a certain base artifact is transformed by means of adding, removing or modifying functionality. Delta languages define domain-specific delta operations, specifying add, remove and modify operations. In delta modeling, delta modules are specified, which contain a set of delta operation calls to realize the changes associated with certain combinations of features.

During the variant derivation, variability is resolved for a specific configuration, i.e., feature selection. For `#ifdefs`, a pre-processor removes the code whose annotations are not satisfied by the feature selection. In FOP and delta modeling, the feature and delta modules are selected and applied to the base code based on the feature selection using the configuration knowledge.

2.5 Delta Modeling with DeltaEcore

DeltaEcore is a tool suite, which supports developers in defining delta languages for their Ecore[5] metamodels [20]. Moreover, delta modules defined using

[5] https://eclipse.org/modeling/emf/

the created language can be applied to a base variant. The different delta modules and the base variant represent the solution space of an SPL.

When creating a language for an ECORE metamodel, DELTAECORE provides a set of pre-defined delta operations based on the structure of the metamodel. Standard operations in a delta language of DELTAECORE can access arbitrary elements of a model of the respective metamodel and, additionally, can have `Java` primitives, String types and model elements as attributes. In addition, *custom operations* can be created by defining a signature of complex operations with user defined semantics. DELTAECORE then generates stubs for these operations which merely have to be implemented by the developer. Finally, if a delta language is completely specified, a text editor is generated, in which it is possible to define delta modules, potentially consisting of multiple *delta blocks*. Each delta block specifies a base variant which is transformed by delta operations called in this block. Note that it is possible to define delta blocks in different delta languages in one delta module to realize logically cohesive changes to realization artifacts of different languages.

To be able to support the whole workflow of an SPL, DELTAECORE also supports the definition of FMs, covering the problem space of an SPL. Moreover, it is also possible to define a mapping between the FM and available delta modules, covering the configuration knowledge. When delta modules, an FM and a mapping are defined, it is possible to define configurations, consisting of selected features. Using such a configuration, DELTAECORE provides the possibility to generate variants by applying delta modules if their respective expression in the mapping is satisfied.

3 Running Example – Emergency Response Systems

In this section, we present the running example used throughout the paper, which is based on a real scenario of the automotive domain: emergency response systems for cars. An emergency response system aims to automatically dial emergency numbers in the event of a serious road accident and to wirelessly send impact sensor information and location coordinates to local emergency agencies. Different programs exist such as the eCall/E112 program of the European Union as well as the Russian ERA GLONASS system.

As different requirements for different countries exist, software for the emergency response system must behave differently depending on the current location of the car. For example, the eCall/E112 program uses the Global Positioning System (GPS) for location information, whereas the Russian ERA GLONASS system employs the Russian Global Navigation Satellite System (GLONASS). Hence, depending on the current location of the vehicle, the system needs to be reconfigured to use a different satellite navigation system.

The feature model of the emergency response system for the particular use case of supporting both the (European) "eCall" and the "EraGlonass" features is presented in Fig. 2. Depending on which system is used, a different satellite system (i.e., "GPS" or "GLONASS") and a different language ("Russian" or

"EU_Languages") is used. Moreover, the feature "ERA_GLONASS" requires the "Diagnostic" feature to be selected, whereas it is optional for the case of "eCall".

