Deterministic Execution Sequence in Component Based Multi-Contributor Powertrain Control Systems

Denis Claraz[1], Stefan Kuntz[2], Ulrich Margull[3], Michael Niemetz[4], Gerhard Wirrer[2]
[1]Continental Automotive France S.A.S., 1 Av. Paul Ourliac, BP 83649, 31036 Toulouse Cedex 1, France, denis.claraz@continental-corporation.com
[2]Continental Automotive GmbH, P.O. Box 100943, D-93009 Regensburg
{stefan.kuntz,gerhard.wirrer}@continental-corporation.com
[3]1 mal 1 Software GmbH, Maxstraße 31, D-90762 Fürth, ulrich.margull@1mal1.com
[4]University of Applied Sciences Regensburg, LaS^3 - Laboratory for Safe and Secure Systems, Faculty of Electrical Engineering and Information Technology, P.O. Box 12 03 27, D-93025 Regensburg, michael.niemetz@hs-regensburg.de

Abstract: Modern complex control applications, e.g. engine management systems, typically are built using a component based architecture, enabling the reuse of components and allowing to manage the complexity of the application in terms of functional content, size and interfaces. This approach of independently developed components is supported by the concepts available in AUTOSAR and therefore can be expected to gain increasing importance. However, due to the nature of the task of control applications there still is a strong coupling between individual parts of the components resulting in signal chains and consequently in sequencing requirements. The challenge to get such execution sequences implemented correctly is increased, as often the components are delivered by different and external parties.
Our approach extends the idea of functional partitioning of the application into the time domain by defining a system of phases with a fixed sequence and a defined content. This allows to design components right from the beginning into this sequencing frame like they are designed today into the component partitioning frame and to define a system sequencing across different suppliers.

Keywords: Real-time systems, integration, reuse, automotive, sequencing, modular software

I. Introduction

Modern control applications, like the ones used in state of the art automotive powertrain systems for controlling combustion engines, require to manage a large number of input signals delivered by all kinds of sensors which are connected directly to the control unit or via serial buses of low to high bandwidth capabilities (e.g. LIN, CAN FlexRay). In addition, a considerable amount of output actuators have to be driven, most of them with hard real time requirements ($\sim \mu s$) and data has to be provided to other control units via the serial buses.
The relations between all these input and output signals are managed by a variety of strongly interacting control algorithms consisting of different functions (or features). This strong interaction seems to be a particularity of engine management systems, e.g. compared to body control units where there are fewer interactions between different features.

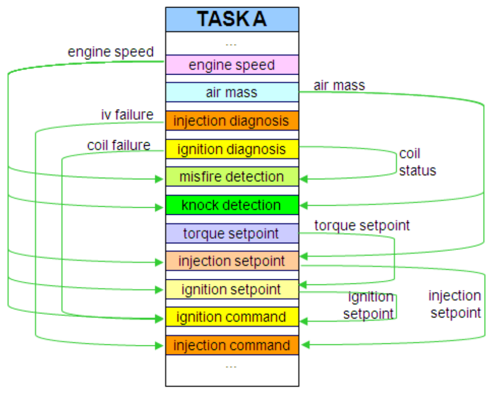

Figure 1. Simplified view of the dynamic dependencies between entry points of different components caused e.g. by signal flow.

During the last two decades, the increasing size of the control application software lead to the development of component based software architectures in this application domain. However, the tight coupling between different components (a large number of components which are interacting closely by exchanging many signals to implement the desired control behavior) as it is shown in Figure 4 as well as between different time domains (see Figure 1) remains, making the task of integration of different components into one correctly working device a challenge. For more information on this subject, see also [1].
The mentioned dependencies in the time domain result typically in data flow sequence dependencies, thus focusing the attention of component developers as well as system integrators on the aspect of data flow details and leading to more and more complex data flow dependencies. In contrary, the approach we present in this work breaks this vicious circle by moving back the sequence of execution into the center of attention, again, thus simplifying independent component development as well as their later integration into a product.

