Certifying delta-oriented programs

SoSyM manuscript No.
(will be inserted by the editor)

Certifying Delta-Oriented Programs

Vítor Rodrigues · Simone Donetti · Ferruccio Damiani

Received: date / Accepted: date

Abstract A major design concern in modern software development frameworks is to ensure that mechanisms for updating code running on remote devices comply with given safety specifications. This paper presents a delta-oriented approach for implementing product lines where software reuse is achieved at the three levels of state-diagram modeling, C/C++ source code and binary code. A safety specification is expressed on the properties of reusable software libraries that can be dynamically loaded at run-time after an over-the-air update. The compilation of delta-engineered code is certified using the framework of proof-carrying code in order to guarantee safety of software updates on remote devices. An empirical evaluation of the computational cost associated with formal safety checks is done by means of experimentation.

Keywords model-driven development · delta-oriented programming · safety properties · proof-carrying code · run-time systems

1 Introduction

Software-intensive systems, like those of modern automobiles, can include up to one hundred small embedded microcontrollers [10, 46, 4, 4]. The trend in such distributed embedded systems is to become increasing complex and, at the same time, to be often subject to changes in functionality. Production environments, where *over-the-air* (OTA) software updates are required after the initial deployment pose specific challenges in terms of integrity of the adaptive system [81, 89]. In these kind of environments, formal verification techniques

This work was partially supported by the European Commission within the project HyVar (http://www.hyvar-project.eu/), grant agreement H2020-644298.

University of Turin
E-mail: vitor.rodrigues@di.unito.it
E-mail: {ferruccio.damiani, simone.donetti}@unito.it

are required to provide protection against code that violates safety specifications of the embedded system. Additionally, protection against untrusted code suppliers require efficient on-board verification mechanisms.

The main obstacle to deploy verified embedded software is the amount of computational effort necessary to perform formal analysis of OTA updates [11]. When formal verification is performed on the remote device, objective measurements of the software system are required in several critical areas: number of updates, static program size/execution metrics, data coupling/binding and modularity [81, 7, 45]. Therefore, empirical evaluation of experimental data using descriptive statistics needs to be incorporated in the software development process in order to guarantee consistency and ease maintenance [6].

A *Software product line* (SPL) [2, 23, 63] is a collection of programs, called *variants*, that are developed from a common codebase and have well documented commonality and variability. *Delta-oriented programming* (DOP) [70] (see also [2, 6.6.1]) is a flexible transformational approach to implement SPLs. *Model-driven development* (MDD) [39, 46] exploits rich domain models (e.g. statechart diagrams [42, 30]) to represent product specifications in terms of abstract representations within a given domain knowledge [66]. *Proof-carrying code* (PCC) [58] is a software technique that allows a host system, like a remote embedded device, to verify properties about an application by exporting a formal proof, called *certificate*, that is carried together with the executable code. The remote device decides whether the supplied code is safe to execute by checking the validity of the certificate.

This paper presents an exploratory study about using DOP in connection with MDD and PCC to support safety of software updates on remote devices. Experimental evaluation is used to aid software designers, developers and maintainers in the process of developing the most effective SPL regarding the use of variability in delta-modeling *versus* the cost of formal verification of OTA updates. The first objective of this paper is to postulate the hypothesis of using an iterative heuristic algorithm where the best trade-off between these two conflicting factors can be found by experimentally-acquired evidence.

The second objective of this paper is to assess the flexibility of the proposed software development methodology. The design choice to apply delta-oriented techniques ubiquitously to the whole framework and the possibility given to designers/developers/maintainers to simulate the PCC overhead on the remote device, helps to define the modularity of the software system. In particular, the process of delta-modeling of procedural models, enriched with delta-modeling of the physical structure of the software system, is carried out in such a way that the amount of transferred data during OTA updates is minimized, at the same time that an efficient PCC-based formal verification mechanism is devised by configuration.

The exploratory study presented in the paper makes simplifying assumptions on the pilot use-case scenario. The modeling/deploying framework is described using a *feature model* inspired by the ECall/E112 Regulation scenario [32], which is considered in the demonstrator of the HyVar project. The integration of the proposed approach in a full-fledged tool chain, like the one

developed by the HyVar project [22, 68], is a major project that remains as future work. Considering only two *executable variants* is sufficient to identify which techniques can be used to provide a quantitative assessment of the contribution. For this purpose, results about the computational cost of the formal verification and the scalability of the proposed framework are guaranteed to be correctly interpreted by means of statistical analysis.

The paper is structured as follows. Section 2 discusses related work, background and motivation. Section 3 presents the modeling/deploying framework by means of a simple example to be used as a running example throughout the paper. Examples of variability realization artifacts, such as finite-state machines and code-level models, are introduced. Section 4 gives an overview on the engineering process of developing delta-oriented SPLs using different kinds of models and Section 5 describes a formal verification mechanism to check cross-consistency between those models. In Section 6, a PCC mechanism used to validate OTA updates according to the safety policy is described. Section 7 presents the experimental context, the correspondent data evaluation, and discusses threats to validity. Section 8 concludes the paper.

2 Related Work, Background and Motivation

2.1 Quantitative Evaluation of Software Quality

Extensive work is reported in the literature describing methodologies to plan and carry out experimentation in software engineering [86, 81, 5, 6, 83, 7, 45, 16]. In particular, process-oriented experimental evaluation [16], focus on assessing the impact of a technology push in order to understand how it can improve performance or increase software quality. Referred examples are those of new design methodologies, programming languages or software configuration management techniques. The existence of a conceptual framework supporting rigorous data gathering and the use of quantitative software evaluation techniques is signaled as fundamental [33, 79].

Experimental data helps to identify design decisions that can significantly improve understanding of language design for reliable software and the maintenance process. Along these lines, flowcharts and program design languages (PDL) were evaluated in [65]. With the objective to improve program composition, comprehension, debugging and modification, flowcharts are also evaluated in [74]. The present paper applies delta-modeling to both state diagrams [64] and source code [31] in order to identify which program functionalities have the highest potential for reuse [9]. Additionally, the proposed modeling approach identifies possibilities for reuse at binary level by integrating build-system models [80, 13] into the DOP-SPL.

Contrary to the framework presented in [16], where descriptive models used for domain analysis [47] are not intended to be formally validated, the DOP approach presented in this paper is proved to facilitate cross-validation between the collaborative models that are used to generate SPL variants.

Moreover, in contrast with [81], OTA updates by means of delta operations allow "patches" to be frequently applied and this facilitates program analysis of partial implementations. In this way, SPL variants are by-products resulting from successive increments to an initial (skeletal) subproject, but with the guarantee of consistency of the final software product.

One fundamental follow-up that contributes for the impact of an experimental evaluation study is the guarantee of replication and the representativeness of the experimental data, hence allowing the extrapolation of the results to other environments [6] and the refinement of the posed hypothesis [86, 83, 16].

In this way, the use of a statistical framework which validates the assumptions made during experimentation to obtain quantitative measures (obtained, for example, using profiling techniques [48]), are independent from any problem domain and allow the confirmation or refutation of the hypothesis by experimental evidence or by operational demonstration.

2.2 Implementation of Software Product Lines

Approaches in the literature to SPL implementation can be classified into three main categories [71]: *annotative approaches* expressing negative variability; *compositional approaches* expressing positive variability; and *transformational approaches* expressing both positive and negative variability. In annotative approaches all variants are included within the same model (often called a 150% model). An example of an SPL annotative approach is represented by C preprocessor directives (`#define FEATURE` and `#ifdef FEATURE`). *Feature-Oriented Programming* [8] (FOP) and DOP are examples of compositional and transformational approaches, respectively.

FeatureIDE [78] is an open source framework of an Integrated Development Environment (IDE) for SPL engineering. It supports the entire life-cycle of a product line by providing comprehensive tools for variability modeling with feature models using different variability realization mechanisms. Similarly to our approach, FeatureIDE covers the design, implementation and maintenance of software product line engineering, including domain analysis and feature modeling. However, since FeatureIDE does not support delta modeling at object-code level, it is not a practical solution to perform OTA updates directly on binary code running on remote devices.

DeltaEcore [73] is a tool suite tailored for DOP using Eclipse Modeling Framework (EMF)[1] Ecore metamodels. Using DeltaEcore, it is possible to semi-automatically create a delta language 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 custom delta languages.

The main characteristic of DeltaEcore is that delta languages are required to be defined using Ecore metamodels. In this way, DeltaEcore

[1] http://www.eclipse.org

supports variant generation of behavior models such as YAKINDU statecharts [38]. For a complete description of automatic code generation with YAKINDU statecharts and reuse of C/C++ implementation artifacts, the reader is referred to [22]. The main limitations of the toolchain presented in [22] are the absence of cross-validation between delta languages, formal verification delta actions at object-level, and configurable compilation mechanisms.

The concept of Dynamic SPLs has been proposed in [41] to demonstrate that dynamically adaptive systems exhibit degrees of variability that depend on user needs and runtime fluctuations in their contexts. Although the work presented in this paper is also concerned with the use variability at runtime, the objective is not to create a self-adaptive product line. Rather, by employing DOP as the variability realization mechanism, we develop a solution that provides OTA updates on already customized executable variants. The PCC formal verification mechanism is used to check whether the update on the production system is consistent with the safety policy of the remote device.

2.3 Formal Verification of Distributed Embedded Software

Even though the theoretical PCC framework [58] is well-accepted in the research community, practical applications still exhibit size and performance issues, e.g. memory consumption, which are sensitive to industry stakeholders. A pilot project like the `ECall/E122` regulation scenario [32], is an example where quantitative assessment is fundamental for determining the applicability of the verification mechanism. In fact, the use of Theorem Proving as the enabling technology to prove software correctness raises questions about the feasibility of using an all-inclusive demonstrator of PCC. The main difficulty found in the existing literature is to present empirical evidence that can demonstrate that computational requirements can be met.

In alternative, examples in the literature [44, 11, 67] tackle the computational resource problem use the theory of Abstract Interpretation [24]. In [44], the Abstraction-Carrying Code (ACC) framework is proposed to replace a costly verification process by an efficient checking procedure on the consumer side. In particular, a methodology to reduce the size of certificates as much as possible while at the same time not increasing checking time in proposed in [1]. An experimental evaluation of formal verification of Java Bytecode using Abstract Interpretation is proposed in [11] in terms of memory usage. A particular analysis of resource requirements using the worst-execution time (WCET) as the safety criteria is presented in [67].

The present work is an approach to PCC for C/C++ object code where the safety policy is based on the runtime consistency of the remote device [50, 21, 18]. Examples in the literature incur different kinds of runtime overhead: runtime program monitoring [18], signatures of binaries verified at runtime [21] and cryptography hashes complemented with processor-specific constraints verified at runtime [50]. In this paper, formal verification of runtime integrity

is based on textual representations of binaries, i.e. their symbols-table, and on the properties of interest that can be part of a *safety specification*.

To the best of our knowledge, our approach is the first to address the problem of delta-(binary) updates using the theoretical framework of PCC. An empirical evaluation is performed in order to determine if the cost of performing formal verification is feasible to perform on embedded systems. The objective is to provide system maintainers with empirical evidence such that the trade-off between checking time and certificate size can be optimized using conventional operating system technologies.

3 Modeling/Deploying Framework : Principles & Techniques

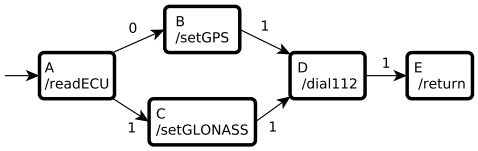
We consider a modeling framework based on state diagrams reusing existing C/C++ artifacts with the aid of code-level design models. State diagrams are graphically represented by finite state-machines (FSMs), which are design models used for describing C/C++ procedures, i.e. a sequence of callable C/C++ functions. Models at code-level are specific to the build system and are used for describing the logical organization of compiled code. Section 3.1 gives an overview of the framework, Section 3.2 briefly recalls the considered finite-state machines and Section 3.3 illustrates the adopted build system.

3.1 Overview

For the purpose of formal verification of the consistency between state diagrams and C/C++ artifacts, we focus on the formal semantics of Moore FSMs. We define a two-letter input alphabet such that the sensible information produced by the FSM is computed by the labeling function. Hence, a string over the output alphabet is used to specify the control-flow of a C/C++ program at function-level. In this case, the state diagrams are called *procedural*, because state transitions do not depend on external events [30]. C/C++ low-level statements are not exposed at the level of the FSM and the input symbols represent possible changes in the *internal* state of the program.

C/C++ programs, like in other procedure-oriented languages, are specified to execute in a step-by-step manner. Informally, the consistency between state diagrams and C/C++ artifacts specifies that for each state labeled with a C/C++ function declaration there must be a file on-disk containing the corresponding function definition. In this way, dangling state declarations are guaranteed not to exist. Conversely, the compilation of the generated code from the FSM against the existing C/C++ artifacts must be minimal in the sense that only the strictly necessary implementation files are compiled.

Figure 1 illustrates a Moore state machine that specifies a simple program inspired by the ECall/E112 Regulation scenario (on the left), and the correspondent build-system model, at code level, that specifies how to compile an executable program named "prog" (on the right).



(a) Moore state machine specifying the use of GPS and GLONASS

```
bin_PROGRAMS = prog
prog_SOURCES = dial1112.c
prog_SOURCES += setGPS.c
prog_SOURCES += setGLONASS.c
prog_SOURCES += readECU.cpp
```

(b) Automake model

Fig. 1 Example of a state-machine model (on the left hand-side) and a GNU Build System model (on the right hand-side) required to compile the executable prog

The state diagram illustrates the sequence of actions when an emergency call is made using a Global Navigation Satellite System (GNSS). This system can be either the Global Positioning System (GPS) or the GLONASS navigation system. Each state is labeled with function names that implement the executable variants generated by the `ECall/E112` DOP-SPL.

The naming convention for C/C++ functions is that for each C/C++ function there is a file on-disk with the same name as the function. Then, the input alphabet of the Moore finite-state machine presented in Section 3.2 is used to redirect the control-flow to either the functions "setGPS" or "setGLONASS". The input symbols 0 and 1 of the FSM are used to specify the value of some global (shared) variable denoting the GNSS being used. The value of such variable is modified by the "readECU" function.

The novelty in the models presented in Figure 1 is that the structure of object (embedded) code is lifted to the modeling space by writing GNU Build System [19] models. By extending the modeling space in this way, the development of embedded code, i.e. code that is deployed to remote devices, gains in flexibility and modularity: for example, the code implementing each state diagram can be compiled as a static library, a dynamic library, an executable, or as a combination of the previous, albeit without changes in functionality.

These decisions have a significant impact on the efficiency of OTA updates. When considering binary patches on mobile code, specially if a great amount of code can be reused, a *one-size-fits-all* solution for compiling software is inadequate because one needs to take into consideration the bandwidth and the safety requirements of the remote device [18, 43]. In one hand, mobile code compiled as a single executable is simple to design but makes OTA updates very inefficient in terms of download time because to the size of the patch is necessarily higher than when using shared libraries.

On the other hand, the use of dynamic shared libraries requires integrity checks before installation and their use can delay the start-up time of the executable [87]. For example, as opposed to a monolithic approach to executable variant generation, the creation of a shared library for each source-code file may appear a reasonable choice because it maximizes the freedom of the user during the modeling phase. However, since each library is individually dynamically loaded, the loading time increases significantly [17, 28].

When developing a DOP-SPL, this aspect is of particular interest because designers/developers/maintainers are able to define the amount of internal

fragmentation of an executable variant. The overhead of shared libraries directly affects the efficiency of the formal checking mechanism because system integrity on the embedded system must be enforced statically before runtime. A typical example is the detection of the *diamond problem* when using dynamically linked libraries [40, 26, 27].

Consider the scenario where two fragments of the executable depend on different incompatible versions of the same library. The problem is when shared libraries are loaded, all the *undefined symbols* (cf. Section 3.3) are resolved inside the same namespace. To ensure safety, two versions in use of the same function with the same name, even if they are in different libraries, must not co-exist because it is not possible for the dynamic linker/loader to choose which version is exposed to the binary containing the dependency.

For this reason, possible conflicts and inconsistencies with pre-installed software on the remote device must be checked before performing the OTA update by using system-level tools, e.g., Linux commands. For instance, an update could contain an executable that "conflicts" with the currently installed version of the same executable. In this case, the reconstruction of binaries by means of patches on the remote device uses a run-time system that is substantially different from the one used by the code supplier. Thus, verification mechanisms are required to avoid compromising the integrity of the run-time system and, consequently, jeopardize the ability to safely update the system.

3.2 A Brief Introduction to Finite-State Machines

The semantics of state diagrams is that of a finite-state machine, i.e. a finite automata enhanced with an *output alphabet* [54, 29]. Because the class of C/C++ programs under consideration are purely procedural where each state specifies a sub-routine to be executed, we use finite-state transducers that have a two-letter *input alphabet* and one final state [61]. When compared to finite automata as language acceptors, the goal of the transducer is not simply to accept or reject input strings over the input alphabet, but to generate a set of output strings over Λ given a set of input strings over Σ. The set of output strings correspond to valid C/C++ function specifications.