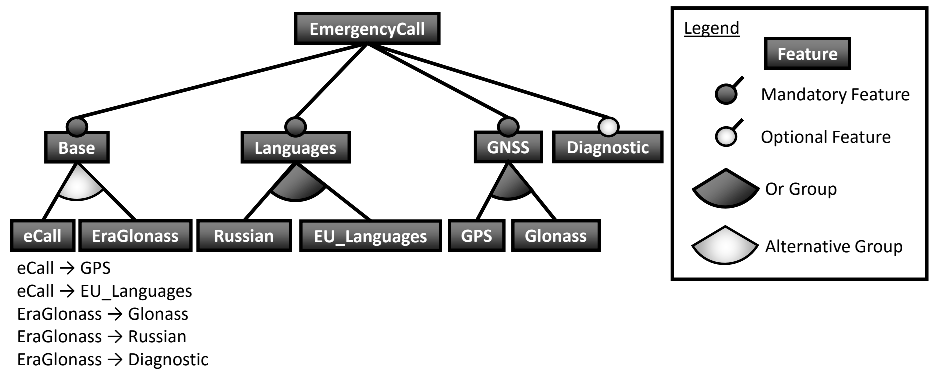


Fig. 2: Feature model and constraints for the emergency response system

To implement emergency response systems in vehicles, an Electronic Control Unit (ECU) must be deployed inside, which can connect to cellular communication networks and integrates a localization module. The Autonomous Telematics Box (ATB2) [6] is such an ECU. It integrates a telephone modules for connection to cellular communication networks and a multi-constellation satellite localization module (e.g., GPS and GLONASS). The ATB2 is particularly suitable in the eCall/E112 use case because it supports remote updates of the running firmware.

4 Overall Concept and Architecture

In this section, we introduce the general workflow and the components of our toolchain. Figure 3 depicts the steps of the workflow, the responsible components and the incorporated artifacts. The workflow is separated in two main phases: the SPL design phase and the variant-generation phase. The components and artifacts involved in the SPL design phase are located to the left of the dashed box. In both phases, both coarse-grained variability in terms of delta modules and fine-grained variability in terms of preprocessor directives are considered. In the SPL design phase, the SPL is defined using Yakindu and DeltaEcore, which consists of the core statechart and the delta modules.

Using the SPL definition in the variant-generation phase in the cloud, the `DeltaEcore Variant Generator` first creates a `Variant Model`, i.e., the variant's statechart, by applying the delta modules which are selected using the configuration knowledge and a feature selection. Then, the `Yakindu Code Generator` creates code out of the statechart. Additionally, the `DeltaEcore Variant`

[6] Magneti Marelli `http://www.magnetimarelli.com`

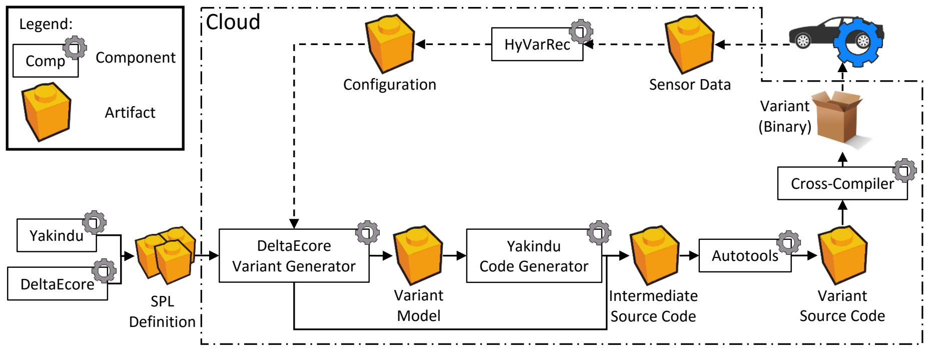


Fig. 3: Components of the Toolchain

`Generator` generates build automation files. To this end, it defines suitable metadata, which specify which features are selected, which external code artifacts should be linked and the name of the variant. The statechart's code together with the build automation files result in the `Intermediate Source Code`. Autotools link existing source code artifacts with the statechart's code and create a concrete makefile which is contained in the `Variant Source Code`. The existing source code artifacts may be annotated with pre-processor directives, whose values may be filled by `Autotools`. The `Cross-Compiler` compiles the source code using the provided makefile. The binary code is then deployed to the end device.

On the end device, the new binary is installed. Subsequently, it collects sensor data and sends it to the cloud. The sensor data is processed by HyVarRec which, if necessary, computes a new configuration and then starts a new loop of the variant-generation phase.

Our architecture is strongly component based. Distributed to the cloud, it achieves very flexible scaling. For instance, if the `Yakindu code generator` is very slow, multiple instances of it could be created in the cloud. In this paper, we explain the complete toolchain except for the deployment, the reconfiguration process (parts connected with dashed arrows in Figure 3) and the cloud technology itself. The reconfiguration process is explained in [13,12].