In the first section of this paper, we will depict the main architectural characteristics of Engine Management System (EMS) software, as background information to understand our motivation and approach. This section will also describe the consequences of these characteristics on the integration process. In a further section, we will describe the practices used in our domain, and the state of the art known by us. Then, the support provided by the AUTOSAR standard will be mentioned, and then the phase concept will be described, including all its related impacts. An additional chapter will describe the complementary means developed in order to completely cover the topic of calculation sequence management.

II. Background

The context of Engine Management Systems (EMS) software is dominated by the classical business drivers of cost reduction (higher integration), shortening of time-to-market (higher reuse), and increase of quality. In parallel to these influence factors, the most important technical constraint for EMS design is of course the reduction of emissions and fuel consumption, which means an increasingly fine adjustment of engine parameters, leading to growing computation needs and tighter real-time constraints.

A. Technical Complexity: some figures

Consequently, an EMS software is characterized by its technical complexity which is induced by the introduction of new standards, legal requirements and new technologies like multi-core controllers, AUTOSAR, model driven development, etc. A typical modern high-end EMS (6 cylinder, gasoline, direct injection) controls today more than 200 inputs (e.g. sensors) and outputs (e.g. actuators). The corresponding software is built from nearly 2.000 modules ("atomic SW components"), more than 5.000 source files and half a million lines of C-code. Nearly 10.000 executable entities ("Runnables", "Services", "Macros") manipulate more than 30.000 data objects. In terms of resource consumption this results in 1.8 MB of program flash, and 100 kB of RAM.

A good impression of this technical complexity is given by Figure 2 which shows the number of runnables to be integrated on a project for some standard execution rates.

B. Coupling and partitioning

Coupling between components is a major characteristic of Engine Management Systems. Even if a functional partitioning following the physical system architecture (all functions related to one system component grouped into one SW abstraction) is applied on the SW, there remains a strong coupling between the parts, making the job of integration and development harder.

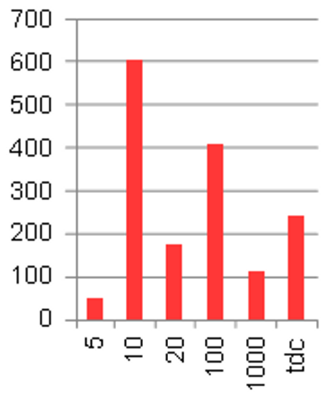

Figure 2. Number of runnables to be integrated in some typical tasks. Not all tasks/runnables are represented.

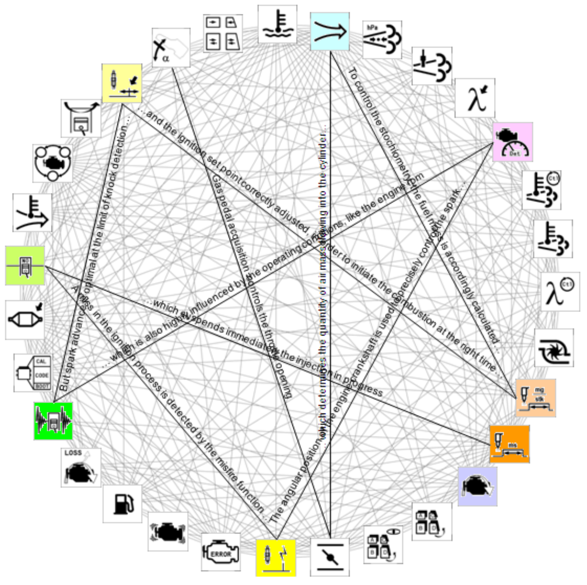

Figure 3. Representation of some functional aggregates, and their relationships. The software link, visible as data or control flow between the components is a consequence of the physical interaction between the controlled system actuators via the physical processes of the controlled system.