Since the purpose of the transducer is to model the `main` function C/C++ programs, we focus on a particular class of Moore machines that reject the input string if the output string does not end with the letter `return` $\in \Lambda$. Formally, the Moore (Finite) state-machine (FSM) is a 6-tuple $(S, S_0, \Sigma, \Lambda, T, G)$:

- S is a finite set (whose elements are think of as *states*)
- $S_0 \in S$ (the *initial state*)
- Σ is a finite alphabet of *input symbols*
- Γ is a finite alphabet of *output symbols*
- T is a function from $S \times \Sigma$ to S (the *transition function*)
- G is an output function from $S \times \Sigma$ to Λ (the *state labeling function*)

For any state $s \in S$ and any symbol $a \in \Sigma$, $T(s, a)$ is interpreted as the state to which the FSM moves when the current state is s and the input is a.

A *language* is simply a set of strings involving symbols from some alphabet. For instance, if the input alphabet is the singleton alphabet is defined as $\Sigma = \{0, 1\}$, example strings over Σ are $\langle 1 \rangle$, $\langle 1.0 \rangle$, $\langle 1.0.1 \rangle$ and so forth.

The same reasoning applies to the output alphabet. Using the terminology of the ECall/E122 Regulation scenario, the output alphabet is defined as $\Lambda = \{\mathsf{readECU}, \mathsf{setGPS}, \mathsf{setGLONASS}, \mathsf{dial112}, \mathsf{return}\}$, where "ECU" is the *Electronic Control Unit*, "GPS" is the global navigation satellite system in western countries and "GLONASS" is the Russian satellite navigation system. Then, a possible and valid string over Λ is $\langle \mathsf{readECU} \,.\, \mathsf{dial112} \,.\, \mathsf{return} \rangle$.

The identification of FSMs and formal-language terminology with the software modeling counterparts is straightforward: the output alphabet corresponds to the *domain of discourse*, i.e. the `ECall/E122 Regulation` scenario; an output symbol represents a C/C++ function call; and an output *scenario* is an output string that specifies the control-flow of a C/C++ program.

3.3 A Brief Introduction to the GNU Build System

The compilation and installation of software systems in distributed embedded systems is a difficult task because of their dependencies on the run-time system. In spite that the many Linux distributions lack a uniform software installation mechanism, this procedure often requires the user to manually edit Makefiles and configuration headers [75]. The GNU Project[2] is special in its approach because it defines auto-configurable source packages, which means that programs are compiled from source and installed automatically in many different environments, without any user intervention.

The GNU Build System consists of three main tools: Autoconf, Automake and Libtool [82].The language of Automake is a logic language where no explicit order of execution is pre-given. The logic relations declared inside an Automake model specify the required make directives. The advantage of using these models is that they abstract from the compilation environment by means of a text-replacement mechanism provided by Autoconf named macro expansion [49]. Libtool hides the complexity of using shared libraries behind a consistent, portable interface.

In the environment of distributed embedded systems, the build-system tool suite is not required to be installed on remote devices after configuration because cross-compilation is efficiently automated. Moreover, Makefile targets can be added to make possible the creation of a *binary distribution*, usually in the form of an RPM. The RPM packaging system[3] is widely used in Linux operating systems to manage the operations of installation, querying and deletion of RPM packages on the system. DeltaRPM[4] is a notable mechanism to create *delta binaries* using directly RPMs. These binary distributions are commonly referred to as delta-RPMs [3].

[2] https://www.gnu.org/gnu/
[3] https://www.rpm.org
[4] https://fedoraproject.org/wiki/

When a program is made of many source files and once the Makefile is created, the best procedure is to compile each source file into a separate object file. Object files contain definitions of variables and subroutines written out in assembly. Most of these definitions will eventually be embedded in the final executable at a specific address, i.e. each variable and subroutine will have an assigned memory address, albeit relative to the program's memory address space. In fact, at compile time, the absolute memory addresses are not yet known so they are referred symbolically. These symbolic references, which are called *symbols* [28], are the means to link the object files together.

The translation of the symbol's relative addresses to absolute memory addresses is not allowed inside the object file. Dynamic linking is accomplished by giving to the program's referring addresses the offset between the object file and the executable. If a given symbol is *undefined*, this is, it is externally defined in a shared object file, then the linker will dynamically map the whole shared object into the address space of the program. Such an object is named a Dynamic Shared Object (DSO). Memory addresses outside the DSO are treated as inter-library dependencies.

This can be a difficult task because inter-library dependencies must be correctly handled by the operating system in a uniform way. LIBTOOL provides compilation and linkage of portable shared libraries using a consistent and portable interface. However, in order to build a dynamic shared library it is necessary to remove position-dependent addresses from the object file archived inside that library. This is the reason why the (relative) complexity of shared libraries incur overhead.

4 Round-Trip Engineering using Delta-Oriented Programming

This section describes the process of building a delta-oriented product line using round-trip engineering (RTE) as the technique to implement MDD. We use `ECall/E122 Regulation` scenario considered by the HyVar project [22] to illustrate how DOP can be made transversal to the design-time, compile-time and deploy-time phases through the use of features. In the present section, we focus on the features that specify which global navigation satellite systems (GNSS) can be used: either the GPS system or the ERA-GLONASS system.

Section 4.1 introduces the fundamental concepts of DOP and the terminology to be used throughout the paper. Section 4.2 provides a brief overview of the RTE methodology for developing a DOP product line suitable to development/compilation/deployment of embedded code. Section 4.3 and Section 4.4 describe DOP product lines for state diagrams and for build-system models, respectively. The method for creating delta-RPMs is described in Section 4.5.

4.1 A Brief Introduction to Delta-Oriented Programming

A delta-oriented SPL [70] comprises a feature model, an artifact base, and configuration knowledge. The *Feature Model* (FM) provides a description of

variants in terms of features: each variant is identified by a set of features, called a *product*, where each *feature* ϕ is an abstract description of functionality. The *Artifact Base* (AB) provides the language dependent reusable artifacts that are used to build the variants.

The AB contains a (possibly empty) base artifact and a set of *delta modules* (deltas for short), δ, which are containers to express changes (additions, removal or modifications artifact elements) on the base artifact. Deltas are usually written using *Domain-Specific Languages* (DSLs) [60], where modifications on the core program are expressed through appropriate notations and abstractions [51].

Configuration Knowledge (CK) connects the feature model and the artifact base. It provides an activation condition for each delta (expressed by a proposition formula ψ over features) and an application ordering between deltas. DOP supports automatic generation of variants: given a software product line, the corresponding variants are generated by applying the activated deltas to the base artifact according to the application ordering.

DOP is a generalization of *Feature-oriented programming* (FOP) (see [2, 6.1]), a previously proposed approach to implement SPLs where deltas correspond one-to-one to features and do not contain remove operations. Both FOP and DOP support validation at domain level using family-base type checking mechanisms [56, 77].

4.2 A DOP Product Line With Multiple Artifact-Bases

The cornerstone of the RTE methodology is that conceptual models, such as state diagrams, contribute to a better understanding of the software system. These models can be created either before implementing the physical system, or they can be derived from a system under development [72]. As explained in Section 3.1, we consider procedural-state diagrams and a pre-existent source code package implementing the sub-routines declared in the state diagrams.

As illustrated in Figure 2, RTE offers a bi-directional interaction between behavioral models and source code. Using DOP terminology, behavioral models like procedural-state diagrams constitute one AB and the source code constitute a second AB. Additionally, the delta-oriented approach to SPLs is also used at object-code level in order to create a third AB of textual representations of binary distributions.

Typically, the workflow of the SPL can be described as follows. The development cycle starts with the instantiation of design models in a many-to-many manner: different valid products inside the FM can be associated with a series of state diagrams. Then, a many-to-many first-pass implementation is performed by applying model-to-code transformations to state diagrams and associating the resulting generated code, i.e. state-diagram code skeletons, with existing C/C++ implementation artifacts.

Hereafter, iterations on the behavioral models, including formal validation and refactoring, occur as illustrated in the lower feedback loops of Figure 2.

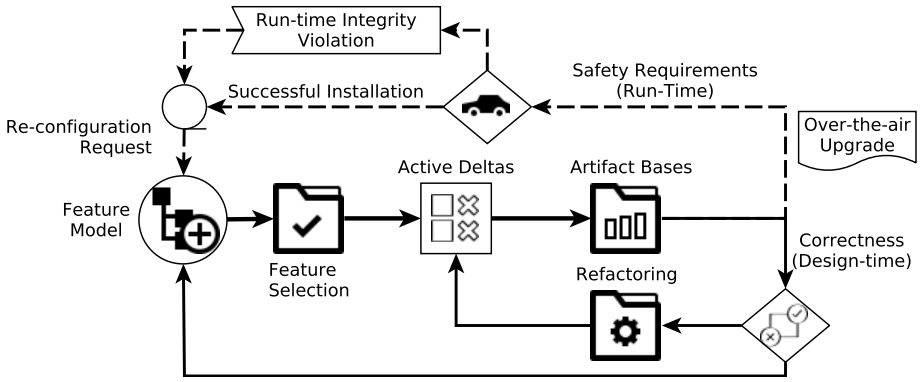


Fig. 2 The Delta-Oriented Software-Product Line for Mobile Code

In each iteration step, new variability realization artifacts can be added to the artifacts bases in order to extend the solution space of the software system. For example, procedural-state diagrams like the Moore FSM in Figure 1(a) can be incrementally updated by applying delta operations on the artifact base of FSMs. Likewise, build-system models like the AUTOMAKE model presented in Figure 1(b), can only be incrementally modified by activated delta modules. Finally, as shown in the upper feedback loops in Figure 2, *deltification* is also performed on RPMs to be updated on remote devices using delta-RPMs.

Considering the scenario of emergency call, there are several initiative to implement this functionality such as the `eCall/E112` program of the European Union as well as the Russian ERA-GLONASS system. By means of a *feature diagram* with *cross tree constraints*, Figure 3 describes the HyVar project demonstrator. There are 4 products that can be generated.

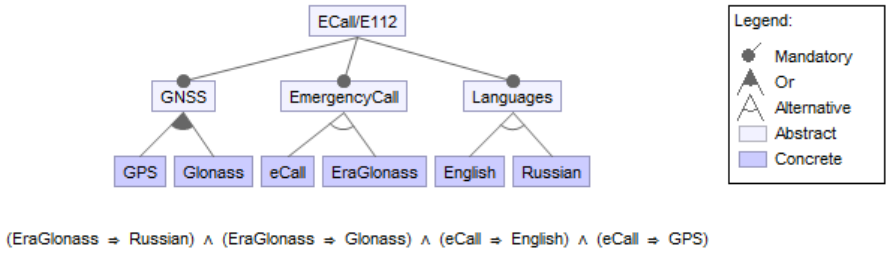


Fig. 3 Feature Model for the SPL inspired by the ECall/E112 Regulation scenario

The *EmergencyCall* feature represents the core system that can be either *eCall* or *EraGlonass*, the *GNSS* feature represents the position system to be used that can be *GPS* (in Europe) or *Glonass* (in Russia), and the *Language* feature represents the language to use, which can be either *English* or *Russian*.

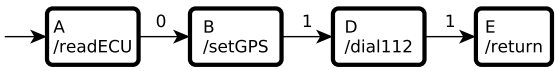
The cross-tree constraints are presented at the bottom of the Figure 3. This expression provides additional conditions on which features must be selected

together to create a valid product. For example, the *EraGlonass* system can only work in Russia and require the *Glonass* positioning system, while the European *eCall* system requires *GPS* positioning.

The AB comprises 4 deltas, named δ_{base}, δ_{gps}, δ_{glonass} and δ_{combo}, respectively. The CK connecting the above FM to the AB specifies the following activation rules:

- ECall/E112 $\Rightarrow \delta_{\text{base}}$
- eCall $\wedge$ English $\wedge$ GPS $\Rightarrow \delta_{\text{gps}}$
- EraGlonass $\wedge$ Russian $\wedge$ Glonass $\Rightarrow \delta_{\text{glonass}}$
- GPS $\wedge$ Glonass $\Rightarrow \delta_{\text{combo}}$

In Section 7, empirical evaluation is focused on a subset consisting of two significant products (executable variants) that can be generated by this SPL. The first executable variant is named "GPS" and is generated upon the selection of the following features: *ECall/E112*, *GNSS*, *GPS*, *EmergencyCall*, *eCall*, *Languages* and *English*; the second is named "GLONASS" and is generated upon the selection of the following features: *ECall/E112*, *GNSS*, *Glonass*, *EmergencyCall*, *EraGlonass*, *Languages* and *Russian*. The state diagrams corresponding to these two variants are presented in Figure 4(a) and Figure 4(b), respectively.



(a) The "GPS" state machine

(b) The "GLONASS" state machine

Fig. 4 Example of products of a DOP-SPL for the `ECall/E112` program

In accordance with the experimental setup, the deltas δ_{base} and δ_{gps} are the realization artifacts used to generate the "GPS" state machine, while the deltas δ_{base} and δ_{glonass} generate the "GLONASS" state machine. In this way the "GPS" variant is generated by the application of $\langle \delta_{\text{base}} ; \delta_{\text{gps}} \rangle$ and the "GLONASS" variant is generated by the application of $\langle \delta_{\text{base}} ; \delta_{\text{glonass}} \rangle$.

4.3 Software-Product Line for State Diagrams

State diagrams provide a high-level abstraction of the behavior of software systems, where computations are abstracted by entities called *states* that are traversed by means of *transitions*. The prominent notation for writing such diagrams is the UML-based *statechart* metamodel [69]. Despite the existence of tool support for state-diagram modeling, this paper defines a delta-oriented

SPL that uses the semantic model of a state-transition machine (STM). In this way, a simple semantics is provided to incrementally create and validate delta-oriented procedural-state diagrams.

Delta modules are containers for a domain language specific to STMs, named State-Diagram DSL. The language constructs of this DSL are designed to provide a simple notation and high-level abstractions to manipulate FSMs. A detailed description of the language constructs of the State-Diagram DSL is given in Table 1.

Table 1 Overview of the State-Diagram DSL for generating Finite State Machines

add(m, s, i, l, n): *adds a new state.* – m is the FSM under design; – s is the state to be added to the set of states S; – $i \in \Sigma$ is the input symbol being read; – $l \in \Lambda$ is the label of the added state given by labeling function: $G(s, i) = l$; – n is the next state of the FSM after reading one symbol from Σ.
modify(m, s, i, t): *modifies a state transition.* – m is the FSM under design; – $s \in S$ is the current state; – $i \in \Sigma$ is the input symbol being read; – t is the *new* "next state" when a symbol from Σ is read: $T(s, i) = t$.
remove(m, s): *removes an existent state*: – m is the FSM under design; – $s \in S$ is the state to be removed; – all transitions from or to s are removed.

Figure 5 shows the delta modules and the correspondent generated FSMs, starting from the base product "base_st". The delta module δ_{base}, shown on the left-hand side, contains the syntax construct add $(M, D, \text{"dial112"}, 1, E)$ which meaning is the addition of a state D to an existing FSM called M, where "dial112" is the label of that state, and E is the "next state". The generated STM, which is named "st1", is shown in the right-hand side as the effect of applying δ_{base} to the empty product by means of "apply $(\delta_{\text{base}}, \text{base\_st})$", where "apply" is the generic function for delta application. The formal definition of the "apply" function will be presented later in Section 6 and Appendix B.

The generation of STMs illustrated in Figure 5 is done according to the following reasoning. The first row defines the base program, which only contains the C/C++ "return" (final) state. The second row shows how to generate an intermediate program in response to the input scenario $\langle 1.1.1 \rangle$. Then, a *run* of the FSM is defined as the sequence of states visited while traversing transitions, as it reads symbols of the input scenario one-by-one [29], until the output symbol return is generated.

Consequently, $\langle \text{readECU . dial112 . return} \rangle$ is the output scenario produced by the FSM generated by the application of δ_{base}. For the generated FSM, the transition functions are: $T(A, 1) = D$, $T(D, 1) = E$ and $T(E, 1) = E$, where

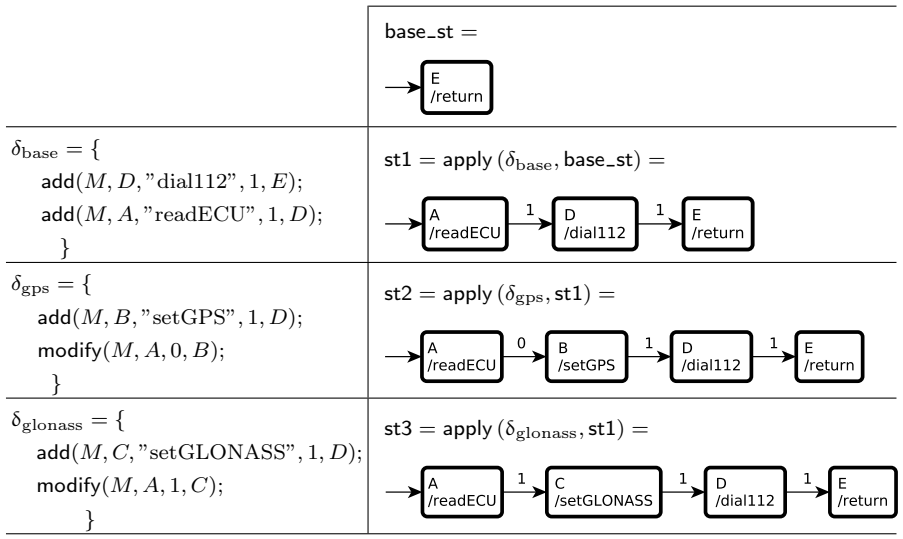


Fig. 5 Example of the correspondence between applied deltas and generated FSMs

$\{A, D, E\} \subseteq S$ and A is the initial state. The output of the FSM is given by the following set of labeling functions: $G(A, 1) = \mathsf{readECU}$, $G(D, 1) = \mathsf{dial112}$ and $G(E, 1) = \mathsf{return}$.