5 Software Product Line Design Phase

For designing the SPL, we use DeltaEcore and Yakindu statechart tools. With DeltaEcore, it is possible to define the complete SPL in terms of the problem space, solution space (based on metamodeling) and configuration knowledge. In the following, we describe how to specify an SPL for our use case, using DeltaEcore, Yakindu and our provided extensions to it.

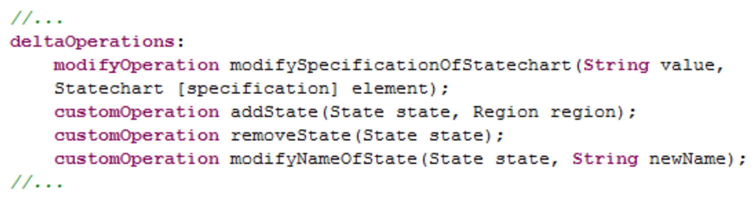
5.1 Core Statechart

As DeltaEcore is relying on delta modeling, which is a transformational approach, we need to specify a core statechart that can serve as source for transformation. In our toolchain, we specify the core statechart by using the standard Yakindu statechart editor. This allows us to use the complete functionality provided by Yakindu and does not break the workflow with statecharts that system engineers are used to. This core statechart is then modified by delta modules to create variants.

5.2 Managing Variability

In the problem space, we use FMs which are implemented in DeltaEcore. DeltaEcore provides an editor, which allows us to define FMs. Figure 2 depicts an FM representing the problem space of our running example.

In the solution space, DeltaEcore relies on delta modeling for metamodels. To be able to transform these models, it is necessary to define a delta language for each metamodel, which consists of a common base delta language and a metamodel-specific delta dialect. In our case, we need to define two delta dialects. One for the statecharts and one for the metadata. Figure 4 shows an



```
//...
deltaOperations:
    modifyOperation modifySpecificationOfStatechart(String value,
    Statechart [specification] element);
    customOperation addState(State state, Region region);
    customOperation removeState(State state);
    customOperation modifyNameOfState(State state, String newName);
//...
```

Fig. 4: Excerpt of delta dialect for Yakindu statecharts

excerpt of the delta dialect for the Yakindu statecharts. With this dialect, it is possible to modify an existing statechart, e.g., by adding, removing or modifying states and transitions. Moreover, it is possible to modify the specification of the statechart via the delta operation `modifySpecificationOfStatechart(...)`. In the specification of a statechart, it is possible to define variables, events, operations and interfaces. This is very important as these operations can be used to call existing external code artifacts. This will be explained later in more detail.

The second dialect is defined to create metadata, which can be seen in Figure 5. As the metadata is not available as Ecore metamodel, we only use the delta dialect to trigger operations provided for the metadata libraries. To this end, we introduced the operations `preMetadata()` and `postMetadata()`, which initialize and write out the metadata, respectively. The delta operation `addSourceFile(...)` defines external code artifacts which should be integrated by Autotools and which should be linked to operations of the statechart. With

```
deltaOperations:
  customOperation preMetadata();
  customOperation addSourceFile(String filename);
  customOperation addFeature(String feature);
  customOperation setVariantName(String variant, String version);
  customOperation postMetadata();
```

Fig. 5: Delta dialect for metadata

the delta operation `addFeature(...)`, the values for annotative variability in the external code artifacts is set. Finally, the `setVariantName(...)` delta operation is used to provide information on how to pack the final binary of a variant.

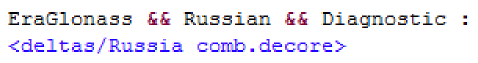
With these two dialects, it is possible to define delta modules consisting of a set of delta operations. Figure 6 depicts an excerpt of delta module of our

```
delta "Russia_Comb"
        dialect <http://www.yakindu.org/sct/sgraph/2.0.0>
    modifies <../core/eCall.sct>
{   //...
State initAdditionalDataState = new State(name: "init_additionaldata");
addState(initAdditionalDataState,
<eCall.main region.init_ecallmessage.InitEcallMessage>);
addTransition(<eCall.main region.init_ecallmessage.InitEcallMessage.init_ecallmessage>,
<eCall.main region.init_ecallmessage.InitEcallMessage.init_additionaldata>,"always");
    //...
}
dialect <http://eu/hyvar/metadata>
{
preMetadata();
addSourceFile("encode_optionaldata.c");
//...
addFeature("Diagnostic");
//...
setVariantName("russia_comb", "1.0");
postMetadata();
}
```

