Transformation of the Software Integration Process

From Classic Software Integration to Co-Integration

M.Sc. Christopher Schwager, ITK Engineering GmbH, Ruelzheim

Abstract
The way how automotive systems are currently designed, implemented and deployed is undergoing a radical shift. This is because of the main technology drivers, namely highly automated driving, connectivity, and electrification as well as new players from the IT industry entering the market with new technologies. Additionally, OEMs have started to take over more responsibility within the value chain. They choose different suppliers for their hardware and software parts or even implement their own “OEM Operating System”. However, managing different software deliveries and speeding up the validation feedback increases the complexity of integration. Today’s classic approach of manually integrating and validating the software is thus no longer sustainable to tackle the complexity. This leads to a transformation of the software integration process.

The key requirement of a modern software integration process is achieving maximal flexibility and pace while minimizing the validation efforts. Today’s integration process is exactly the opposite. Typically, the software integrator at Tier-1 level receives the software deliveries in a random format together with an integration manual. When the software is manually integrated based on given instructions, existing test vectors have to be adjusted or new test vectors have to be created. Both derive from minimal knowledge to validate the newly integrated software and to detect regressions.

This paper emphasizes on new available technologies such as software packaging, Continuous X as well as scalable cloud solutions in combination with well-established approaches like HiL testing. Together they form a high collaborative co-integration environment. Such an environment builds the foundation for scalability for a high number of users (>1000), for a homogeneous build framework with seamless support of multiple operating systems and clients as well as for a high grade of automation which provides fast feedback loops.
1. Mobility mega trends as drivers for modern software integration

The mobility value change is undergoing one of the most transformational disruptions shaped by personalization, automation, connectivity and electrification. Unrestricted mobility is a key aspect of our day-to-day lives. Smart connectivity and user-friendly apps turn mobility into a personal all-in-one service. Individuality is the new desire. Additionally, self-driving vehicles can be an important part of the solution of the challenges of mobility we are currently facing like greater road safety as well as the lack of time and space. Connecting systems and services inside and outside vehicles opens new opportunities to develop and improve vehicles and mobility services.

Software-defined car requires new integration approaches

Those megatrends lead to a massive increase of software (SW) share which still doubles every three years. Furthermore, integration efforts are skyrocketing due to increasing difficulties to handle the complexity of distributed SW features with current EE architectures. New developments focus on centralized / zonal EE architectures as a countermeasure to enable scalability and to reduce the existing investment gap as well as the total cost of ownership of e.g., software components. Simultaneously, OEMs force the separation of hardware (HW) and SW along with deepening their impact on the value chain by taking over more control of critical system components. Ultimately, this leads to a SW-defined car which demands for a heavy multi-party collaboration to be successful. Therefore, there are also changed key requirements for a modern SW integration process because the SW integrator is often in the middle of such a collaboration.

2. Limitations of today's classic software integration

Today's classic software integration process, which is applied daily in most ECU projects, is similar to what is depicted in Fig. 1.

![Fig. 1: Classic Software Integration](https://doi.org/10.51202/9783181023846-215)
Different teams start developing based on the current SW revision \([N]\) with their next development cycle. The developers adjust to SW platform changes like interfaces, or they introduce new features or bugfixes. The new / changed SW portion is dropped to the integrator for integration once the changed version is available. The integrator then starts to stitch the different deliverables to a new SW release candidate during the integration phase. Based on this release candidate and testing, a new SW revision \([N+1]\) is created. Afterwards the next cycle starts all over again.

**Key Requirements: agile development, fast feedback, and automation**

It is getting clear that there are multiple bottlenecks by trying to apply the classic software integration approach to the SW-defined Car and its multi-party collaboration demands as well as agile development models.

New features, SW components and services are developed mostly independently to each other. But problems typically occur or are detected when the different parts are integrated and must fulfil their functionality. There are often development cycles of 10 to 12 weeks which is often also the duration until feedback is available for a new software delivery. This deadtime is anything except “agile”. This is not sufficient for a modern software integration approach.

Tier-1 was responsible for the complete system on ECU level in the worn-out OEM / Tier-1 working model. This is even true if the Tier-1 had to integrate some standard modules or software components provided by OEM. However, when the OEM takes over more responsibility in the value chain and individually sources different SW parts or create them on its own, the role of integration changes. The complexity to manage SW deliveries increases dramatically and demands a close cooperation as well as automation.

3. **Transformation of software integration processes**

Software integration must undergo a transformation to achieve the key requirements. The next three subchapters focus on three main transformations areas: Software Packaging, Build Framework and Continuous X.
3.1 Software Packaging

Typical content of SW deliveries differs in a wide range. On the one hand, SW could be source code (white box) and on the other hand binary (black box). Documentation is another big variation point. There is sometimes no documentation at all e.g., during early project phases when everyone is focusing on the features and building up technical depts. Another extreme is if so many documents like integration manuals, safety manuals, changelogs and more are provided, and the integrator cannot find the necessary information in the bounty of pages. Taken all together, software deliveries are surprise packages. Handling those deliveries in a high number requires a big percentage of manual work and the following questions weight even heavier:

- When can I accept deliveries for integration?
- Is the delivery completed?
- Are there any dependencies that must be considered?
- Will it really run with the current SW revision [N] and how can I prove it?