The main reason for this the fact that the managed sensors and actuators interact on the same physical process/system: the combustion engine.

This becomes clear in a small (and simplified) example:

When the driver presses the gas pedal, the throttle gets opened, resulting in an higher mass air flow into the intake.

To correctly control the stochiometry, the fuel mass to be injected is adjusted according to the increase in available oxygen. Then, the spark advance is adjusted in order to initiate combustion at the right time.

Depending on engine load, temperature and engine speed, the risk of engine knocking arises, requiring additional adjustments in the ignition timing.

At the end, we see that all these system functions, which are controlling (apparently independent) sensors

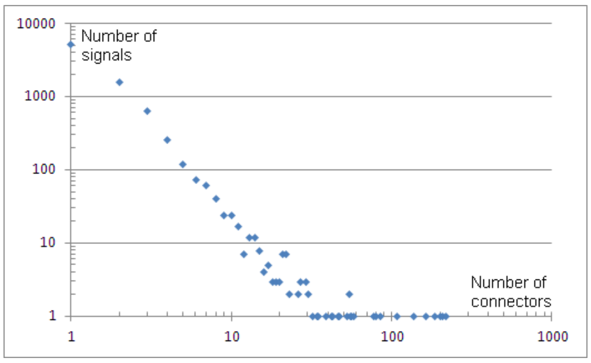

Figure 4. Representation of the coupling between modules, based on the involvement of signals in different connectors.

and actuators, and are implemented as software functions, permanently interact with each other, as seen in Figure 3.

The big challenge is to define a functional (or static) partitioning, which provides the lowest coupling between the parts. Continental introduced such functional partitioning some years ago, together with a reuse concept based on so-called aggregates (see [2]), a strategy widely used in the meantime which can be found also in AUTOSAR concepts.

A clear consequence of the coupling is seen in Figure 4 which shows the relationship between signals and their involvement in connectors: Whereas around 5.000 signals (data objects) are involved in only one connector (1:1 relation), many signals are involved in more than 10 connectors, and some are involved in more than 100 connectors (1:100 relation), i.e. coupling together 100 modules.

Moreover, it has to be noted, that the coupling between modules does not depend only on quantity of signal connectors, but also on their nature, as well as the type of usage of the signal.

A direct consequence of this coupling (signal flow) between the modules is a coupling (signal flow) between the corresponding runnables. And, as there are typically several runnables per module (for diverse reasons like e.g. functionality or efficiency), a corresponding picture of Figure 3 can be drawn for the timing domain: Operating system tasks play the role of integration artifacts in the timing domain like aggregates (compositions) play it in the static/functional domain of the architecture. Figure 5 provides an overview over the dependencies between the tasks available in a typical system.

C. Component based development

In order to reduce price and time to market, a strategy of component based development has been developed, leading to a high level of reuse. The organization and responsibilities have been adapted to this process, as well as the SW architecture: For instance, the static partitioning has been defined to master the above describe

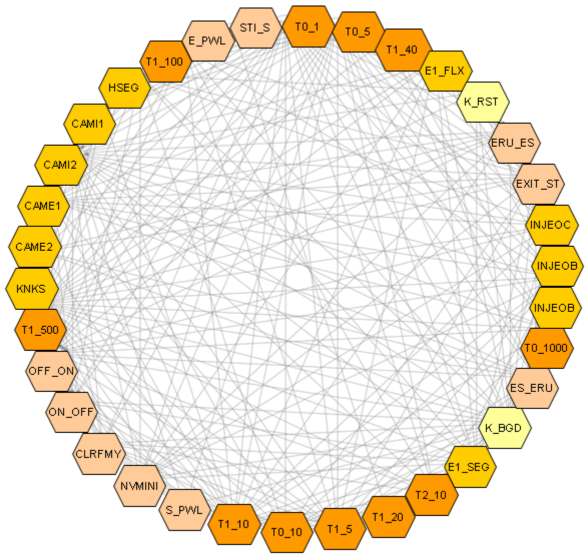

Figure 5. Representation of some tasks ("Timing Aggregates"), and their relationships. Information which is produced at a certain rate is often needed in functions running at different rates.