Fig. 6: Excerpt of a delta module of the running example

running example. Among others, this delta module adds a state which is responsible for adding diagnostic information of the feature "Diagnostic" of Figure 2. In each delta module, a core model can be specified with the keyword `modifies`, which should be modified by the delta module. Additionally, it is possible to define whether a delta module is dependent on another delta module via the keyword `requires`. As Figure 6 shows, in one delta module, multiple delta dialects can be used. In our case, this is sensible, as we define operations in the statechart's specification which need to be linked to existing code artifacts. Thus, it is necessary to specify the respective code artifact to be integrated. However, in each delta module, it is possible to reference existing elements of the modified model. For example, the `addState(...)` operation adds the newly created state to an existing region of the statechart. In particular, the state `initAdditionalDataState` is added to the region identified by `<eCall.main region.init_ecallmessage.InitEcallMessage>`. Additionally, a new transition is added from an existing state to the new state. For the metadata, a source file, i.e., the `encode_optionaldata.c`, is marked for integration. Moreover, it is specified that the feature `Diagnostic` has been selected and the variant's name is `russia_comb` in version `1.0`.

In the problem space, we defined the features of our product line. In the solution space, we defined delta modules, which specify how a given statechart is modified and which existing code artifacts should be integrated. It is further necessary to link the problem and the solution space. This is done by the configuration knowledge, consisting of mappings between features and delta modules. Figure 7 depicts part of the configuration knowledge for our running example.



```
EraGlonass && Russian && Diagnostic :
<deltas/Russia comb.decore>
```

Fig. 7: Exemplary mapping of features to a delta module

First, an application condition needs to be specified. This is a Boolean expression on features, defining in which feature selection this mapping should be executed. In the example, the application condition is `EraGlonass && Russian && Diagnostic`, saying that the following deltas should only be applied if the features "EraGlonass", "Russian" and "Diagnostic" of our running example are selected together. The second part is a list of delta modules that should be applied if the application condition is true. In our example, this is only one delta module, i.e., the `Russia_comb.decore`.

6 Variant-Generation Phase

The second phase of the workflow consists of the variant generation. The variant generation is triggered by receiving a new configuration. For our running example, this could be a configuration consisting of the features "EmergencyCall", "Base", "Languages", "GNSS", "EraGlonass", "Russian" and "Diagnostic". After having the SPL defined, the component `DeltaEcore Variant Generation` uses the SPL artifacts and selects the delta module according to the feature selection in the configuration and the mapping in the configuration knowledge. Assuming the above mentioned configuration and the mapping of Figure 7, this would be the delta module `Russia_comb.decore`. Then, the delta modules are applied to the core statechart. According to the delta module `Russia_comb.decore` depicted by Figure 6, a new state for collecting the diagnostic information and an external operation call would be added. Moreover, new source files are specified in the metadata. As a result, this component produces the variant model (i.e., the variant's statechart) and the build automation files. In our case, the build automation files consist of an .am file and an .ac file, which are both used as input for `Autotools`. Assuming the application of the `Russia_comb.decore` delta module, the build automation files contain references to the newly added source files.

In the next step, the `Yakindu Code Generator` receives the variant's statechart as input and generates code out of it. In our case, this is currently C code but it is also possible to generate C++ or Java code. However, this code includes interfaces/header files, which represent the operations of the statechart (see Section 5.2). These interfaces/header files do not yet have an implementation but will be linked in a following phase.