Next, consider the delta modules δ_{gps} and δ_{glonass}. Both deltas are applied to the FSM generated by δ_{base} by means of $\mathsf{apply}\,(\delta_{\text{gps}}, \mathsf{st1})$ and $\mathsf{apply}\,(\delta_{\text{glonass}}, \mathsf{st1})$, respectively. The "GPS" generated FSM receives the input scenario $\langle 0.1.1 \rangle$ and the "GLONASS" generated FSM receives the input scenario $\langle 1.1.1 \rangle$. The first output scenario is $\langle \mathsf{readECU} \,.\, \mathsf{setGPS} \,.\, \mathsf{dial112} \,.\, \mathsf{return} \rangle$ and the second output scenario is $\langle \mathsf{readECU} \,.\, \mathsf{setGLONASS} \,.\, \mathsf{dial112} \,.\, \mathsf{return} \rangle$.

From the observation of the CK, we see that the selection of the concrete features eCall, English and GPS triggers the generation of a FSM declaring the `setGPS` C/C++ function and that the selection of the concrete features EraGlonass, Russian*and*Glonass triggers the generation of a FSM declaring the `setGLONASS` C/C++ function. The C/C++ functions `readECU` and `dial112` are used in both the variants.

4.4 Software-Product Line for the Build System

This section presents the artifact base that abstracts the C/C++ behavioral implementation by considering only the physical realization of artifacts, i.e. filenames. These artifacts are declared in a build-system model like the one presented in Figure 1(b). The benefits of using DOP techniques is to introduce variability in the source-code package and use the FM presented in Section 4.2 in order to promote customized compilation and allow software remodularization. Similarly to Section 4.3, the manipulation of build-system models is made by means of a specific DSL.

In order to build an SPL where the domain of discourse are filenames, the contents of Makefile.am models are manipulated through a small DSL that defines the source-code files that are to be compiled and the output format of that compilation, i.e. a static library, a dynamic library or an executable. For sake of simplicity, it is assumed that each variant contains only one source directory containing a single Makefile.am.

The Build-System DSL defined two syntax constructs: addsource (...) and removesource (...). The source directory may contain several files that are incrementally added or removed to/from a Makefile.am model using the DOP paradigm. The DSL constructs which provide the means to declare source-code filenames is summarized in Table 2.

Table 2 Overview of the Build-System DSL for manipulating Automake models

addsource (f): *adds a source file* to the contents of some Makefile.am. – f is the name of the file on-disk that implements the behavior specified inside the state of a a given FSM.
removesource (f): *removes a source file* from the contents of the Makefile.am. – f is the name of the file on-disk.

Base programs specify the output format of the compilation and linkage processes. For example, assuming that the symbolic name of the compiled program is '`prog`', Table 3 describes the possible AUTOMAKE base programs. The three possible output formats of the compilation are discriminated by means of AUTOMAKE variables. Static libraries are specified by using the variable '`lib_LIBRARIES`' and dynamic libraries use '`lib_LTLIBRARIES`'.

Table 3 Three possible ways to compile the program '`prog`'

Executable	Static Library	Dynamic Library
`bin_PROGRAMS = prog`	`lib_LIBRARIES = libprog.a`	`lib_LTLIBRARIES = libprog.la`

Executables are compiled by means of '`bin_PROGRAMS`'. Additionally, the set of source code files to be compiled are specified inside a list that is mapped by the AUTOMAKE variable suffixed by '`_SOURCES`'. Figure 6 illustrates how to incrementally modify this variable by means of delta actions.

After defining the Build-System DSL, the process of generating variants of the Build-System DOP product line is straightforward. Using the FM previously presented in Figure 3, a set of delta modules is sketched in order to build the two executable variants under consideration: "GPS" and "GLONASS".

The delta actions addsource (...) and removesource (...) illustrated in Figure 6 are used inside delta containers which are named as $\dot{\delta}_{\text{base}}$, $\dot{\delta}_{\text{gps}}$ and $\dot{\delta}_{\text{glonass}}$. Then, the configuration knowledge of the Build-System SPL is composed of the following activation rules:

- ECall/E112 $\Rightarrow \dot{\delta}_{\text{base}}$
- eCall $\wedge$ English $\wedge$ GPS $\Rightarrow \dot{\delta}_{\text{gps}}$
- EraGlonass $\wedge$ Russian $\wedge$ Glonass $\Rightarrow \dot{\delta}_{\text{glonass}}$

	base_am =
	`bin_PROGRAMS = prog`
$\dot{\delta}_{\text{base}}$ = { addsource("dial112.c"); addsource("readECU.cpp"); }	am1 = apply($\dot{\delta}_{\text{base}}$, base_am) = `bin_PROGRAMS = prog` `prog_SOURCES = dial112.c` `prog_SOURCES += readECU.cpp`
$\dot{\delta}_{\text{gps}}$ = { addsource("setGPS.c"); }	am2 = apply($\dot{\delta}_{\text{gps}}$, am1) = `bin_PROGRAMS = prog` `prog_SOURCES = dial112.c` `prog_SOURCES += readECU.cpp` `prog_SOURCES += setGPS.c`
$\dot{\delta}_{\text{glonass}}$ = { addsource("setGLONASS.c"); }	am3 = apply($\dot{\delta}_{\text{glonass}}$, am1) = `bin_PROGRAMS = prog` `prog_SOURCES = dial112.c` `prog_SOURCES += readECU.cpp` `prog_SOURCES += setGLONASS.c`

Fig. 6 Correspondence between deltas modules and generated Automake models

The opportunities for reuse in the Build-System SPL arise from the way in which pieces of software are compiled. As in Section 4.3, the delta module $\dot{\delta}_{\text{base}}$ indicates the possibility to compile the files `dial112.c` and `readECU.cpp` as a shared library, i.e. a DSO (cf. Section 3.3). Then, each executable variant, which name is always `prog`, is compiled using either `setGPS.c` or `setGLONASS.c`, depending on which delta is applied, either $\dot{\delta}_{\text{gps}}$ or $\dot{\delta}_{\text{glonass}}$.

The build-system models created using DOP techniques are used to configure the compilation schemes of the source-code package attaching behavioral implementation to state diagrams. However, despite this flexible methodology to promote reuse and modularity of binary artifacts, it is necessary to correlate each delta sequence defined in Section 4.3 with $\langle \dot{\delta}_{\text{base}}; \dot{\delta}_{\text{gps}} \rangle$ and $\langle \dot{\delta}_{\text{base}}; \dot{\delta}_{\text{glonass}} \rangle$. To this end, a formal validation mechanism according to the semantics of DOP is proposed in Section 5.

4.5 Generation of delta-RPMs using Linux Commands

In this section, we briefly describe the process of building and applying deltas at object-level using Linux commands. Having defined two delta-oriented product lines for state diagrams and the build system, delta actions can be performed on the binary outputs, i.e. the RPMs containing executables and shared libraries, in order to create a binary distribution in the form of a delta-RPM.

As previously mentioned in Section 3.3, delta-RPM packages are binary "patches" to existing RPM packages that can have different names. In other words, delta-RPMs contain only the *binary difference* [62] between an "old" RPM and a "new" RPM. In order to create and reconstruct delta-RPMs, the following commands are respectively used: makedeltarpm[5] and applydeltarpm[6]. A description of these two Linux commands is given in Table 4.

Table 4 Overview of the Linux Delta-RPM suite

makedeltarpm (*oldrpm*, *newrpm*, *deltarpm*); creates a delta-RPM from two RPMs. The result is later used to reconstruct the new RPM from the old RPM.
applydeltarpm (*oldrpm*, *deltarpm*, *newrpm*); reconstructs a new RPM from a delta-RPM by applying a binary "patch" to the old RPM.

Note that delta-RPM is an enabling technology for creating delta-binaries, but this is accomplished using black-box utilities which output requires semantic validation. Section 6 explains how to enforce DOP semantics on delta-RPMs through the verification of the outputs of makedeltarpm on the provider side and of applydeltarpm on the device side. The PCC verification mechanism uses *delta*-table of symbols and defines an elucidating safety property.

5 Validation of Delta-Oriented Software Product Lines

The delta actions contained in each delta module can be the *addition*, *modification* or *removal* of elements of the artifact base (cf. Section 4.1). For example, in the particular case of the build-system product line, the domain of the artifact base is composed by on-disk filenames f, and the delta actions are addsource (f) and removesource (f). However, as stated in Section 3, the use of multiple kinds of models like state diagrams and build-system models required cross-validation at delta-level.

More precisely, consistency means that, given a total order on delta modules, there are no dangling state declarations and the binary program is minimal in the sense that only strictly necessary C/C++ files are compiled. Hence, for each pair of deltas with the same index in the order, the following rules to be satisfied: 1) a filename can only be added to an AUTOMAKE model if some state in the FSM specifies a compatible ouput; 2) a filename can be removed from an AUTOMAKE model only if it exists inside the correspondent FSM.

The perspective on consistency is that the typing rules must be valid for all applicable sequences of delta modules. For this purpose, the *family-based specification* [77] technique is used. In practice, for a particular DOP product line, a specification is assumed to hold for all generated variants. Therefore, the family-based analysis must decide if the specification holds for all possible

[5] https://www.mankier.com/8/makedeltarpm
[6] https://www.mankier.com/8/applydeltarpm

delta sequences that can be applied under a particular FM. In other words, family-based specifications incorporates the knowledge about all possible valid feature combinations and, consequently, all possible deltas that can be applied. Consequently, the checking mechanism considers that all implementation artifacts of all features are merged into a *single virtual product* [77].

In this section, we present a formulation of the consistency criteria using the mechanism of Satisfiability Modulo Theories (SMT) [57]. Given a set of delta modules like the ones defined in Figure 5 and Figure 6, the correctness of a delta-oriented product line is determined by checking the consistency of the virtual product modulo a theory that encode the typing rules over the artifact domains. We start by giving an overview of the SMT formalism in Section 5.1 and then we present a theory for specifying the consistency between State-Diagram SPL and the Build-System SPL in Section 5.2.

5.1 Fundamentals of Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is about checking the satisfiability of logical formulas over one or more theories. It combines the problem of Boolean satisfiability with domains, and term-manipulating symbolic systems. In general, satisfiability is the problem of determining whether a formula expressing a constraint has a solution [15]. In one hand, the most well-know constraint problem is *propositional satisfiability* SAT, where the goal is to decide whether a formula over Boolean variables, formed using logical connectives, can be made true by choosing `true`/`false` values for its variables.

On the other hand, *satisfiability modulo theories* (SMT) combine SAT with theories, e.g. of arithmetic, arrays and uninterpreted functions. It is a *decision problem* that generalizes SAT by replacing sets of Booleans variables by predicates from a variety of underlying theories. Formulas are given a meaning, that is, a truth value from the set $\{\texttt{true}, \texttt{false}\}$, by means of first-order models with signature Σ. The models of interest in SMT are Σ-models belonging to a given Σ-theory $\mathcal{T}$ constraining the interpretation of the symbols of Σ [14].

Since Σ-theories are most generally just one or more (possibly infinitely many) Σ-models, a Σ-formula φ is satisfiable in a Σ-theory $\mathcal{T}$, or $\mathcal{T}$-*satisfiable*, iff there is an element of the set $\mathcal{T}$ that satisfies φ. A set Φ of Σ-formulas $\mathcal{T}$-*entails* φ, written $\Phi \models_{\mathcal{T}} \varphi$, iff every model of $\mathcal{T}$ that satisfies all formulas in Φ satisfies φ as well. Moreover, Φ is $\mathcal{T}$-*consistent* iff $\Phi \not\models_{\mathcal{T}} \bot$, where $\bot$ is the empty clause which truth evaluation is always `false`.

In general, the satisfiability problem for the Σ-theory $\mathcal{T}$ is *decidable* if there is a decision procedure for quantifier-free formulas. Nevertheless, it is possible to find *semi-decision* procedures for formulas of the form $\forall^* \varphi$, where φ is a quantifier-free formula [35]. Since it is infeasible to implement a semi-decision procedure for every possible theory, SMT solvers normally implement meta-procedures for classes of theories that can be described by a *finite number* of closed formulas [57].

5.2 A Theory for Consistency Between DOP Product Lines

For the purpose of specifying a theory for cross-validation between the State-Diagram SPL and the Build-System SPL and, consequently, check the satifisfiability of a particular against that theory, we simplify the domain of the FSM artifact base and consider only the labels of states and abstract from the remaining elements of the Moore FSM definition. Additionally, because filenames are the single entities manipulated by the delta actions defined in the Build-System SPL, the consistency theory is required to take into consideration the index in the application order of pairs of deltas, δ and $\dot{\delta}$, and the "states" and "filenames" that are added or removed inside each container.

The process of defining a theory specifying the notion of consistency starts with the definition of a set of predicates, i.e. functions that return Boolean values, for each delta action. For example, the predicate $\mathsf{add\_state}\,(s, i)$ specifies that the state-label s is added inside the delta module at index i in the application order. Another example is the predicate $\mathsf{remove\_filename}\,(f, j)$, which specifies that the filename f is removed inside the delta $\dot{\delta}$ at index j. These kind of predicates are commonly referred to as *presence conditions* [77].

In order to check that all applicable sequences of delta modules satisfy the above predicates, a *family-based specification* technique is used. Therefore, the correctness specification according to the semantics of DOP must be checked for all valid products in the FM and the correspondent activation rules as defined in the CK. This is equivalent to prove the consistency of a *single virtual product*. To this end, the predicate $\mathsf{ck}\,(f, i)$ is defined, where i is the index of some delta module and f is some propositional formula.

As explained in Section 4.1, the CK maps each product (i.e. a set of features) to a sequence of delta modules (AB). In other words, each product is specified by a propositional formula over selected features that can activate more than one delta module (i.e. a delta sequence). By means of SMT-typing rules, a given virtual product is valid iff there exists a propositional formula in the CK that activates every delta module and each delta module satisfies the cross-validation constraints between SPLs.

Let T be a Σ-theory defined for the DOP-SPL combining the State-Diagram and the Build-System SPLs by means of the Σ-formulas φ_1, φ_2 and φ_3. The sorts contained in the signature Σ are those of *Bool*, *String*, *Index* and *Formula*. The sort *Index* defines delta-module indexes in the total order of deltas. The sort *Formula* is used to distinguish propositional formulas inside the CK.

The set of predicate symbols included in the signature Σ are $\mathsf{delta\_state}$, $\mathsf{delta\_file}$, $\mathsf{add\_state}$, $\mathsf{remove\_state}$, $\mathsf{add\_file}$ and $\mathsf{remove\_file}$. For sake of simplicity, the correctness specification involving the DSL "removal" constructs (Table 1 and Table 2) are omitted in the definition of Theory (1).

In the following, without loss of generality, we assume that activation conditions are written in Conjunctive Normal Form (CNF). Hence, each propositional formula used in the CK to activate a delta module at a given index i, is in the form $\psi_1 \wedge \psi_2 \ldots \psi_k$. Each $\psi_j (1 \leq j \leq k)$ is disjunction of the form $\psi_j^1 \vee \psi_j^2 \vee \ldots \vee \psi_j^h$ where each each $\psi_j^l (1 \leq l \leq h)$ is either a feature ϕ the

negation of a feature $\neg\phi$. Then, the required presence conditions are expressed by the constraint in Eq. (1).

$$\forall\,(f : \mathit{Formula})\ (i : \mathit{Index})\ .\ \mathtt{ck}\ (f, i) \Rightarrow \bigwedge_{k=1}^{n} (\psi_k : \mathit{Bool}) \tag{1}$$

Theory 1: Cross-consistency between "states" and "filenames"

(i) $\mathsf{cross\_c}\ ((i_1 : \mathit{Index})\ (i_2 : \mathit{Index})) : \mathit{Bool}\ := \mathsf{delta\_state}\ (i_1) \wedge \mathsf{delta\_file}\ (i_2))$

(ii) $\mathsf{bijection}\ ((x : \mathit{String})\ (i_1 : \mathit{Index})\ (i_2 : \mathit{Index})) : \mathit{Bool}\ :=$
$(i_1 = i_2) \Rightarrow \neg(\mathsf{add\_state}\ (x, i_1) \oplus \mathsf{add\_file}\ (x, i_2))$

(iii) $\varphi_1 := \forall\,(i\ j : \mathit{Index})\ (x : \mathit{String})\ .\ (\mathsf{cross\_c}\ (i, j) \Leftrightarrow \mathsf{bijection}\ (x, i, j))$

Theory 2: Consistency of the combined DOP-SPL

(i) $\varphi_2 := \forall\,(i : \mathit{Index})\ .\ \exists(f : \mathit{Formula})\ .\ \mathtt{ck}\ (f, i) \Leftrightarrow \mathsf{cross\_c}\ (i, i)$

(ii) $\varphi_3 := \varphi_1 \wedge \varphi_2 \Leftrightarrow \mathtt{true}$

Assume that delta-module instances of a combined delta-oriented SPL are encoded in SMT producing the set of formulas Φ. In order to prove that Φ is T-satisfiable, i.e., to prove the consistency between delta modules belonging to the two different delta-oriented SPLs, it is sufficient to prove that $\Phi \cup \{\varphi_3\}$ is satisfiable, written $\Phi \cup \{\varphi_3\} \not\models_T \bot$. In this way, correctness properties that can be checked against theory T are: 1) there is one propositional formula that must be satisfiable for all declared delta-module indexes; 2) there are no dangling states and the program is minimal in the set of all delta modules.