coupling, and to ease reuse. Generic functions, like for instance the engine rotation and speed acquisition or diagnostic management, are developed only once centrally by a generic team, and then "just" integrated in different projects. Thanks to a standardized partitioning and thanks to advanced configuration capabilities (particularly important for a multi-customer with diverging needs approach), several millions of our ECUs in the field share the same sub-set of modules.

This component based approach is becoming more common, and allows to serve a wide range of business models: From the turn-key project, where the complete system (mechatronical components, electronic hardware, software) is designed by Continental, to "box-business" projects, where we just deliver an ECU plus low-level software layer. Finally a typical project is an integration of components developed by Continental, our customers, other standard component suppliers (e.g. OS, communication layers), and competitors. It is clear that in this context, integration — and the communication about the integration related requirements — becomes a central topic.

D. Dynamic integration – a multi-dimensional problem

While static integration takes care of choosing components, configuring them, establishing signal connections, stubbing or adapting missing parts, the dynamic (timing domain) integration consists, among other things, in calling the runnables in the right place. While a mistake in the static integration is in general immediately detected, a mistake in the dynamic integration may be very hard to identify and reproduce, as its effects may be very sporadic.

Basically, the activity of dynamic integration requires for each of the runnables:

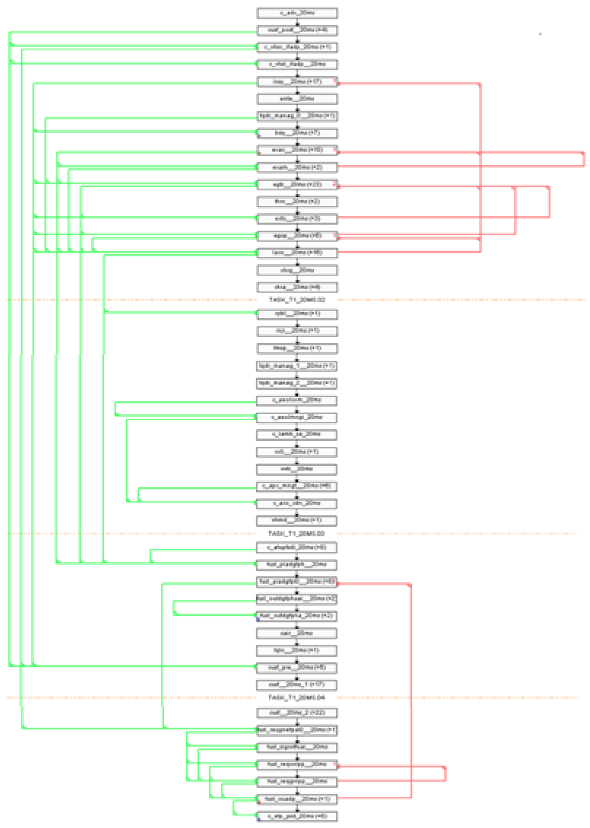

Figure 6. Real life example of a task, with dependencies between the runnables. Each box represents a runnable, green lines represent a "natural flow", red lines represent a "reverse flow" generating retard. Each line gathers one or more signals.

1) Selection of the calculations rate and a matching OS task. Here, the required recurrence is not the only criteria.
2) Selection of position in the task, taking into account different constraints: sequence constraints relative to other runnables, data protection needs, timing constraints, parallel execution on multi-core systems, etc.

The topic we address in this paper is the topic of dynamic integration, and in particular the correct handling of sequence of treatments.