Figure 8 illustrates the general workflow of the `Autotools` component. As

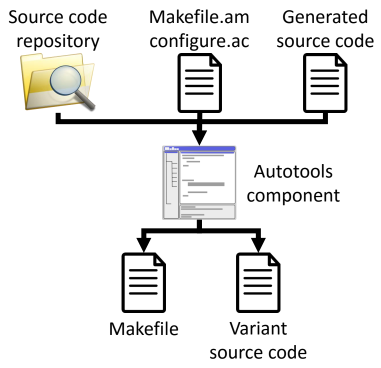


Fig. 8: Overview on the workflow of the `Autotools` component

input, the `Autotools` component receives the build automation files (i.e., the `Makefile.am` and the `configure.ac`), the source code repository, in which existing code artifacts are stored and the generated code from the variant's statechart. It processes these artifacts in three phases: In the first phase, it collects the existing code artifacts from a repository, which were specified in the metadata (see Section 5.2). In the second phase, the generated interfaces/header files of the `Yakindu Code Generator` are linked to the concrete implementations of the existing code artifacts. In the third phase, a makefile is created as Figure 1 illustrates. The `Autotools` component's output is the variant's source code, i.e., the generated source code and the linked existing code artifacts as well as the makefile.

The `Cross-Compiler` component receives the variant's source code as well as the makefile and compiles the code for a certain target platform. In our case, the target platform is the Autonomous Telematics Box (ATB2) which is based on an ARM platform. As a result, we receive a binary, which is deployed to the end device. In our case, the binary code is packed to a firmware image for the ATB2 and then deployed. For brevity, we do not explain the process of deployment in this paper. After having a new firmware installed, the end device continuously collects sensor data and sends it to our cloud infrastructure. The component `HyVarRec` receives the data as input and, if necessary, computes a new configuration. This new configuration triggers a new cycle of the variant generation phase. However, a description of how `HyVarRec` works can be found in [12,13].

7 Related Work

A prominent framework for research on Feature-Oriented Software Development is FeatureIDE [21], which is based on the Eclipse platform[7]. FeatureIDE provides comprehensive tools for variability modeling with feature models as well as the ability to include different variability realization mechanisms. However, FeatureIDE focuses on the development of SPLs, their analysis, configuration and finally, the variant derivation for different variability realization mechanisms. In contrast, we present an integrated toolchain for the development and deployment of software variants for distributed host devices.

DeltaEcore [20] is a tool suite for delta modeling on languages based on EMF Ecore metamodels. Using DeltaEcore, it is possible to semi-automatically create a delta languages for a particular EMF-based target language consisting of delta operations specific to that target language. Moreover, DeltaEcore provides comprehensive tools for variability modeling and configuration as well as variant derivation using the custom delta languages. In this work, we employ DeltaEcore for the development of SPLs and the variant derivation. However, using our toolchain, we further process the model variant by producing an executable variant and deploying it to the target device (cf. Section 4).

DeltaJava [17,11] is a custom-tailored delta language for the Java programming language. Thus, DeltaJava provides delta operations to transform legacy Java software to another variant of that software. However, DeltaJava is tailored to Java, whereas our target language is C/C++. To this end, FeatureC++ [3], employing FOP as the variability realization mechanism for SPLs based on C/C++, would be more suitable. However, delta modeling allows for more flexibility and improves expressiveness over FOP as elements may be removed. Other implementation approaches to SPLs include FeatureHouse [2], a language-independent composition tool that requires a general structure model of the base language called Feature Structure Tree (FST), as well as AHEAD [4], an algebraic foundation for module composition, which is implemented for Java.

pure:variants[8] is a tool suite that uses family models as 150% models of supported realization artifacts. Configuring these family models, a variant can be generated. BigLever's[9] Gears Product Line Engineering Tool and Lifecycle Framework^{TM} is an SPL development tool suite based on feature models and an annotative variability realization mechanism. However, neither pure:variants nor Gears Product Line Engineering Tool and

[7] `http://www.eclipse.org`
[8] `https://www.pure-systems.com`
[9] `http://www.biglever.com`

Lifecycle Framework^{TM} support delta modeling as a variability realization mechanism. Moreover, pure:variants does not support C/C++ as the target language.

Clafer Tools [1] is an integrated set of tools based on Clafer, a language for structural modeling of SPLs. Using Clafer Tools, variability modeling, configuration, and variant derivation of structural models (e.g., class diagrams) can be performed. However, Clafer Tools does not support behavioral modeling of systems, such as statecharts. Therefore, generating executable variants is not possible using Clafer Tools, which is essential to our work.