The main advantage of using the proposed SMT theory is to express consistency as the *composition* of each of the properties (φ_1 and φ_2). An encoding of the satisfiability decision problem modulo Theory 1 and Theory 2 using the Z3 [25] SMT solver is given in Appendix A together with a set of examples.

6 Safety Assurance on Over-the-air Updates using delta-RPMs

As illustrated in Figure 2, OTA updates are typically required when the mobile code needs to execute as efficiently as possible in order to make the best use of network bandwidth and the scarce supply of computational power on the remote device. This suggests that whatever technique is used to check the desired code safety properties, e.g. type safety on untrusted agents [37, 53, 59], must not penalize the performance the uploaded code over resident programs.

The *proof-carrying code* (PCC) [58] infrastructure provides a communication model where *code produces* upload mobile code into *code receivers*, which then install and execute the code in their own environment. Furthermore, the code producer sends along with the mobile code a representation of a formal proof that the code meets certain safety and correctness requirements. The main advantages of proof-carrying code are the following: 1) reduced communication between suppliers and remote devices; 2) the attached proof can be validated using a small *trusted* infrastructure; and 3) the verification pass does not impose run-time penalties for the purpose of ensuring safety.

The evidence of *code safety* takes the form of a formal proof according to a safety specification. This is conceptually different from traditional solutions that judge the safety of the mobile code by the identity of code producer. Despite the fact personal authority techniques, such as Pretty Good Privacy (PGP) [34], which is used in the RPM package management system, allow the remote device to verify not only the identity of the producer, but also that the integrity of the uploaded code is maintained, they fail to prevent trusted producers from creating an uploading erroneous code. In such cases, the OTA update can eventually break the integrity of the device's runtime system.

Alternatively, PCC allows safety assurance through programming language semantics using, for example, type safety approaches [53]. In these cases, type safety is accomplished through data abstraction in order to efficiently perform the static checking. In this way, the execution of the mobile code is not even started unless it is guaranteed to be safe.

Next, a step-by-step description of the interaction between the code producer and the code receiver is presented. In each iteration step of the protocol there is a particular software component with a well-defined functional behavior. The software components of the PCC infrastructure are depicted in Figure 7, where the right-hand side contains components of the code producer and left-hand side contains the components of the code receiver.

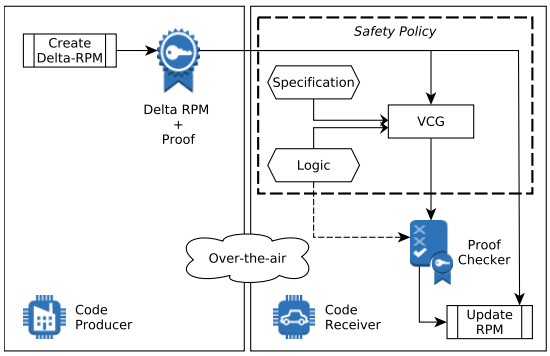


Fig. 7 Overview on PCC protocol

The component Create Delta-RPM creates an RPM that contains the program binaries plus a proof produced by the COQ theorem prover [12] that

asserts the type safety of the binaries using their table-of-symbols (cf. Section 3.3 and Section 4.5).

The proof that is carried with the code is called a *certificate*. In order to validate a delta-RPM upgrade, the code receiver establishes a *safety policy* by means of a Specification, which defines in which circumstances the existent RPM can be updated. For this purpose, the *verification condition generator* (VCG) component computes a verification-condition (VC) predicate that needs to be proven using the COQ theorem prover.

To prove that the VC predicate holds under the chosen Logic is sufficient to guarantee the safety of the OTA update. For this purpose, the code receiver relies on the Proof Checker to use the producer-provided certificate attesting the properties of interest of the table-of-symbols of the uploaded code. Finally, if an *evidence* of the safety specification is obtained, the component Update RPM performs the code patch by upgrading the RPM on the target device. In the following sections, each of the above steps are described in detail.

6.1 Defining the Safety Policy on the Remote Device

The safety policy is defined in advance according to the properties of the mobile code that can be certified. In general, a safety policy can be seen as a set of action preconditions. Therefore, it is defined in the context of a given language or data structure. When analyzing the contents of an RPM, the table of symbols is a data structure used by runtime system programs called the *linker* and the *loader* [28].

The information about the symbols table of an object file is essential for linking purposes at compile-time, when a linker is required to identify and resolve symbol references. However, when using shared libraries, the symbol table is required by the *dynamic linker*, which loads and links the shared libraries needed by an executable at run-time. When performing OTA updates, the table of symbols is at the foundation of the safety policy because it is required both at compile-time and run-time. Through a proper analysis of the table of symbols, it is possible to check if the update breaks the integrity of the run-time system before executing the dynamic loader.

The table of symbols provide valuable information to answer relevant questions that affect safety and occurrence of runtime errors [88]. The need to answer these question by means of formal checking mechanism is an essential task because substantial differences may exist between the embedded system and the runtime system of the code provider. Examples of such questions are:

1. How does the linker/loader resolve undefined symbols defined in multiple versions of the same shared library?
2. How does the linker resolve global symbols defined in multiple libraries?

The problem posed in the 1st question is that if some C/C++ function with the same name is present in more than one installed shared library, it is not possible to choose which version of the function is seen by the invoking code.

This scenario is commonly referred as the "diamond problem" (cf. Section 3.1). In order to ensure safety, it is required to check that all *undefined* symbols can be resolved before running the program because the overlapping of two shared libraries force one of them to be a hidden dependency.

The problem raised by the 2nd question is concerned with the fact that, at compile time, global symbols can be exported as either *strong* or *weak* in order to potentially be overridden at link-time by other symbols. A particular threat to safety is when multiply-defined symbols are weak symbols, the linker chooses an arbitrary one. In this case, runtime errors are difficult to track, especially if duplicate symbols definitions have different C/C++ types.

In this paper, we focus on ensuring that an OTA update is allowed if and only if all the symbols table of an executable or shared-library that are undefined, according to their ELF (Executable and Linking Format) [28] *symbol type*, can be proven to be resolved by the run-time system before execution. The symbols-table is obtained by means of the GNU `nm`[7] Linux command.

The safety policy is embodied in the PCC infrastructure using three different components. First, a mathematical logic that is required to describe preconditions under which a given update is allowed. The logic is the language used to describe and verify preconditions. The same logic is the language used to encode the verification conditions and the proofs. For the purpose of verification of properties on the symbols table, a first-order predicate logic is used.

The ELF standard is common standard for executable files and shared objects that provides a set of binary interface definitions that extend across multiple operating environments. This reduces the number of different binary interface implementations, thereby reducing the need for recoding and recompiling code. Each ELF file is made up of one ELF header, followed by file data. The table of symbols is data located in the *section header table*. This information is obtained using the `nm` command, which textual output is parsed by the verification-condition generator (VCG).

Secondly, a safety policy must specify the Linux commands that must be used to inspect meta-information contained in the delta-RPM in order to create the "new" RPM that is to be installed. In fact, the reconstruction of the "old" RPM is essential to extract the carrying proofs that might be contained in the delta-RPM. In spite of the fact that only the RPM update needs to be safe according to the safety policy, it is required to provide with a precondition for such action. This precondition establishes that all symbols are well-typed in the sense that they can be resolved by the *loader* prior to installation.

Thirdly, the safety policy contains methods to extract and inspect the contents of delta-RPMs. This task is accomplished by the VCG, which output is a VC, i.e. a predicate in the logic. The advantage of using delta-RPMs is that, instead of verifying the entire symbols table of the reconstructed RPM, only the *difference* between the symbols tables of the "old" and "new" RPMs needs to be considered. This significantly reduces the size of the VC and, consequently, the CPU cycles required to *statically* resolve undefined symbols.

[7] https://www.mankier.com/1/nm

6.2 Defining the Safety Specification Language

The language under consideration contains expressions that perform the required delta actions to create a symbols table, such as adding or removing table entries. The logic is an extension of first-order predicate logic with four predicates. The first is a typing predicate written in infix notation as $e : \tau$, where e is an expression and τ is a type.

Symbols are also typed by retaining an *abstraction* of their value, i.e. their ELF type. The container of the delta is defined as a list of symbols denoted by π. Hence, the predicate In specifies if a given ELF symbol belongs the *typing abstraction* of a table of symbols.

The DOP "apply" mechanism is encoded by the predicate $\mathsf{apply}\,(e, \pi)$, where e is a delta and π is a variant (table of symbols). This predicate denotes the evaluation of a set of delta actions to a final symbols table assuming an empty base program. The third predicate $\mathsf{resolve}\,(\mathsf{list}\ t)$ denotes that a given set of symbols which have the ELF native type U (meaning that they are undefined), can be successfully resolved at run-time. Last but not the least, the predicate $\mathsf{safe}\,(\pi)$ denotes the type-safety condition of the symbol tables.

The logic also defines a set of inference rules required to prove the verification condition. A fragment of the set of inference rules is shown at the bottom of Table 5. The reader is referred to Appendix B for a description of the remaining semantic rules. A single symbol has the type $\mathsf{ty\_symbol}\,(t)$, where t is the implicit (machine-level) EFL type of that symbol. The three following rules refer to the $(e : \tau)$ typing predicate.

Besides the typing predicates, which assigns the type $\mathsf{ty\_delta}(\dots)$ to deltas, the addition and removal of symbols to/from table assumes that these symbols are well-typed. Finally, the expression containing delta actions that build a table of symbols must agree on the corresponding $\mathsf{ty\_delta}(\dots)$ types.

Table 5 also give examples of the evaluation rule $\mathsf{apply}\,(e, \pi)$, which reduces an expression to a final table of symbols. Since the container of a delta is defined as a list of symbols, the $\mathsf{add}\,(s,\ e)$ expression evaluates to a list of symbols using functional concatenation $(\cdot \mathbin{+\!\!+} \cdot)$ of symbols. Conversely, the $\mathsf{remove}\,(s,\ e)$ expression evaluates to a list of symbols where the state s is deleted from a table of symbols using the deletion infix operator $(\cdot \setminus\!\setminus \cdot)$.

The last two rules in Table 5 specify how to classify a given table of symbols as safe. The base case for induction considers a single symbol with ELF type t, such that the predicate $\mathsf{safe}\,(t :: \mathsf{nil})$ holds. For a single symbol to be safe, it is required the predicate $\mathsf{resolved}\,(t :: \mathsf{nil})$ to hold. Using this assumption, the safety of an arbitrary table of symbols can be proven by induction of the apply evaluation predicate, by considering all valid expressions in the language.

The objective of formalizing the safety specification in this way is that the property of interest for safety is isolated by the predicate $\mathsf{resolved}$, making the proof of any verification condition an efficient process by means of a dedicated COQ tactic. In the same way, the evaluation predicate, $\mathsf{apply}\,(e, \pi)$, and the typed-expression predicates, $s : \tau$ and $e : \tau$, can be automatically proven by means of COQ tactics.

Table 5 Logic used for (delta-) type safety of an abstract table of symbols

ELF Types:	t	::=	$A \mid B \mid C \mid D \mid G \mid N \mid R \mid S \mid T \mid U \mid u \mid V \mid W \mid ?$
Symbols:	s	::=	$\mathsf{symbol}\,(v,\ t,\ n)$
Expressions:	e	::=	$\mathsf{add}\,(s,\ e) \mid \mathsf{remove}\,(s,\ e) \mid \mathsf{delta}\,(\mathsf{list}\ e)$
Symbols Table:	π	::=	$\mathsf{list}\ s$
Types:	τ	::=	$\mathsf{ty\_symbol}\,(t) \mid \mathsf{ty\_delta}\,(\mathsf{list}\ t)$
Predicates:	P	::=	$P_1 \to P_2 \mid \neg P \mid \forall_x.P_x \mid s : \tau \mid e : \tau \mid \mathsf{In}\ s\ \pi$
			$\mathsf{apply}\,(e, \pi) \mid \mathsf{resolved}\,(\mathsf{list}\ t) \mid \mathsf{safe}\,(\pi, \tau)$

Semantic rules

$\boxed{s : \tau}$

$$\frac{}{\mathsf{symbol}\,(v,\ t,\ n) : \mathsf{ty\_symbol}\,(t)}$$

$$\frac{s : \mathsf{ty\_symbol}\,(t) \qquad e : \mathsf{ty\_delta}\,(ts) \qquad \mathsf{resolved}\,(t :: ts)}{\mathsf{remove}\,(s, e) : \mathsf{ty\_delta}\,(ts \setminus\setminus t)}$$

$\boxed{e : \tau}$

$$\frac{s : \mathsf{ty\_symbol}\,(t) \qquad e : \mathsf{ty\_delta}\,(ts)}{\mathsf{add}\,(s,\ e) : \mathsf{ty\_delta}\,(t :: ts)}$$

$$\frac{e : \mathsf{ty\_delta}\,(ts_1) \qquad \mathsf{delta}\,(es) : \mathsf{ty\_delta}\,(ts_2)}{\mathsf{delta}\,(e :: es) : \mathsf{ty\_delta}\,(ts_1 +\!\!+ ts_2)}$$

$\boxed{\mathsf{apply}\,(e, \pi)}$

$$\frac{\mathsf{apply}\,(e, \pi_1 +\!\!+ s :: \pi_2)}{\mathsf{apply}\,(\mathsf{remove}\,(s, e), \pi_1 +\!\!+ \pi_2)}$$

$$\frac{\mathsf{apply}\,(e, \pi_1) \qquad \mathsf{apply}\,(\mathsf{delta}\ es, \pi_2)}{\mathsf{apply}\,(\mathsf{delta}\,(e :: es), \pi_1 +\!\!+ \pi_2)}$$

$\boxed{\mathsf{safe}\,(\pi, \tau)}$

$$\frac{s : \mathsf{ty\_symbol}\,(t) \qquad \mathsf{safe}\,(\pi, \mathsf{ty\_delta}\,(ts))}{\mathsf{safe}\,(s :: \pi, \mathsf{ty\_delta}\,(t :: ts))}$$

$$\frac{s : \mathsf{ty\_symbol}\,(t) \qquad \mathsf{safe}\,(\pi_1 +\!\!+ (t :: \pi_2), \mathsf{ty\_delta}\,(syms))}{\mathsf{safe}\,(\pi_1 +\!\!+ \pi_2, \mathsf{ty\_delta}\,(syms \setminus\setminus t)}$$

The safety specification formalized in COQ in presented in Theorem 1. The proof of the Safety proposition and the required COQ tactics are given in full detail in Appendix B.

Theorem 1: Safety Specification for Delta Application in COQ

```
Inductive resolved : (list ascii ) → Prop :=
  | DS : ∀ t, ¬ In "U"  t → resolved t.
Definition Safety :=
  ∀ (e : Expr) (v : Table) (t : list ascii),
    apply (e, v) → e : (ty_delta t) → resolved t → safe (v, ty_delta t).
```

If the assumptions given in the definition of Safety in Theorem 1 can be automatically proven, then the proof image of Safety included in the delta-RPM package, is sufficient to prove the safety of a particular (binary) delta application using one single system call to the theorem prover. This amounts to prove a verification condition in the of safe $(v, \mathsf{ty\_delta(t)})$, for some final value v with type t. In this sense, we implement a formal mechanism to implement proof-carrying code that certifies the safety of an OTA update.

The advantage of defining a logic to express safety properties is the new properties are encoded as predicates using the information contained in the table of symbols. For example, for the 2nd property of interest identified in Section 6.1, the absence of runtime errors due to multiply-defined global variables explicitly tagged as weak symbols is enforced by means of a predicate that forbids the existence of (user-defined) symbols with the ELF type "V".

Note that the described syntax is able to express the safety specification for the update of an RPM, but it does not include the actions related to the reconstruction of RPM from a delta-RPM. These actions include reading meta-information from the delta-RPM in order to have to access to the table of symbols of the binary to be checked against the safety specification. These actions are described in the following section.

6.3 Generating the Verification Condition

The fact that delta-RPMs can be manipulated by Linux commands only increases the complexity of the implementation of these two components. For example, the invocation of commands like applydeltarpm, makedeltarpm, etc., which are normally done using directly the command line, need to be included in shell scripts, which semantics are difficult to formalize (cf. Section 4.5).

Therefore, some level of abstraction is required to specify the behavior of the components on the receiver side. This is achieved by defining a workflow of *abstract actions*, as shown on Figure 8. The program accomplishing these set of abstract actions is referred to as the PCC agent. The main goal of the PCC agent is to discharge and prove a COQ verification condition (VC) for the logic presented in Table 5 as a precondition to apply the delta-RPM.

Compared to Figure 7, where the two main components are the VCG and the *proof checker*, the workflow described in Figure 8, shows that the VCG input depends on the results of previous computational steps, such as:

1. Parse the contents of the incoming delta-RPM in order to read the names of the RPMs used on the producer side to create the that delta-RPM;
2. Reconstruct the RPM by invoking the applydeltarpm Linux command;
3. Extract the contents of the reconstructed RPM by means of the Linux command rpm2cpio[8]. This includes both the carrying proofs, which are binary images of the produced COQ proofs and the patched binary files.