III. State of the Art Approach

The sequence is today empirically defined and mostly driven by a data flow approach: data are supposed to be produced before being consumed. Nevertheless, due to the amount of data flow, this strategy is difficult to apply, particularly including the communication between different software component suppliers. This can be easily understood by just having a look at a real-life example, as shown in Figure 6.

Reverse flows and algebraic loops cannot be avoided in every case, and do not have always a functional impact. The resolution of these "flow weaknesses" requires an identification of the need, establishment of priorities/trade-offs (in particular in the case of loops), and finally leads to a high integration effort, when feasible.

One concept introduced by Continental already with the EMS2 platform in the late 90's, which helps to perform integrations correctly and with a reasonable effort, is the simple approach of Schedulers: These "front-end" functions reduce significantly the integration effort on project side by predefining once re-usable sequences on reused component side.

The main shortcoming of this approach is that only a chosen sequence can be documented (on component level as well as on project level) while it remains unclear what exactly are the sequence requirements behind the sequence.

In the more and more frequent use case of OEM or 3rd party software integration, it becomes common to get a required sequence of treatments as input. But it covers the 3rd party vs. integration SW, and so does not cover the whole picture.

Anyway, even if there is a documentation as input to the integration, there remains the problem of the relevance and exactness of this documentation, on which criteria it has been established.

IV. Inadequateness of AUTOSAR

During the last years, AUTOSAR gained increasing attention in the automotive community, mainly because of its focus on supporting the assembly of the software for a control unit out of a pool of components while even permitting a seamless moving of components between different control units. However, even if AUTOSAR has grown extremely feature rich (and consequently extremely complex) it still lacks support for important aspects of the integration of the software system due to the complexity of the requirements to be taken into account at integration time.

When looking for example on the support for sequencing definition in AUTOSAR, we find that the sequence of the execution of runnables is purely seen as an integration artifact within the project context and the sequencing information is not reusable but stored in a project-wide information container. For instance, the previously mentioned basic principle of scheduler is simply not supported by AUTOSAR. Consequently, there is no formal way to set up requirements or even to establish a common idea of sequencing between contributors of different components within a system.

The AUTOSAR Timing Extensions [3] address this point for describing temporal dependencies in a formal way thus bringing a huge improvement into AUTOSAR, as they provide the possibility to document sequence requirements between different runnables within a system by using the timing constraint feature. They provide the means to define temporal relations between known items (runnables, events) in the system. However, the timing extensions do not actually aim at defining or describing the dynamic aspects of an architecture frame

for a product line or a product platform: In such a setup, the sequencing requirements have to be expressed without knowing the system in detail — especially not knowing all the components and their exported runnables.
Therefore, we are lacking the description of a common set of sequences in the system, on which the developers can rely when designing their components and e.g. are defining the breakdown into individual runnables.
In the following section we propose a concept of a dynamic backbone that complements the existing approaches by adding such an architectural frame.

V. The Phase Concept

The idea described in this section is inspired by the idea used in the boot concepts of UNIX-like operating systems [4]. To overcome the current practice of engineering the execution sequence that leads to a correctly operating control device when integrating ready-to-use components into a project, we are proposing to define in the beginning (i.e. before the component development is started) a dynamic partitioning, constructed out of several phases. It is important to note, that the needs of this dynamic partitioning are not fitting to the borders defined by the state-of-the-art feature driven static partitioning of the automotive powertrain software systems. Typically, there is the need to interleave the execution of different components within the calculation sequence of e.g. one task and consequently, one element of the static partitioning will typically export calculations to be executed within several different phases (in addition to exporting for different OS tasks).
When defining such a dynamic partitioning for the phases there needs to be defined an unambiguous sequence of execution as well as a clear description of what kinds of activities shall be performed within each phase. Of course, this definition will depend on the needs of the application domain of the control unit. Here we present an example designed for being used in an engine control unit (see Figure 7).