8 Conclusion and Future Work

In this paper, we presented an integrated toolchain for the development and deployment of increasingly individualized software that is adapted to changing contexts. To this end, we employed SPL engineering in a model-driven context where realization artifacts are modeled as Yakindu statecharts and variability is realized using delta modeling with DeltaEcore. After variant derivation, the model variant is further processed to derive executable C/C++ code, including metadata consisting of build automation files. To allow a flexible and parallel reconfiguration, the variant derivation, linking and compilation processes are performed in a cloud infrastructure. Using the industry-related case of emergency response systems, we showed the feasibility of our toolchain for developing the SPL and deriving and deploying specific variants to target devices.

In the future, we plan to extend the application of the toolchain to a different and more challenging scenario where the generation of Java code is required. Moreover, we plan to support evolution of features (i.e., feature versions) [19] and evolution of the feature model [14] within the toolchain. This way, we would enable distributing different (e.g., older) versions of particular (parts of) variants.

Acknowledgments

This work was partially supported by the European Commission within the project HyVar (grant agreement H2020-644298), by ICT COST Action IC1402 ARVI (`www.cost-arvi.eu`), by the DFG (German Research Foundation) under grant SCHA1635/2-2, and by the Ateneo/CSP D16D15000360005 project RunVar. We thank Christoph Seidl from Technische Universität Braunschweig for sharing with us his useful comments and knowledge on DeltaEcore in particular and also the helpful comments of the anonymous reviewers.

References

1. M. Antkiewicz, K. Bąk, A. Murashkin, R. Olaechea, J. Liang, and K. Czarnecki. Clafer tools for product line engineering. In *Software Product Line Conference*, Tokyo, Japan, 2013. Accepted for publication.

2. S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-independent, Automated Software Composition. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society.
3. S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, GPCE'05, pages 125–140, Berlin, Heidelberg, 2005. Springer-Verlag.
4. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proc. of ICSE 2003*, pages 187–197. IEEE, 2003.
5. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371, 2004.
6. D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. *ACM SIGPLAN Notices*, 46(2):13, 2011.
7. J. Greenfield and K. Short. Software factories: Assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 16–27, New York, NY, USA, 2003. ACM.
8. ISO/IEC 9899:1990 - Programming Languages—C. Standard, International Organization for Standardization, 1999.
9. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon Software Engineering Institute, 1990.
10. K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5(1):143–168, Jan. 1998.
11. J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and F. Damiani. Deltaj 1.5: Delta-oriented programming for java 1.5. In *Proc. of PPPJ 2014*, pages 63–74. ACM, 2014.
12. J. Mauro, M. Nieke, C. Seidl, and I. C. Yu. Context Aware Reconfiguration in Software Product Lines. *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '16*, 2016.
13. M. Nieke, J. Mauro, C. Seidl, and I. C. Yu. User Profiles for Contet-Aware Reconfiguration in Software Product Lines. In *7th International Symposium on Leveraging Applications, ISoLA 2016, Imperial, Corfu, Greece*, 2016. In this volume.
14. M. Nieke, C. Seidl, and S. Schuster. Guaranteeing configuration validity in evolving software product lines. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '16, pages 73–80, New York, NY, USA, 2016. ACM.
15. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques.* Springer, 2005.
16. C. Prehofer. *Feature-oriented programming: A fresh look at objects*, pages 419–443. Springer Berlin Heidelberg, 1997.
17. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-Oriented Programming of Software Product Lines. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond (SPLC 2010)*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2010.
18. I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012.

19. C. Seidl, I. Schaefer, and U. Aßmann. Capturing variability in space and time with hyper feature models. In P. Collet, A. Wasowski, and T. Weyer, editors, *The Eighth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '14, Sophia Antipolis, France, January 22-24, 2014*, pages 6:1–6:8. ACM, 2014.
20. C. Seidl, I. Schaefer, and U. Aßmann. Deltaecore - A model-based delta language generation framework. In *Modellierung 2014, 19.-21. März 2014, Wien, Österreich*, pages 81–96, 2014.
21. T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. Featureide: An extensible framework for feature-oriented software development. *Sci. Comput. Program.*, 79:70–85, Jan. 2014.