[8] https://www.mankier.com/8/rpm2cpio

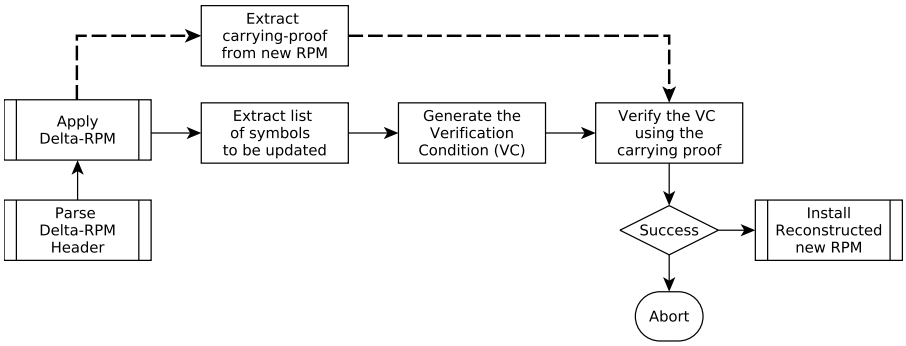


Fig. 8 Detailed overview of the PCC-agent workflow on the receive side

After building a representation for the textual output the nm Linux command, it is necessary to take into account the computational cost of computing and analyzing tables of symbols. For instance, consider that the "new" RPM includes a shared library (DSO) that needs to be verified against the software already installed on the remote device.

At run-time, the execution of the *dynamic linker* is required to load all the shared libraries that might be needed by the object files contained in the re-constructed RPM. In most implementations, this process does not scale linearly as it is asymptotically at least $\mathcal{O}(R + nr \log s)$, where R is the number relative relocations, r is the number of named relocations, n is the number of participating DSOs (plus the executable), and s is the number of symbols [28].

Therefore, the static checking of the symbols table of the executable is asymptotically $\mathcal{O}(ns^2)$ because it is necessary to *compare* the symbols of each dependency with the symbols of the executable and verify if they can be relocatable at run-time. This is equivalent to three nested `for`-loops, where the outer `for`-loop iterates $\mathcal{O}(s)$ times, the inner `for`-loops iterate $\mathcal{O}(n)$ times for each dependency, and each dependency iterates again $\mathcal{O}(s)$ times.

Since the static symbol resolution is a very time-consuming task, the use of abstractions for checking the type safety of symbols tables is crucial. Besides the type abstraction presented in Section 6.2, another design solution to efficiently check the safety policy is to consider only the table of symbols that results from the lexical comparison between the "old" and "new" RPMs. Then, the COQ tactic that is used to prove the predicate resolved takes as input a table of symbols that is, in general, small in terms of size. This technique contributes for the minimization of the proof effort on the receiver side.

The number of symbol comparisons performed by the VCG can also be reduced using the ELF concept of *symbol visibility* [28]. Symbol visibility indicates if the symbol is visible outside the file being built. In one hand, the opportunity to export symbols as needed does not only benefit library security, but also reduces dynamic linking time. On the other hand, a small number of exported symbols has a negative impact on modularity and chances for reuse of shared libraries.

6.4 Proving the Verification Condition

The following step is to verify the VC using the proof-carrying images. In this phase, an instance of COQ is used to check the validity of the delta-update using the certificate reconstructed from the delta-RPM. As illustrated on the right-hand side of Figure 8, the result of the proof checker is either success or failure. In the first case, if the VC is provable within the logic, then the contents of the delta-RPM do not violate the safety policy and the precondition for updating the reconstructed RPM is guaranteed. On the second case, the binary patch cannot be proven safe and the operation is simply aborted.

Next, we present in Table 6 the high-level syntax required in to express the behavior of the components of the PCC agent. More precisely, for each of the component participating in the PCC workflow described in Figure 8, there is a syntax phrase abstracting the corresponding low-level operations.

Table 6 Syntax of the PCC agent on the reciver side

Variables	v	::=	x
Internals	i	::=	y
AbstActions	a	::=	ParseDeltaRPM x | ApplyDeltaRPM x | GetSymTable x | GenerateVC x | ProveVC x | updateRPM x
ConcActions	f	::=	Function x $(y_1 \rightarrow y_2)$
BoolExprs	b	::=	Verify a
Commands	c	::=	abort | $x := a$ | c_1 ; c_2 | if b then c_1 else c_2

Semantic rules

$\boxed{\psi, \sigma \vdash v \Downarrow^v i}$ $\qquad$ $\boxed{\psi, \sigma \vdash a \Downarrow^a i}$

$$\frac{}{\psi, \sigma \vdash x \Downarrow^v \sigma(x)} \qquad \frac{\psi, \sigma \vdash \psi(e) = \text{Function } x\ g \qquad \psi, \sigma \vdash x \Downarrow^v y}{\psi, \sigma \vdash e \Downarrow^a (g\ y)}$$

$\boxed{\psi, \sigma \vdash c \Downarrow^c \sigma'}$

$$\frac{\psi, \sigma \vdash e = \Downarrow^s y}{\psi, \sigma \vdash \text{Assign } x\ e \Downarrow^c [x \mapsto y]\sigma} \qquad \frac{\psi, \sigma \vdash c_0 = \Downarrow^c \sigma' \qquad \psi, \sigma' \vdash c_1 = \Downarrow^c \sigma''}{\psi, \sigma \vdash c_0\ ;\ c_1 \Downarrow^c \sigma''}$$

$$\frac{\psi, \sigma \vdash v = \Downarrow^b \mathit{true} \qquad \psi, \sigma \vdash c_1 \Downarrow^c \sigma'}{\psi, \sigma \vdash \text{if } b \text{ then } c_1 \text{ else } c_2 \Downarrow^c \sigma'} \qquad \frac{\psi, \sigma \vdash v = \Downarrow^b \mathit{false} \qquad \psi, \sigma \vdash c_2 \Downarrow^c \sigma'}{\psi, \sigma \vdash \text{if } b \text{ then } c_1 \text{ else } c_2 \Downarrow^c \sigma'}$$

The abstract actions are defined in the syntax category $a \in \mathit{AbstActions}$, where the constructs ParseDeltaRPM, ApplyDeltaRPM, GetSymTable, GenerateVC, ProveVC and updateRPM represent functions on values which are internal to the implementation and defined in the syntax category $f \in \mathit{ConcActions}$. The mapping between an abstract action and a concrete action is given by the function $\Psi :: (\mathit{AbstActions} \rightarrow \mathit{ConcActions})$.

Both abstract and concrete actions depend on variables $v \in \mathit{Variables}$, which are associated with low-level implementation values denoted by $i \in \mathit{Internals}$ inside a *store*. The set of all stores is denoted Σ. Because internal values are specific to the Haskell implementation of the PCC agent, they are simply denoted by $y_1, y_2, \ldots$. Examples of internal values are datatypes for the file-system, lists, exit codes, etc. Like other simple imperative languages, PCC agent programs are constructed using elements $c \in \mathit{Commands}$.

The big-step operational semantics for the agent language is given at the bottom of Table 6. The meaning of an agent program is expressed by semantic rules that specify the entire transition from an initial *configuration*, i.e. a pair $(\Psi \times \Sigma)$, to a final (internal) value or *store*. Intuitively, any configuration (ψ, σ) represents the dynamic instances of the two maps during a computation, containing the current value for variables.

There are three types of semantic rules, denoted by three different relational operators: $\Downarrow^v$, $\Downarrow^a$ and $\Downarrow^c$. For abstract actions, the final value is an value internal to the low-level implementation; for Boolean expressions, the final value is $t \in \{\mathit{true}, \mathit{false}\}$; and for commands, the final values is a store. We also assume that $[x \mapsto y]\sigma$ denotes the update of σ on the variable x with value y.

The Haskell interpreter for the language of the PCC Agent presented in Table 6 is given in Appendix C and available from https://github.com/esmifro/CertifiedDOP.

7 Data Analysis and Evaluation

In this section, we present and analyze experimental results that insight on the trade-off between fast download times of OTA updates and efficient checking mechanisms when features are dynamically selected upon a reconfiguration requests (cf. Figure 2). The objectives of this empirical evaluation are: 1) to establish a rationale, using a small pilot project, in order to conclude if the hypothesis posed in Section 1, i.e. the existence of a trade-off, can be extrapolated to more complex examples by means of statistical significance; 2) to evaluate the feasibility of the modeling/development/deployment methodology using the execution time of the PCC agent as a process-oriented metric.

The experiments are done in the context of the Hyvar European project, which considers the use of the Autonomous Telematics Box ATB2 (developed by Magneti Marelli) as the Electronic Control Unit (ECU). This ECU integrates a telephone module for inter-connection with cellular communication networks, a multi-constellation satellite localization module and a remote update mechanism suitable for the `ECall/E112 Regulation` case study. The production scenario being tested is OTA update that needs to be deployed upon a reconfiguration request, depending on Global Navigation Satellite System (GNSS) that is currently locating the vehicle. Using DOP-SPL terminology, the deployed code depends on the selected features (cf. Section 4.2).

7.1 Experimental Overview

The results presented in this section correspond to the executable variants that are generated using the source code specifically developed for the Hyvar project. Experimental data is measured with respect to factors such as the size of the OTA update, modeling/developing methodology, available hardware and support software. By setting the goal of the experiment in the process of finding the best trade-off between the size of the OTA update and the execution time of the PCC agent, we are able to identify questions of interest that permit quantitative analysis.

Questions of interest define the objective measurements to be performed, the way experimental data is presented and the statistical framework that is used to validate the use of the selected metric. The addressed questions are:

- *What are benefits of modeling data/control coupling at binary level?*
- *How much bandwidth can be saved by using the delta-RPM technology?*
- *Is the execution of the PCC agent fast enough to meet the computational resources of embedded system?*
- *Can the DOP-SPL increase the flexibility and adaptability of a self-check framework supporting OTA updates?*

Answers to these questions are obtained by applying an iterative heuristic algorithm that carries out the empirical validation process. The steps of such algorithm are briefly described by the following:

1. *Available compilation schemes are analyzed according to the selected features (cf. Figure 3) and the two variants involved in the OTA update;*
2. *These schemes are then checked against the distributed embedded system requirements (e.g. Worst-Case Execution Time of the PCC agent);*
3. *The hypothesis of performing the OTA update is confirmed or refuted depending whether there is a trade-off that satisfies the previous requirements;*
4. *If the hypothesis is refuted, start a new iteration at the level of build-system modeling (cf. Figure 2) to increase or decrease data/control coupling;*
5. *Repeat the previous steps until the best trade-off is found.*

The above algorithm follows from the round-trip methodology presented in Section 4.2, which considers a SPL with multiple artifacts bases. The use of multiple artifact bases introduces binary re-modularization into the SPL development process. In the following, we refer the two different variants by the names “GPS” and “GLONASS” introduced in Section 4.2.

The screening of experimental data is done using histograms and the statistical framework ensures the representativeness of the sampling by using a (univariate) linear regression model, where the *goodness-of-fit* statistical measure, r^2, is required to be equal to 1. This means the regression line must perfectly fit the data and that errors are normally distributed [5, 85].

The series of experiments consider the available set software by-products, without taking into consideration the correlation between the delta-oriented baseline and the derived products. Scalability is addressed by demonstrating

that experimental results are correctly interpreted, so that experiments can be replicated on more complex case studies.

7.2 Experimental Design

The purpose of using execution time as a process-oriented metric instead of code-level metrics such as lines of code or Cyclomatic complexity [55, 84], is to give the metric a prescriptive nature. In particular, the execution time of the PCC agent provides an insight, on a test-first basis, on the trade-offs that are intrinsic to the modeling/developing/deployment process.

This testing strategy aid software engineers to evaluate and validate the product using a systematic methodology based on empirical evidence. Depending on the distributed embedded system where the product is deployed, the execution time metric is a factor that decides if the posed hypothesis is either refutable or confirmed.

In the context of the Hyvar European project, the ECU that managed the ECall/E112 services was based on an ARM9 core with 256KB of RAM and 2MB of flash memory. With the current technological developments of the control units, the ECUs that will manage the OTA updates will be increasingly performing: e.g. currently we can talk about 4xARM-R52 cores in lockstep (8 cores total) and up to 4GB of memory.

To make the experiment reproducible without the need for specific hardware and more flexible for future works, it was decided to use a virtual machine for the empirical evaluations that will follow. The experimental results were obtained using the Criterion benchmarking library[9] on a Linux Fedora virtual machine, configured to use a maximum of 4GB of memory, running on a 4x Intel(R) Core(TM) i7 CPU@2.70Ghz, 8GB memory, hardware machine. The virtual machine is available at http://cdop.di.unito.it/.

7.3 Empirical Evaluation

As previously explained in Section 4.4 in Table 3, there are three possibilities to compile a DOP executable variant using the GNU Build System. For the ECall/E112 Regulation case study, software packages can be compiled either as: 1) a *single executable* file; 2) an *executable distributed with an internal shared library*; 3) as an *executable that uses a pre-installed shared library*.

Next, we describe a set of experiments that focus on the measurement of the size of binary distributions, i.e. RPMs and delta-RPMs, and the static checking time required to ensure the safety policy described in Section 6.2. These two experiments support empirical evaluation of the cost of performing formal verification during OTA updates when the logical structure of binary code is subject to delta-modeling. Although the decision of accepting or rejecting a particular product is ultimately dictated by the operational requirements

[9] http://www.serpentine.com/criterion/

of the embedded system, empirical analysis allows to anticipate which design choices are the most adequate.

7.3.1 Evaluating the Size of OTA Update

The first set of measurements corresponds to the size of RPMs before the creation of delta-RPMs. The histogram presented in Figure 9 discriminates six possibilities to customize `ECall/E112` programs with Automake models: the "GPS" variant compiled as a single executable, an executable distributed with an internal DSO or as an executable using a pre-installed DSO and the same for the "GLONASS" variant. The values displayed in **blue** background represent the size in KBytes of the executables, the values displayed in **red** background represent the size of DSOs and the values displayed in **yellow** background represent the size of the correspondent RPM.

The analysis of Figure 9 shows that the size of the "GPS" variant is smaller than that of the "GLONASS" variant. For the cases where no DSOs are used, the size of the RPMs are approximately equal (GPS – 60KB / GLONASS – 65KB). As expected, when the executable is distributed with an internal shared library, the size of the RPM is significantly bigger because shared libraries include a relocation entry on the DSO header (cf. Section 3.3).

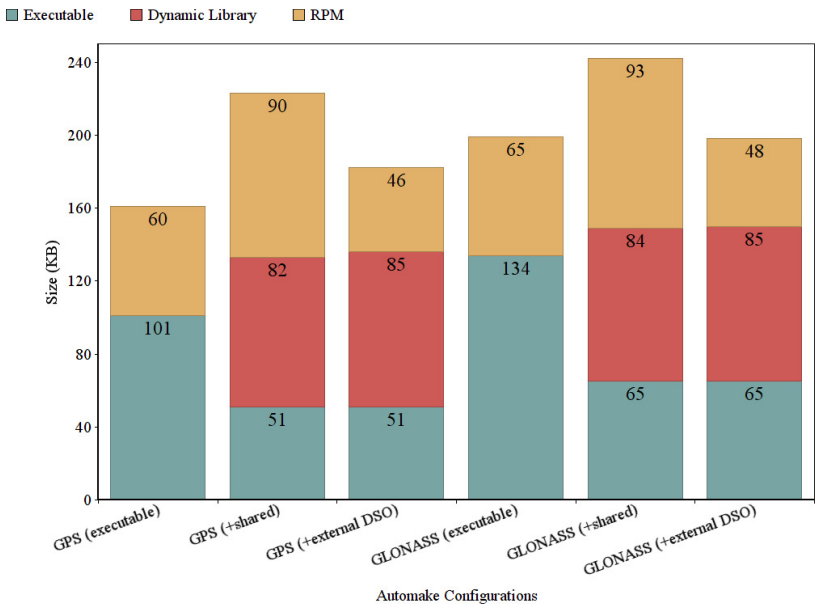


Fig. 9 Size of Executables, Shared Libraries and RPMs

For example, compared with the "GPS" single executable, the reduction of the "GPS" variant using DSOs is of 50KB, but the size of the entire program (containing the DSO) increases to 82KB + 51KB = 133KB (a greater value

than the initial 101KB). For this reason, DSOs incur overhead and are effective only when linked with at least a few programs.

As a side effect, the size of the RPMs also increases greatly (90KB / 93KB). However, the size of RPMs can be reduced when the DSOs are compiled in a single (external) pre-installed DSO (46KB / 48KB). This is the best trade-off is terms of RPM size and the cost of installing a common DSO is negligible (85KB − 82KB = 3KB / 85KB − 84KB = 1KB).

After compiling the several RPMs, it is possible to evaluate the efficiency of the makedeltarpm command in terms of the delta-RPM size (KB) and the time required to verify the safety properties as a pre-condition to install the delta-RPM. The results for the sizes of delta-RPMs are displayed in Figure 10 for all the possible compilation schemes (nine) when applying delta-RPMS. Table 7 describes these schemes in terms of the arguments that are passed to makedeltarpm, i.e. the "old" and the "new" RPM. Please note that, for each of the compilation schemes, the output depends on the order of the arguments.