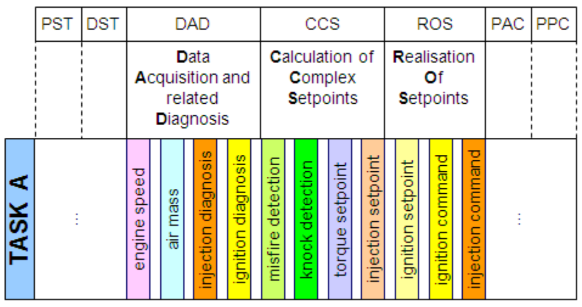

Figure 7. Simplified example of a partitioning of the dynamics. The potential content of each task is structured into a (maybe task specific) set of phases. Each phase has assigned an unique position within the task sequence and an exact description of which kinds of jobs in terms of the application domain have to be performed within it.

With these phases having been defined, the development of the components can be started. Typically, the component developer will face the situation that one component needs to perform activities which have been assigned to different phases due to the phase descriptions. Consequently, the component developer will now provide different entry points (typically function calls or runnables) in his component in order to allow the different parts of the component to run (i.e. being invoked) within the different phases.
This kind of proceeding is not entirely new in the area of automotive powertrain control units. Already today there is some agreement about e.g. the available recurrences and associated OS tasks and the component developers take this constraint into consideration when designing the components. Our approach just proposes to step to a more fine grained structuring of this dynamic framework for allowing the consideration of sequencing needs.
As already mentioned before, the definition of the phases will strongly depend on the application domain of the control unit. Also, it may be necessary to adapt the set of phases for each available control flow source (e.g. interrupt, OS task, software event) within the system which is calling a varying set of functions that are composed at integration time based on the set of components available in the system. There are multiple benefits with this approach:

- Integration is made easier: As soon as the task is chosen based on the timing constraint (recurrence, jitter, . . .), the position in the sequence is predefined by the phase. A fine tuning inside the phase can be done, if needed. Due to the number of involved functions within each phase this task is much easier than before.
- Integration is reproducible: As the phase is attached by definition to the reused runnable, the integration of the same runnable is identical on all projects.
- Integration is prepared at development time: The developer of the component selects the phase for the calculation feature. He even can base his design on the selected phase and on the phase where the information he uses as input for his calculation is produced in.
- The integration constraints of one runnable are expressed independently of the context: The most interesting aspect of the phase is that it allows to express a sequence need, by essence an extrinsic information involving different ”partners”, without any reference to any other artifact.

Different extensions of this concept are under investigation: Mapping of these dynamic design patterns to the static patterns, which are used to define the modularity of functions. This would reduce the gap between static and dynamic architectures. Another extension is to derive, from the phase of a runnable, the phase of the data which are produced by this phase. This would provide a better classification of the thousands

Concept	AUTOSAR Support
Phase concept	○
Component schedulers	○
Runnable sequence constraints	✓
Project sequence	✓

Table I
OVERVIEW OF SEQUENCE MANAGEMENT CONCEPT SUPPORT PROVIDED BY AUTOSAR.

of data handled an EMS-SW. These extensions are currently under evaluation, and would require additional description of modules, or data. Here again, AUTOSAR support would be required.

VI. Complementary Approaches

In addition to the phase concept some complementary measures are deployed. In effect, with more than 200 runnables to be integrated in a task the phase can only be a first level of integration as the number of phases will be limited. So, additional kinds of sequence constraints are used in combination with the phase concept: Sequence constraints between runnables and sequence constraints relative to data flow (producer vs. consumer of data). These constraints are like the phase assignment defined at development level and reused at integration time. They are used to fine-tune the position of the runnables inside the phase and they may be expressed using the AUTOSAR Timing Extension concepts *ExecutionOrderConstraint*, and partially *AgeConstraint* (for details, please see [3]).
As already mentioned above a complementary mechanism used to handle the sequence of runnables is the so-called "scheduler" concept. When building a composition of different modules, an encapsulation of the internal sequence is reached by using schedulers. The big benefit of this approach is that by the integrator the multiple modules are seen as one single runnable to be integrated. The integrator is only responsible for correctly setting the position of this front-end call within the task, but does not need to take care about the component internal sequence, which is defined in a reusable way at design time.
At the end, the correct integration of runnables into tasks is not only a question of sequence (based on the previously described mechanisms): in parallel, the execution time of functions, data consistency issues, schedulability aspects need also to be considered. But this could be the subject of another paper.