The introduction of a package manager enables structuring deliverables. In detail, this means that a binding package content such as a SW component itself in binary, machine readable documentation, SW component test vectors and metrics like test coverage and resource consumptions are contained in every delivery in the exact same manner. Consequently, automated (continuous) integration and validation is possible based on the software package. Even the dependency between SW components could be modelled to speed up the impact analysis. A widely used package manager for C/C++ is Conan [1].
### 3.2 Build Framework

The second transformation area focuses on the build infrastructure. There are different target systems, such as microcontrollers and microprocessors, in a typical vehicle computer which also expect architecture specific binary code. Multiple compilers and linkers build the basis to cover this technical aspect, but it could be configured with a huge number of parameters. Additionally, other tools such as an interface to the previously introduced package manager or tools for creating documentation is needed to simplify the daily work of developers as well as to create an all-in-one user experience. Independently of the creation, validating according to necessary standards like ISO26262 and maintaining such build frameworks are error prone and cost intensive. Therefore, a build framework that fulfils the involved stakeholder needs must be established at the integrator and distributed. Using a package manager for the build framework similarly adds the possibility of versioning and dependency management. Moreover, putting everything within a lightweight container virtually enables shipping and execution everywhere. This also includes automation pipelines within a cloud or on premise. A step further would be creating an Internal Developer Platform (IDP) [2]. An IDP is a layer on top of the technologies and tooling that an engineering team is using and enables developer self-service. This means, e.g., in the context of software integration that a SW component developer could setup a new pipeline within seconds and can deploy the SW component on a testbench with just a few clicks.

### 3.3 Continuous X

It is quite easy to open the time frame for integration from a concentrated point in time to a longer period. But this does not necessarily speed up the development cycles or provide faster feedback loops. The main problem still exists that testing takes place during the release phase. Therefore, there is no benefit of continuous integration without automated integration tests at the yellow-green dots shown in Fig. 3.

![Continuous Software Integration](https://example.com/continuous_integration.png)

Fig. 3: Continuous Software Integration

Let us assume that a development team wants to integrate a newly enhanced SW component to the next release. The team creates a pull request as initiation of the integration phase.
Multiple test levels, as stated in Fig. 4, ensure the quality and integrability. Once these phases are successfully passed, the merge is the actual integration.

Fig. 4: Continuous testing as key for continuous integration

The “delivery acceptance tests” are executed within the first test level. The test goal is to automatically test pre integration quality gates like test coverage or other defined metrics. Additionally, general integrability of the SW component in the current SW revision \([N]\) is validated. Integrity check of the delivery and metric compliance as well as rudimentary smoke test vectors are the test measures that are applied here.

The second test phase “delta integration tests” focuses on the interaction of the SW component with its communication partners as well as on the resource usage. The validation is executed with the help of specific SW component test vectors that stimulate impacted event chains. The tests are selected based on modelled dependencies between SW components.

The last test level “integration tests” is very similar to phase two but does not strip the test cases down to an essential minimum. Multiple SW component integration branches should be combined as this phase is still highly time-consuming.

4. **Co-Integration as a new foundation**

Consistently carrying out the transformation on all areas transforms the integrator from an on-demand software stitcher into an infrastructure provider. Such a Co-Integration platform consists of two main parts: an exchange platform and the Co-Integration environment itself.

The exchange platform builds the foundation of the Co-Integration platform. It could be hosted on premise or in the cloud. It is the central point of collaboration of all SW component developers, the software platform developer as well as the software integrator. Fig. 5 shows features and content of such an exchange platform. Central elements are the artifact repository manager as well as the continuous x capabilities.
Those two elements are intensively used within the Co-Integration platform which is depicted in the middle of Fig. 6. The software platform as well as multiple SW components independently develop their piece of SW in their environments based on a build framework or within the Co-Integration space. At the end of their development process a software package is created and delivered to the artifact repository manager for integration. The actual integration is then carried out following the validation process like described in the Continuous X chapter before. At the end of the automation the integrated and validated software is available via the Exchange platform or feedback to e.g., failed testcases are provided back to the SW component developers for analysis.

Unfortunately, Co-Integration is not the silver bullet. The HiL infrastructure is the bottleneck of the concept as they do not scale due to their cost-intensive sourcing. By assuming that there are roughly 20 SW components where 6 of them deliver daily and the others 1-2 times a week results in a >95% utilization of 3 HiLs. Additionally, SiL as alternative is not applicable to the integration use case as HW / SW interaction is an important perspective that could not be
missed. SW emulations of HW accelerators are often also not available or too slow which excludes huge parts of the software.

5. Conclusion

Achieving maximal flexibility and pace while minimizing the validation effort is the key requirement of modern software integration. Today’s classic approach of manually integrating and validating the software is no longer sustainable. Multiple transformations must be carried out in areas such as software packaging, build frameworks, and continuous x. A high collaborative Co-Integration environment is the foundation for scalability, homogeneous build frameworks as well as a high grade of automation.

[1] https://conan.io/