Table 7 Description of all possible delta-RPM compilation schemes using makedeltarpm

delta-RPM Compilation Scheme	Combinations of "new/old" RPMs (and vice-versa) used by the makedeltarpm Linux command
1	GPS (executable) / GLONASS (executable)
2	GPS (executable) / GLONASS (+shared)
3	GPS (executable) / GLONASS (+external DSO)
4	GLONASS (executable) / GPS (+shared)
5	GLONASS (executable) / GPS (+external DSO)
6	GLONASS (+shared) / GPS (+shared)
7	GLONASS (+shared) / GPS (+external DSO)
8	GLONASS (+external DSO) / GPS (+shared)
9	GLONASS (+external DSO) / GPS (+external DSO)

Scheme 1 defines two delta-RPMs that can be built using single executables for the "GPS" and "GLONASS" variants. The first is obtained when "GPS" is the "new" RPM and the second is obtained when "GPS" is the "old" RPM. The size of these two delta-RPMs is different because the symbols that need to be patched in the "old" RPM depends on the selected executable variant.

For example, when the "new" RPM is built when the "GPS" executable variant is selected, there are no symbols to be added (only to be removed). Conversely, when "GLONASS" is selected, the executable contains additional symbols exclusive to the "GLONASS" variant. In Scheme 1, all additional symbols have the ELF types T or D (Section 5) and they belong to the *text-code* and *initialized* ELF sections, respectively.

As Fig. 10 illustrates, the order of the arguments of makedeltarpm is not symmetrical. This fact is more visible when DSOs are used. In Scheme 2, for

example, when "GPS" is the "new" variant, a restricted set of symbols are added to the ELF *text-code* (T), *uninitialized* (B) and *initialized* (D) sections. However, when the "GLONASS" (using an internal DSO) is the new variant, several undefined symbols (U) need to be patched because position-dependent memory addresses need to be removed. For this reason, the "GLONASS/GPS" delta-RPM is significantly bigger than the "GPS/GLONASS" version.

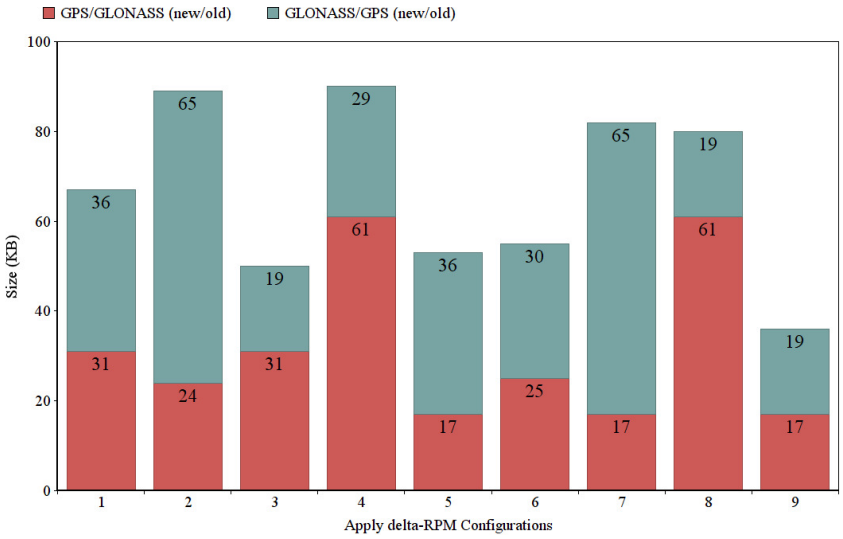


Fig. 10 Comparison Between the size (KB) of delta-RPMS

The choice between the inclusion the DSO inside the binary package of the executable or the installation of the DSO as a separate package also has a significant impact in the number of symbols that need to be patched. For example, in Scheme 7, is visible a great asymmetry between the size of the delta-RPMs. The reason for this fact is that when the DSO is pre-installed, the size of the GPS executable (+ external DSO) is very small (51KB). Consequently, the corresponding delta-RPM shown in Figure 10 is also small (17KB).

Conversely, if the reconstructed RPM includes the GLONASS executable (+ shared), it requires patches for undefined and text-code symbols because it contains both the internal DSO and the executable. Consequently, the size of the delta-RPM is necessarily bigger (65KB). In summary, the more efficient compilation scheme in terms of the delta-RPMs sizes are obtained when reusing pre-installed DSOs. This is the case of Scheme 9, where the size of delta-RPMs sizes are approximately the same (17KB / 19KB).

7.3.2 Evaluating the Execution Time of the PCC Agent

Next, for each of the delta-RPM compilation schemes presented in Table 7, we present the corresponding execution time of the PCC agent. The execution

time is a statistical estimation that is computed via Least-Ordinary Squares (OLS) [5, 85] by running the Criterion profiler on the PCC agent a certain number of times. The conditions for statistical significance are evaluated through the application of the heuristic algorithm that, iteratively, run the experiments until the criteria for using OLS regression is satisfied. Next, we briefly describe the observations made during the application of such algorithm.

The first observation was that the goodness-of-fit coefficient, r^2, increases towards 1 when we increase the number of runs of the PCC agent, hence increasing the sample population. The second observation is that a relatively small number of samples, 10 runs in this case, is found to be sufficient to assure the soundness of the OLS predictor because the condition $r^2 = 1$ applies to all the compilation schemes described in Table 7.

The main reasons for this fact are: 1) the elimination of correlation between the initial DOP-SPL baseline and the derived variants, and the consideration of individual variants independently; and 2) the avoidance of measurement bias by using a virtual machine without a running graphical interface. The third observation is a substantive measure of how much the "outliers" affect the regression model, which is 19% in all experiments that consider 10 samples.

Comparative analysis between the available compilation schemes takes into consideration the number of symbols contained in all object files inside each delta-RPM. The corresponding histogram is presented in Figure 11, where the number of updated symbols from the "GPS" variant to the "GLONASS" variant is displayed in **green** background, and the number of updated symbols from the "GLONASS" variant to the "GPS" variant is displayed in **yellow** background. Likewise, the histogram showing the estimation for the execution-time of the PCC agent is presented in Figure 12.

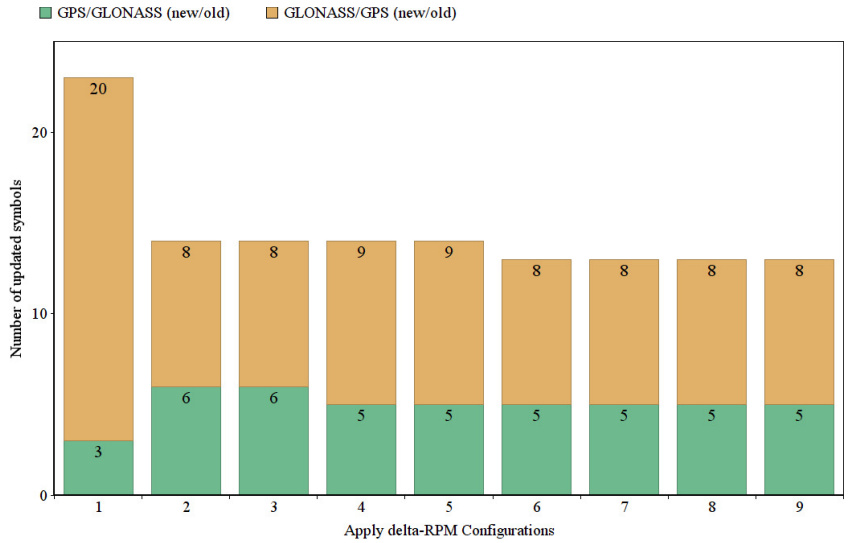


Fig. 11 Number of symbols for "new" and "old" RPMs per compilation scheme

The first conclusion is that execution time strongly depends on the number of symbols that need to be checked against the safety specification. Secondly, we conclude that the type of symbol has influence not only on the performance of the proof-checker, but also on the verification-condition generator (VCG) (see Figure 7). For example, in Scheme 1, where the variants are compiled as single executable files (absence of *undefined* symbols), there are 3 patched symbols on the "GLONASS" RPM when the "GPS" variant is being deployed.

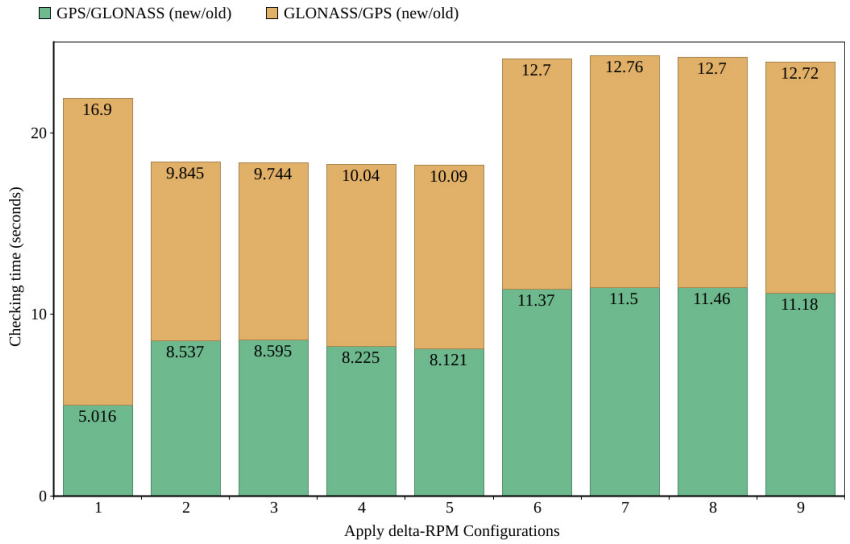


Fig. 12 Checking time for "new" and "old" RPMs per compilation scheme

Conversely, there are 20 patched symbols when the the "GLONASS" variant is deployed. The histogram of Fig. 12 shows that the update of the "GLONASS" RPM over of the "GPS" RPM takes 16.9 seconds, whereas when the update over the "GLONASS" RPM only takes 5.016 seconds. Hence, we conclude that the choice to compile variants as single executables files is potentially expensive from the checking-time point of view, even though the size of the delta-RPMs are relatively small (31KB / 36KB).

In this particular case, the proof effort for checking the VC generated by the PCC agent can be non-negligible when the number of patched symbols is high. Therefore, the scenario that uses only executables refutes the hypothesis of finding the best trade-off between the size of delta-RPMs and the PCC agent checking time because feature selection give rise to very disparate execution times. This fact makes impossible to extrapolate results to other scenarios and makes scalability difficult to assess.

Whenever DSOs are included in the binary distributions, the number of patched symbols in the executable decreases significantly. Nevertheless, since the table of symbols of the "GLONASS" variant is bigger than that of the "GPS" variant, the number of patched symbols is always higher when perform-

ing an update of the "GLONASS" RPM over of the "GPS" RPM. However, the variation of the PCC agent execution time is not caused only by the number of patched symbols, but also by the ELF type of each symbol. In particular, the former has an impact on the execution of the COQ tactics used by the proof-checker and the latter has an impact on the VCG execution time.

As previously explained in Section 6.3, only the symbols that are *undefined*, i.e. with the ELF type U, need to be statically resolved in order to guarantee run-time integrity using a $\mathcal{O}(ns^2)$ algorithm, where n is the number of participating DSOs and s is the number of symbols. Independently of either using a DSO distributed along with the executable or using an external DSO pre-installed in the system, the increase of the execution time of the VCG is caused by the number of comparisons that need to be performed between undefined symbols contained in the executable and all its dependencies.

With respect to the number of dependencies of the executable module, our observation is that the increase of the total execution time of the PCC agent is non-negligible, even if the number of patched symbols is the same when compared to the other compilation schemes. For example, when the delta-RPM compilation scheme defines that both "new" and "old" RPMs include DSOs, like Table 7 shows for Schemes 6, 7, 8 and 9, the additional computational cost is above 2 seconds. Nevertheless, the execution time is less dependent on the selected features.

7.3.3 Empirical Validation of the Best Compilation Scheme

Experiments demonstrate the absence of an *all-purpose* solution when compiling software for the purpose of performing efficient OTA updates by means of binary patches. Factors that contribute for this facts are bandwidth constraints and computational constrains on the distributed embedded system. In one hand, to design embedded code as a single executable file is simple, but very inefficient in terms of bandwidth usage due to the amount of code that needs to be transferred to apply the patch. This fact can be extrapolated to other environments, suggesting that a monolithic approach to the logical organization of binary code is not appropriate.

On the other hand, the compilation of shared libraries installed separately incur on non-negligible computational overhead when performing the integrity check against the safety policy. The first conclusion is that increasing modularity is not necessarily the best direction when developing a DOP-SPL supporting OTA updates. This fact alone can be sufficient to make the OTA update impossible to perform when the embedded system has, for example, real-time constrains that need to be preserved. Therefore, the negative effect excessive dynamic linkage can also be extrapolated to other environments.

This heuristic analysis suggests the existing of trade-offs between fast downloads of OTA updates and efficient static verification on the remote device. The rationale for validating the *best trade-off* hypothesis is the following: given a certain available bandwidth and a specification of the resource capabilities of the remote device, if a given binary patch can be formally verified

by the PCC agent by empirical evidence, the hypothesis of performing the OTA update is confirmed. When a given binary fails the test, the hypothesis of performing the OTA update is, therefore, refuted.

Considering the `ECall/E112` case study, we conclude that Scheme 9 is the optimal compilation scheme (trade-off) because: 1) the of size of the delta-RPM is the smaller among all schemes, i.e. 17KB or 19KB (cf. Figure 10), for the "GPS" or the "GLONASS" update, respectively; 2) the estimation for execution time of the PCC agent is 11.18 or 12.72 seconds (cf. Figure 12), for the "GPS" or the "GLONASS" update, respectively. However, as previously mentioned, this fact should not be extrapolated to other environments because we are considering a relatively small number of patched symbols (cf. Figure 11).

The scalability of this pilot use-case scenario requires precise knowledge about the available computational resources on the remote device. As previously mentioned in Section 7.1, upper-bound estimates on the execution time of the PCC agent (e.g. WCET) can be an essential aid to the decision process of confirming or refuting the posed hypothesis. Despite this fact, the experimental evaluation given in Section 7.3.1 suggest that scalability can be achieved using round-trip engineering, making use of build-system models to assist variant re-modularization.

7.4 Threats to validity

We will use the terminology of [20]. Scalability is an *external* validity inference and the empirical evaluation described in this paper lacks deductible conclusions. However, the empirical assessment strongly relies on *statistical conclusion* validity, which never undermines *internal* validity. In fact, and despite the use of regression and a virtual machine to simulate the computational resources of the embedded system, the causal relationship between the number of symbols inside binaries and the execution time of the PCC agent is demonstrated by empirical evidence.

The integration of WCET estimation [67] is proposed to maximize *construct* validity. The use of formal techniques such as Abstract Interpretation in order to determine if predictions are supported across contexts, for example, if the theoretical hypothesis can be proven with respect to real-time constraints, would improve the assessment of threats to external validity.

Outside the scope of this study are threats to external validity caused by the non-inclusion of background variables, e.g. available downlink bandwidth. Since the interaction of these variables might modify the observed effects, external validity is very difficult to obtain due to the impossibility to anticipate all the background variables implied in OTA updates.

8 Conclusions

This paper presents a SPL that integrates MDD in order to provide flexible OTA updates for a simplified ECall/E122 Regulation scenario. The DOP

paradigm is used as the variability realization mechanism. The DOP product line uses delta actions at the three levels of state-diagram modeling, customized compilation of C/C++ code and deployment of delta binaries on remote devices.

A formal verification mechanism ensures the correctness of the realization artifacts, i.e. state-diagram and build-system models, according to the semantics of DOP. The safety property of interest when performing self-check OTA updates on the distributed embedded system is the run-time integrity on the remote device. For this purpose, a particular PCC framework was developed.

Empirical validation shows that the physical realization of executable variants has a deep impact on the size and performance of OTA updates. Empirical evidence demonstrates that the generation of executable variants does not scale if the structure of the binary code is not taken in consideration. We provide a solution for this problem through an heuristic algorithm that facilitates empirical validation in each iteration of the DOP-SPL round-trip development. Such algorithm provides experimentally-acquired evidence that the size of binary patches can be significantly reduced, hence capable of improving download efficiency, but also reveals that the cost of performing self-check OTA updates on remote devices may preclude the entire operation.

The observation of a trade-off between the size of binary patches and the computational effort required to guarantee the integrity of the runtime system after deployment requires the evaluation of the software system and the simulation of the environment in which the OTA update is to be performed before taking action. The hypothesis to find the best trade-off between the conflicting characteristics of the software system is confirmed or refuted by applying the iterative heuristic algorithm on individual products. A statistical framework supports this process by means of a process-oriented metric.

The evaluation of the pilot use-case shows that the scalability of the software system can only be achieved by performing incremental OTA updates, where the trend is to minimize program size and data/control coupling in each of such individual binary updates. The hypothesis of using the best available compilation scheme in the DOP-SPL is order to certify OTA updates is realistic only when the cost of formal verification is empirically validated, in each iteration separately, on a test-first basis, and considering the computational resources of the remote device. The design choice to apply DOP techniques ubiquitously in the distributed embedded system provides additional flexibility to define the appropriate modularity of the software system by configuration.

The pilot use-case described in this paper is an idealized version of a demonstrator developed in the HyVar project. The approach to MDD makes strong simplifying assumptions by not considering tool-support for state diagrams. Hence, the processes of code-generation and analysis of C/C++ source code are not included in the proposed methodology. Without imposing limitation on the proposed DOP-SPL methodology, the analysis of different kinds of design models use the semantics of abstract machines and the cost analysis of self-check OTA updates is performed directly on binary files.

The integration of the proposed approach in a full-fledged tool chain, like the one developed by the HyVar project [22, 52], is a major project that re-

mains as future work. This integration should incorporate tool-support for data and control coupling analysis at the level of C source code [36] (as done, e.g., in the context of RTCA DO-178 projects [76]), the use of code-level metrics [55, 84] and static analysis of embedded code (e.g. Worst-Case Execution Time (WCET) analysis in distributed embedded systems [67]).

In addition to the process-oriented metric used presented in this paper, i.e. the total execution time of the PCC agent on the remote device, the referred analyses can assist the heuristic algorithm in the process of finding the optimal trade-off in the presence of more complex use-cases and more detailed specifications about the available computational resources on the remote device.

Acknowledgements We thank the anonymous SoSyM referees for comments and suggestions for improving the paper.

Appendices

Appendix A Consistency Check Using the Z3 SMT Solver

This appendix shows the Z3 [25] SMT solver scripts that check the satisfiability and validity of the consistency specification established between the State-Diagram and Build-System DOP product lines. The consistency Σ-theory [57] is formalized using constraints between pairs of deltas that share the same index in the application order. Further, the Configuration Knowledge (CK) of the DOP-SPL is consistent only if there is an activation rule for every delta index. The activation rules are defined in conjunctive normal form (CNF) and implicitly determine the application order.

In order to demonstrate the use of Z3 encoding of the Σ-theory presented in Section 5.2 considers two delta indexes and, correspondingly, two activation rules. Next, we present the CKs of the two DOP-SPLs under consideration. The delta sequences under consideration are $\langle \delta_1 \,; \delta_1 \rangle$ and $\langle \dot{\delta}_1 \,; \dot{\delta}_1 \rangle$, respectively.

1. Configuration Knowledge of the State Diagram DOP-SPL:

F_1	δ_1
$F_1 \wedge F_2$	δ_2

2. Configuration Knowledge of the Filenames DOP-SPL:

F_1	$\dot{\delta}_1$
$F_1 \wedge F_2$	$\dot{\delta}_2$

The Z3 definitions are given in Listing 1. The encoding of theory specifying the cross-consistency between the State-Diagram DOP-SPL and the Build-System DOP-SPL is presented in Listing 2 and the encoding of the DOP semantics with respect to cross-consistency between SPLs in given in Listing 2. The source script is available at https://github.com/esmifro/CertifiedDOP/tree/master/z3.

Examples of valid and unsatisfiable instances of product lines are described in Listing 4, Listing 5 and Listing 6.

Listing 1: Z3 Script – Definitions

```
 ;; defined predictates
(declare-datatypes () ((Index I1 I2)))
(declare-datatypes () ((Formula F1 F2)))

(declare-const name String)

(declare-fun delta_state (Index) Bool)
(declare-fun delta_file (Index) Bool)
(declare-fun state (String Index) Bool)
(declare-fun filename (String Index) Bool)
(declare-fun ck (Formula Index) Bool)
```

Listing 2: Z3 Script – Theory 1

```
(define-fun cross_c ((i1 Index) (i2 Index)) Bool
    (and (delta_state i1) (delta_file i2)))

(define-fun bijection ((x String) (i1 Index) (i2 Index)) Bool
    (=> (= i1 i2) (not (xor (state x i1) (filename x i2)))))

(define-fun prop1 () Bool
  (forall ((x String) (i1 Index) (i2 Index))
   (= (cross_c i1 i2) (bijection x i1 i2))))
```

Listing 3: Z3 Script – Theory 2

```
(define-fun prop2 () Bool
  (forall ((i Index)) (exists ((f Formula)) (= (ck f i) (cross_c i i)))))

(assert (= true (and prop1 prop2)))
```

The following encoding SPL shows that both deltas have the same position in the order and both declare the functions named `name_1` and `name_2` and the corresponding filenames (which must have the same name). Further, the predicate `ck` is satisfiable for every formula and delta index. This instance corresponds to a single virtual product (cf. Section 5) that is satisfiable.

The constraints `(assert (=> (ck formula_1 index_2) false))` and `(assert (=> (ck formula_1 index_2) false))` are necessary to properly encode CK (1) and CK (2).

Listing 4: Z3 Script – Consistent Configuration Knowledge

```
(declare-const index_1 Index)
(declare-const index_2 Index)
(assert (distinct index_1 index_2))

(declare-const formula_1 Formula)
(declare-const formula_2 Formula)
(assert (distinct formula_1 formula_2))

(declare-const feature_1 Bool)
(declare-const feature_2 Bool)

 ;; both features are selected
(assert (= true feature_1))
(assert (= true feature_2))

 ;; encoding of the 1st entry in the CK
(assert (=> (ck formula_1 index_1) feature_1))
(assert (=> (ck formula_1 index_2) false))

 ;; encoding of the 2nd entry in the CK
(assert (=> (ck formula_2 index_1) false))
(assert (=> (ck formula_2 index_2) (and feature_1 feature_2)))

 ;; define cross-consistent functions and filenames
(assert (=(state "name_1" index_1) true))
(assert (=(filename "name_1" index_1) true))

(assert (=(state "name_2" index_2) true))
(assert (=(filename "name_2" index_2) true))

;;(assert consistency)
(echo "-is it satisfiable?")
(check-sat)
```

Now consider Configuration Knowledge that is inconsistent, where F_2 is disabled by writing `(assert (= false feature_2))`. In this case, the activation rule for `index_2` is not satisfiable. Therefore, the answer from the SMT solver is going to be "unsat".

Listing 5: Z3 Script – *In*consistent Configuration Knowledge

```
(declare-const index_1 Index)
(declare-const index_2 Index)
(assert (distinct index_1 index_2))

(declare-const formula_1 Formula)
(declare-const formula_2 Formula)
(assert (distinct formula_1 formula_2))
```

```
(declare-const feature_1 Bool)
(declare-const feature_2 Bool)

(assert (= true feature_1))
 ;; disable F2
(assert (= false feature_2))

(assert (=> (ck formula_1 index_1) feature_1))
(assert (=> (ck formula_1 index_2) false))

(assert (=> (ck formula_2 index_1) false))
(assert (=> (ck formula_2 index_2) (and feature_1 feature_2)))

(assert (=(state "name_1" index_1) true))
(assert (=(filename "name_1" index_1) true))

(assert (=(state "name_2" index_2) true))
(assert (=(filename "name_2" index_2) true))

;;(assert consistency)
(echo "-is it satisfiable?")
(check-sat)
```

Now consider the case of inconsistency between deltas, δ_1 and $\dot{\delta}_1$, written `(assert (=(state "name_1" index_1) false))`. The answer from the SMT solver is again going to be "unsat".

Listing 6: Z3 Script – *In*consistency between "state" and "filename" deltas at `index_1`

```
(declare-const index_1 Index)
(declare-const index_2 Index)
(assert (distinct index_1 index_2))

(declare-const formula_1 Formula)
(declare-const formula_2 Formula)
(assert (distinct formula_1 formula_2))

(declare-const feature_1 Bool)
(declare-const feature_2 Bool)

(assert (= true feature_1))
(assert (= true feature_2))

(assert (=> (ck formula_1 index_1) feature_1))
(assert (=> (ck formula_1 index_2) false))

(assert (=> (ck formula_2 index_1) false))
(assert (=> (ck formula_2 index_2) (and feature_1 feature_2)))
```

```
 ;; introduce a wrong state name
(assert (=(state "name_1" index_1) false))
(assert (=(filename "name_1" index_1) true))

(assert (=(state "name_2" index_2) true))
(assert (=(filename "name_2" index_2) true))

;;(assert consistency)
(echo "-is it satisfiable?")
(check-sat)
```

Appendix B The COQ Proof System

This appendix shows the COQ definitions required to define the safety specification presented in Section 6. Listing 1 shows the inductive definitions for symbols, where the ELF-type is an "ascii", table of symbols and delta actions. The dynamic semantics of the delta-oriented "apply" function is encoded by predicate `Apply` in Listing 2. The definitions for types of symbols and table of symbols is shown in Listing 3.

Listing 1: Delta Definition

```
Inductive Symbol :=
  | Sym : ascii → string → Symbol.
Inductive Table :=
  | List : list Symbol → Table.
Inductive Expr :=
  | Add_operation : Symbol → Expr → Expr
  | Rem_operation : Symbol → Expr → Expr
  | Delta : list Expr → Expr.
```

Listing 2: Delta-Oriented Programming Semantics

```
Inductive Apply : Expr × Table → Type :=
  | EMPTY
    : Apply (Delta nil, List nil)
  | ADD_object
    : ∀ core sym l,
        Apply (core, List l) →
        Apply (Add_operation sym core, List (sym::l))
  | REMOVE_object
    : ∀ core sym l l',
        Apply (core, List (l++sym::l')) →
        Apply (Rem_operation sym core, List (l++l'))
  | APPLY_delta
    : ∀ expr actions v1 v2,
```

```
      Apply (expr, List v1) →
      Apply (Delta actions, List v2) →
      Apply (Delta (expr::actions), List (v1 ++ v2)).
```

Listing 3: Delta Types

```
Inductive DialectType :=
  | Type_Base : ascii → DialectType
  | Type_Delta : list ascii → DialectType.
```

Listing 4 shows the encoding of typed symbols and delta-actions ($e{:}\tau$), and the safety predicate `Resolved`. Finally, the inductive defintions for well-typed values (table of symbols) is shown in Listing 5. The proof of the safety theorem, which was presented in Section 6, is shown in Listing 6 and the required tactics to solve the verification conditions are given in Listing 7.

Listing 4: Well-Typed Delta Expressions

```
Inductive TypedSymbol : (Symbol × DialectType) → Type :=
  | TE_Symbol :
      ∀ f n, TypedSymbol (Sym f n, Type_Base f).

Inductive Resolved : (list ascii) → Prop :=
  | DS : ∀ t, ¬ In "U" t → Resolved t.

Inductive TypedExpression : (Expr × DialectType) → Type :=
  | TE_Empty :
      Delta nil : Type_Delta nil
  | TE_Symbols_Add :
      ∀ core op s syms,
        TypedSymbol (op, Type_Base s) →
        (Delta core) : (Type_Delta syms) →
        (Add_operation op (Delta core)) : (Type_Delta (s::syms))
  | TE_Symbols_Rem :
      ∀ core op s syms,
        Resolved (s::syms) →
        TypedSymbol (op, Type_Base s) →
        (Delta core) : (Type_Delta syms) →
        (Rem_operation op (Delta core)) : (Type_Delta (remove ascii_dec s syms))
  | TE_Delta :
      ∀ expr actions syms_a syms_b,
        expr : (Type_Delta syms_a) →
        (Delta actions) : (Type_Delta syms_b) →
        (Delta (expr::actions)) : (Type_Delta (syms_a ++ syms_b))
```

Listing 5: Well-Typed Variants

```
Inductive SafeTable : (Table × DialectType) → Type :=
  | TC_Empty : SafeTable (List nil, Type_Delta nil)
  | TC_Delta_Rem : ∀ s syms (sym : Symbol) l l',
                        TypedSymbol (sym, Type_Base s) →
                        SafeTable (List (l++sym::l'), Type_Delta syms) →
                        SafeTable (List (l++l'), Type_Delta (remove ascii_dec s syms))
  | TC_Delta_App : ∀ l1 s1 l2 s2,
                        SafeTable (List l1, Type_Delta s1) →
                        SafeTable (List l2, Type_Delta s2) →
                        SafeTable (List (l1 ++ l2), Type_Delta (s1 ++ s2))
  | TC_Delta_Add : ∀ s syms sym l,
                        TypedSymbol (sym, Type_Base s) →
                        SafeTable (List l, Type_Delta syms) →
                        SafeTable (List (sym::l), Type_Delta (s::syms)).
```

Listing 6: Proof of the Safety Theorem

```
Theorem TypeLemma :
  ∀ expr v t,
  Apply (expr, v) →
  expr : (Type_Delta t) →
  Resolved t →
  SafeTable (v, Type_Delta t).
Proof.
  intros expr v t HEv HTy HWf.
  generalize dependent t.
  dependent induction HEv;
  intros t HExp HDef;
  inversion HExp; subst.
  - apply TC_Empty.
  - apply TC_Delta_Add; auto.
    eapply IHHEv; eauto.
    constructor.
    inversion HDef; subst. intro. apply H.
    apply in_cons; assumption.
  - eapply TC_Delta_Rem; eauto.
    eapply IHHEv; eauto.
    constructor.
    inversion H2; subst. intro. apply H.
    apply in_cons; assumption.
  - apply TC_Delta_App; [eapply IHHEv1 | eapply IHHEv2 ]; eauto;
    inversion HDef; subst;
    constructor; intro; apply H;
    apply in_or_app; [ left | right ]; auto.
Qed.
```

Listing 7: Tactic to prove the `Resolved` and the `Apply` predicates

```
Ltac solve_consistency :=
  repeat match goal with
         | [ ⊢ Resolved _] ⇒ constructor
         | [ ⊢ ¬ In _ _] ⇒ apply not_in_cons; intuition
         | [ H : _ = _ ⊢ False ] ⇒ discriminate H
         | [ H : In _ _ ⊢ False ] ⇒ apply in_inv in H; intuition
       end.

Ltac solve_apply :=
  repeat match goal with
         | [ ⊢ Apply (_,_)] ⇒
           repeat (apply APPLY_delta || apply ADD_object || apply EMPTY)
       end.

Ltac solve_typed_expr :=
  repeat match goal with
         | [ ⊢ TypedExpression (_,_)] ⇒
           repeat (apply TE_Delta || apply TE_Symbols_Add ||
                   apply TE_Empty || apply TE_Symbol)
       end.
```

Appendix C Haskell Interpreter for the PCC Agent

This appendix shows the Haskell source code that implements an interpreter for the big-step semantics of the PCC Agent presented in Table 6, Section 6. The complete source code project is available at https://github.com/esmifro/CertifiedDOP.

Listing 1 shows the algebraic datatypes used to represent tables of symbols, the verification condition proof result and other type constructors for files and directories. The internal function signatures shown in Listing 4 are defined exclusively over the `Internal` domain. Listing 2 shows the algebraic datatypes and types used by the interpreter using Monad transformers (`runCheck`). The correspondence between high-level syntax and system-(Linux)level syntax is stored inside the map `MetaMap`. Abstract programs are created using `Command`.

Listing 1: Internal Definitons

```
data Atom     = Symbol Char String | Flag Char
data Internal = Table [Atom]
              | RelDir (Path Rel Dir)
              | RelFile (Path Rel File)
              | ProofChecker ExitCode
              | ValueList [Internal]
```

Listing 2: Interpreter Definitons

```
type Var        = String
data AbstExpr   = ParseDeltaRPM Var
                | ApplyDeltaRPM Var
                | MakeDeltaRPM Var
                | GetSymbolsTable Var
                | CheckProof Var
                | InstallRPM Var
                deriving (Eq, Show, Ord)
data ConcExpr   = F String (Internal → Internal)
data BoolExpr   = Check AbstExpr
data Command    = Assign Var AbstExpr
                | Seq Command Command
                | Cond BoolExpr Command Command
                | Abort
type Env        = Map String Internal
type MetaMap    = Map AbstExpr ConcExpr
type Checker a  = ReaderT MetaMap (StateT Env (IO)) a
runCheck :: Checker a → MetaMap → Env → IO (a, Env)
```

The interpreter shown in Listing 3 implements the big-step semantics of the PCC Agent. For each of the big-step relations, that is, for variables, expressions and commands, there is a correspondent interpretation function. The map between abstract functions and internal functions is defined in Listing 4.

Listing 3: PCC Agent Interpreter

```
evalVar                 :: Var → Checker Internal
evalVar v               = flip (!) v <$> get
                          >>= λval → return val

evalExp                 :: AbstExpr → Checker Internal
evalExp e               = flip (!) e <$> ask
                          >>= λ(F var fun) → evalVar var
                          >>= λarg → return (fun arg)

evalBool                :: BoolExpr → Checker Bool
evalBool (Check e)      = evalExp e
                          >>= λv → if v ≡ ProofChecker (ExitSuccess)
                                   then return True
                                   else return False
evalComm                :: Command → Checker ()
evalComm (Assign v e)   = evalExp e
                          >>= λval → put =<< insert v val <$> get
```

```
evalComm (Seq a b)        = evalComm a ≫ evalComm b
evalComm (Cond b c1 c2) = evalBool b
                            ≫= λchecked → if checked
                                            then evalComm c1
                                            else evalComm c2
evalComm (Abort)          = liftIO $ putStrLn ("Abort") ≫ exitFailure
```

Listing 4: Meta-Functions and Internal Function Signatures

```
metaFunctions :: [(AbstExpr, ConcExpr)]
metaFunctions = [(ParseDeltaRPM "a", F "a" readDeltaRPM),
                 (ApplyDeltaRPM "x", F "x" applyDeltaRPM),
                 (MakeDeltaRPM "m", F "m" makeDeltaRPM),
                 (GetSymbolsTable "y", F "y" getSymbolsTable),
                 (CheckProof "z", F "z" runChecker),
                 (InstallRPM "x", F "x" installRPM)]
```

with the follwoing internal function signatures:

```
readDeltaRPM   :: Internal → Internal
applyDeltaRPM  :: Internal → Internal
makeDeltaRPM   :: Internal → Internal
getSymbolsTable :: Internal → Internal
runChecker     :: Internal → Internal
installRPM     :: Internal → Internal
```

Finally, the function that runs the interpreter with a given set of arguments is shown in Listing 5.

Listing 5: Entry Function of the PCC Agent

```
runApply
  :: Internal → IO ((), Env)
runApply args
  = do
    let a = Assign "x" (ParseDeltaRPM "a")
        b = Assign "y" (ApplyDeltaRPM "x")
        c = Assign "z" (GetSymbolsTable "y")
        i = Assign "i" (InstallRPM "x")
        prog = Seq a (Seq b (Seq c (Cond (Check (CheckProof "z")) i Abort)))
    runCheck (evalComm prog) (fromList metaFunctions)
             (insert "a" args empty)
```

References

1. Albert E, Arenas P, Puebla G, Hermenegildo M (2006) Reduced Certificates for Abstraction-Carrying Code, Springer Berlin Heidelberg, Berlin,

Heidelberg, pp 163–178. DOI 10.1007/11799573_14
2. Apel S, Batory D, Kstner C, Saake G (2013) Feature-Oriented Software Product Lines: Concepts and Implementation. Springer Publishing Company, Incorporated
3. Arndt J, Behlert S, et al (2006) SUSE Linux. Tech. rep., Novel Inc.
4. Ayavoo D, Pont MJ, Parker S (2005) Observing the development of a reliable embedded system. In: Proceedings of the 10th Ada-Europe International Conference on Reliable Software Technologies, Springer-Verlag, Berlin, Heidelberg, Ada-Europe'05, pp 167–179, DOI 10.1007/11499909_14
5. Bailey JW, Basili VR (1981) A meta-model for software development resource expenditures. In: Proceedings of the 5th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '81, pp 107–116
6. Basili VR, Weiss DM (1984) A methodology for collecting valid software engineering data. IEEE Trans Softw Eng 10(6):728–738, DOI 10.1109/TSE.1984.5010301
7. Basili VR, Selby RW, Hutchens DH (1986) Experimentation in software engineering. IEEE Trans Softw Eng 12(7):733–743
8. Batory D, Sarvela J, Rauschmayer A (2004) Scaling Step-Wise Refinement. IEEE TSE 30(6):355–371
9. Bavota G (2012) Using structural and semantic information to support software refactoring. In: 2012 34th International Conference on Software Engineering (ICSE), pp 1479–1482, DOI 10.1109/ICSE.2012.6227057
10. Berger T, Rublack R, Nair D, Atlee JM, Becker M, Czarnecki K, Wąsowski A (2013) A survey of variability modeling in industrial practice. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, ACM, New York, NY, USA, VaMoS '13, pp 7:1–7:8, DOI 10.1145/2430502.2430513
11. Bernardeschi C, Francesco ND, Lettieri G, Martini L, Masci P (2008) Decomposing bytecode verification by abstract interpretation. ACM Trans Program Lang Syst 31(1):3:1–3:63, DOI 10.1145/1452044.1452047
12. Bertot Y, Castran P (2010) Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions, 1st edn. Springer Publishing Company, Incorporated
13. Bezemer CP, Mcintosh S, Adams B, German DM, Hassan AE (2017) An empirical study of unspecified dependencies in make-based build systems. Empirical Softw Engg 22(6):3117–3148, DOI 10.1007/s10664-017-9510-8
14. Biere A, Biere A, Heule M, van Maaren H, Walsh T (2009) Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam, The Netherlands
15. Bordeaux L, Hamadi Y, Zhang L (2006) Propositional satisfiability and constraint programming: A comparative survey. ACM Comput Surv 38(4), DOI 10.1145/1177352.1177354
16. Brown AW, Wallnau KC (1996) A framework for evaluating software technology. IEEE Software 13(5):39–49, DOI 10.1109/52.536457

17. Bryant RE, O'Hallaron DR (2010) Computer Systems: A Programmer's Perspective, 2nd edn. Addison-Wesley Publishing Company, USA
18. Burow N, Carr SA, Brunthaler S, Payer M, Nash J, Larsen P, Franz M (2017) Control-flow integrity: Precision, security, and performance. ACM Comput Surv 50:16:1–16:33, DOI 10.1145/3054924
19. Calcote J (2010) Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool. No Starch Press
20. Calder B, W Phillips L, Tybout A (1982) The concept of external validity. Journal of Consumer Research 9:240–244
21. Catuogno L, Visconti I (2003) A format-independent architecture for run-time integrity checking of executable code. In: Cimato S, Persiano G, Galdi C (eds) Security in Communication Networks, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 219–233
22. Chesta C, Damiani F, Dobriakova L, Guernieri M, Martini S, Nieke M, Rodrigues V, Schuster S (2016) A toolchain for delta-oriented modeling of software product lines. In: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II, pp 497–511, DOI 10.1007/978-3-319-47169-3_40
23. Clements P, Northrop L (2001) Software Product Lines: Practices and Patterns. Addison Wesley Longman
24. Cousot P, Cousot R (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM, New York, NY, USA, POPL '77, pp 238–252, DOI 10.1145/512950.512973
25. De Moura L, Bjørner N (2008) Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer-Verlag, Berlin, Heidelberg, TACAS'08/ETAPS'08, pp 337–340
26. DeTreville J, Leijen D, Swierstra W (2006) Dependable software deployment. Tech. rep., Microsoft Research
27. Donald J (2003) Improved portability of shared libraries. Tech. rep., Princeton University
28. Drepper U (2011) How to Write Shared Libraries. Tech. rep., Red Hat Inc.
29. Drusinsky D (2006) Chapter 1 - Formal Requirements and Finite Automata Overview. In: Drusinsky D (ed) Modeling and Verification Using UML Statecharts, Newnes, Burlington, pp 1 – 41, DOI 10.1016/B978-075067949-7/50003-9
30. Drusinsky D (2006) Chapter 2 - Statecharts. In: Drusinsky D (ed) Modeling and Verification Using UML Statecharts, Newnes, Burlington, pp 43 – 102, DOI 10.1016/B978-075067949-7/50004-0
31. Ducasse S, Nierstrasz O, Schärli N, Wuyts R, Black AP (2006) Traits: A mechanism for fine-grained reuse. ACM Trans Program Lang Syst 28(2):331–388, DOI 10.1145/1119479.1119483

32. European Commission (2017) eCall: Time saved = lives saved. https://ec.europa.eu/digital-single-market/en/ecall-time-saved-lives-saved
33. Fenton NE (1991) Software Metrics: A Rigorous Approach. Chapman & Hall, Ltd., London, UK, UK
34. Garfinkel S (1996) PGP: Pretty Good Privacy, 1st edn. O'Reilly & Associates, Inc., Sebastopol, CA, USA
35. Ge Y, de Moura L (2009) Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani A, Maler O (eds) Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 306–320, DOI 10.1007/978-3-642-02658-4_25
36. GmbH VSI (2018) RTT-DCC: Data & Control Coupling Analyser. https://www.verified.de/products/rtt-dcc/
37. Gosling J, Joy B, Steele GL, Bracha G, Buckley A (2014) The Java Language Specification, Java SE 8 Edition, 1st edn. Addison-Wesley Professional
38. Guo C, Ren S, Jiang Y, Wu PL, Sha L, Berlin RB Jr (2016) Transforming medical best practice guidelines to executable and verifiable statechart models. In: Proceedings of the 7th International Conference on Cyber-Physical Systems, IEEE Press, Piscataway, NJ, USA, ICCPS '16, pp 34:1–34:10
39. Haber A, Rendel H, Rumpe B, Schaefer I, van der Linden F (2011) Hierarchical Variability Modeling for Software Architectures. In: Proceedings of the 15th International Software Product Line Conference, IEEE, pp 150–159, DOI 10.1109/SPLC.2011.28
40. Habets T (2012) Shared libraries diamond problem. https://blog.habets.se/2012/05/Shared-libraries-diamond-problem.html
41. Hallsteinsen S, Hinchey M, Park S, Schmid K (2008) Dynamic software product lines. Computer 41(4):93–95, DOI 10.1109/MC.2008.123
42. Harel D, Politi M (1998) Modeling Reactive Systems with Statecharts: The Statemate Approach, 1st edn. McGraw-Hill, Inc., New York, NY, USA
43. Hawkins RD, Kelly TP (2009) Software safety assurance - what is sufficient? In: 4th IET International Conference on Systems Safety 2009. Incorporating the SaRS Annual Conference, pp 1–6, DOI 10.1049/cp.2009.1542
44. Hermenegildo MV, Albert E, López-García P, Puebla G (2005) Abstraction carrying code and resource-awareness. In: Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM, New York, NY, USA, PPDP '05, pp 1–11, DOI 10.1145/1069774.1069775
45. Hutchens DH, Basili VR (1985) System structure analysis: Clustering with data bindings. IEEE Trans Softw Eng 11(8):749–757, DOI 10.1109/TSE.1985.232524
46. Hutchinson J, Rouncefield M, Whittle J (2011) Model-driven engineering practices in industry. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '11, pp 633–

642, DOI 10.1145/1985793.1985882
47. Iscoe N, Williams GB, Arango G (1991) Domain modeling for software engineering. In: [1991 Proceedings] 13th International Conference on Software Engineering, pp 340–343, DOI 10.1109/ICSE.1991.130660
48. Jiang L, Su Z (2008) Profile-guided program simplification for effective testing and analysis. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, SIGSOFT '08/FSE-16, pp 48–58, DOI 10.1145/1453101.1453110
49. Kernighan BW, Ritchie DM (1977) The M4 Macro Processor. Tech. rep., Bell Laboratories
50. Kirovski D, Drinić M, Potkonjak M (2002) Enabling trusted software integrity. SIGPLAN Not 37(10):108–120, DOI 10.1145/605432.605409
51. Krueger CW (1992) Software reuse. ACM Comput Surv 24(2):131–183, DOI 10.1145/130844.130856
52. Lienhardt M, Damiani F, Testa L, Turin G (2018) On checking delta-oriented product lines of statecharts. Science of Computer Programming 166:3 – 34, DOI 10.1016/j.scico.2018.05.007
53. Lindholm T, Yellin F (1999) Java Virtual Machine Specification, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA
54. Martin JC (1997) Introduction to Languages and the Theory of Computation, 2nd edn. McGraw-Hill Higher Education
55. McCabe TJ (1976) A complexity measure. In: Proceedings of the 2Nd International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, USA, ICSE '76, pp 407–
56. Menon V, Pingali K (1999) A case for source-level transformations in matlab. SIGPLAN Not 35(1):53–65, DOI 10.1145/331963.331972
57. Moura L, Bjørner N (2009) Satisfiability modulo theories: An appetizer. In: Oliveira MV, Woodcock J (eds) Formal Methods: Foundations and Applications, Springer-Verlag, Berlin, Heidelberg, pp 23–36, DOI 10.1007/978-3-642-10452-7_3
58. Necula GC (1997) Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, POPL '97, pp 106–119, DOI 10.1145/263699.263712
59. Nelson G (ed) (1991) Systems Programming with Modula-3. Prentice-Hall, Inc., Upper Saddle River, NJ, USA
60. Oliveira N, ao Varanda Pereira MJ, Henriques PR, da Cruz D (2009) Domain-Specific Languages - A Theoretical Survey. In: Proceedings of the 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA'2009), pp 35–46
61. Parkes AP (2008) Finite State Transducers, Springer London, London, pp 189–207. DOI 10.1007/978-1-84800-121-3_8
62. Percival C (2006) Matching with mismatches and assorted applications. PhD thesis, University of Oxford, UK

63. Pohl K, Böckle G, Linden FJvd (2005) Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA
64. Raistrick C, Francis P, Wright J (2004) Model Driven Architecture with Executable UML(TM). Cambridge University Press, New York, NY, USA
65. Ramsey HR, Atwood ME, Van Doren JR (1983) Flowcharts versus program design languages: An experimental comparison. Commun ACM 26(6):445–449, DOI 10.1145/358141.358149
66. Rodrigues V, Lopes JC, Moreira A (2008) An hybrid design solution for spacecraft simulators. In: Proceedings of the Forum at the CAiSE'08 Conference, Montpellier, France, June 18-20, 2008, pp 29–32
67. Rodrigues V, Akesson B, Florido M, de Sousa SaM, Pedroso JaP, Vasconcelos P (2015) Certifying execution time in multicores. Sci Comput Program 111(P3):505–534, DOI 10.1016/j.scico.2015.06.006
68. Røst TB, Seidl C, Yu IC, Damiani F, Johnsen EB, Chesta C (2018) Hyvar. In: Mann ZÁ, Stolz V (eds) Advances in Service-Oriented and Cloud Computing, Springer International Publishing, Cham, Communications in Computer and Information Science, vol 824, pp 159–163, DOI 10.1007/978-3-319-79090-9_12
69. Rumbaugh J, Jacobson I, Booch G (2004) Unified Modeling Language Reference Manual, The (2Nd Edition). Pearson Higher Education
70. Schaefer I, Bettini L, Bono V, Damiani F, Tanzarella N (2010) Delta-Oriented Programming of Software Product Lines. In: Bosch J, Lee J (eds) Software Product Lines: Going Beyond (SPLC 2010), Springer, Lecture Notes in Computer Science, vol 6287, pp 77–91, DOI 10.1007/978-3-642-15579-6_6
71. Schaefer I, Rabiser R, Clarke D, Bettini L, Benavides D, Botterweck G, Pathak A, Trujillo S, Villela K (2012) Software diversity: state of the art and perspectives. International Journal on Software Tools for Technology Transfer 14(5):477–495, DOI 10.1007/s10009-012-0253-y
72. Schürr A, Selic B (eds) (2009) Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings, Lecture Notes in Computer Science, vol 5795, Springer, DOI 10.1007/978-3-642-04425-0
73. Seidl C, Schaefer I, Aßmann U (2014) Deltaecore - A model-based delta language generation framework. In: Modellierung 2014, 19.-21. März 2014, Wien, Österreich, pp 81–96
74. Shneiderman B, Mayer R, McKay D, Heller P (1977) Experimental investigations of the utility of detailed flowcharts in programming. Commun ACM 20(6):373–381, DOI 10.1145/359605.359610
75. Stallman RM, McGrath R (2002) GNU Make: A Program for Directed Compilation. Free Software Foundation
76. Team CAS (2004) Clarification of Structural Coverage Analyses of Data Coupling and Control Coupling. https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/archive/

77. Thüm T, Apel S, Kästner C, Schaefer I, Saake G (2014) A classification and survey of analysis strategies for software product lines. ACM Comput Surv 47(1):6:1–6:45, DOI 10.1145/2580950
78. Thüm T, Kästner C, Benduhn F, Meinicke J, Saake G, Leich T (2014) Featureide: An extensible framework for feature-oriented software development. Sci Comput Program 79:70–85, DOI 10.1016/j.scico.2012.06.002
79. Tichy WF, Lukowicz P, Prechelt L, Heinz EA (1995) Experimental evaluation in computer science: A quantitative study. J Syst Softw 28(1):9–18, DOI 10.1016/0164-1212(94)00111-Y
80. Tu Q, Godfrey MW (2001) The build-time software architecture view. In: Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, pp 398–407, DOI 10.1109/ICSM.2001.972753
81. Turner AJ (1975) Iterative enhancement: A practical technique for software development. IEEE Trans Softw Eng 1(1):390–396, DOI 10.1109/TSE.1975.6312870
82. Vaughan GV, Elliston B, Tromey T, Taylor IL, Mac Kenzie D (2001) GNU Autoconf, Automake and Libtool. Expert Insight into Porting Software and Building Large Projects using GNU Autotools. New Riders, Indianapolis, IN
83. Weiss DM, Basili VR (1985) Evaluating software development by analysis of changes: Some data from the software engineering laboratory. IEEE Transactions on Software Engineering SE-11(2):157–168, DOI 10.1109/TSE.1985.232190
84. Weyuker EJ (1988) Evaluating software complexity measures. IEEE Trans Softw Eng 14(9):1357–1365, DOI 10.1109/32.6178
85. Williams M, Grajales C, Kurkiewicz D (2013) Assumptions of multiple regression: Correcting two misconceptions. Practical Assessment, Research & Evaluation
86. Wolverton RW (1974) The cost of developing large-scale software. IEEE Transactions on Computers C-23(6):615–636, DOI 10.1109/T-C.1974.224002
87. Wong B, Czajkowski G, Daynes L (2003) Dynamically loaded classes as shared libraries: an approach to improving virtual machine scalability. In: Proceedings International Parallel and Distributed Processing Symposium, DOI 10.1109/IPDPS.2003.1213123
88. Yourdon E (1986) Techniques of Program Structure and Design, 1st edn. Prentice Hall PTR, Upper Saddle River, NJ, USA
89. Yu D, Hamid NA, Shao Z (2004) Building certified libraries for PCC: dynamic storage allocation. Science of Computer Programming 50(1):101 – 127, DOI 10.1016/j.scico.2004.01.003, 12th European Symposium on Programming (ESOP 2003)