VII. Case study

In order to validate the approach, a real application was reworked by applying the concept. Existing runnables were analyzed and classified into the different phases, depending on the contained control algorithm. The analysis of the runnable content was performed based on functional aspects and the data flow between the runnables was not considered. However, due to the fact that a re-design of existing components according to the defined framework of phases was not possible during the case study, in some special cases data flow based placement could not be avoided. Figure 8 shows as an example the differences in calculation sequence due to the piloting for one task. After the redesign of the sequence of runnables inside the two main tasks, containing 80% of the control functions the project was successfully tested on a software test bench as well as in the car.

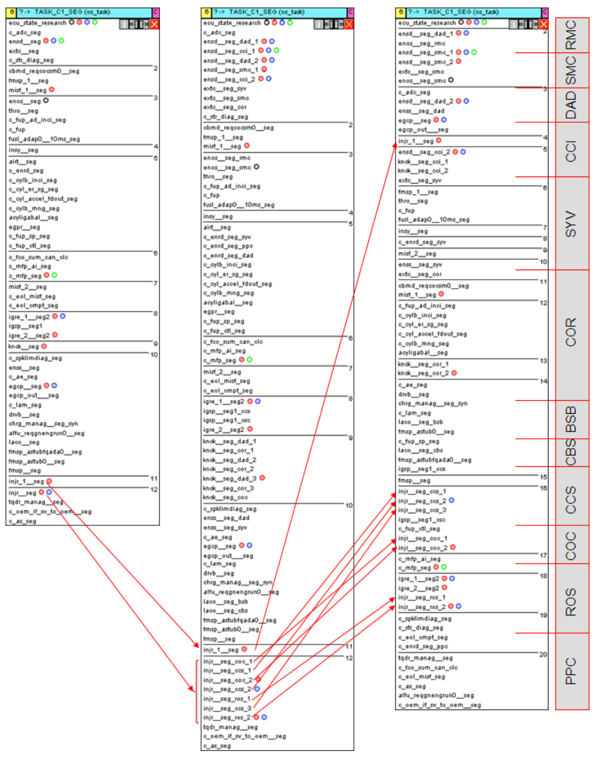

Figure 8. Changes that were applied during the case study to the sequence of calculations. The diagram shows the starting point (left), the first intermediate status, consisting of the identification of the runnable phases, but without re-sequencing, and (right) the final status. It shows also, for the particular function of injection (that we can expect to be critical), how its original placement has changed and is now distributed over the task phases.

VIII. Summary and Outlook

The high complexity and close coupling of Engine Management software makes integration of runnables into tasks a real challenge. This topic gains increasing importance due to the following recent trends:

- Increased portion of third party software.
- Increased complexity and size of the software.
- Introduction of multi-core controllers.
- Shortened development cycles.

Different mechanisms have been developed by Continental Automotive Powertrain in order keep the integration under control. The phase concept is the corner stone of our strategy, and enables us to express integration constraints without referring to a specific environment and thus allows a definition of sequences across competing suppliers of components.

The migration of our portfolio is starting now, and any new functionality will be developed according to this framework.

Concerning AUTOSAR use, the approach followed by Continental has been to first start from our original needs and situation; then, to build-up the most adequate solution independently of any standard; and finally check at the end how (and if) the standard can be applied to support the concept.

For the unsupported aspects, model transformation will be used to connect our Architectural model to the AUTOSAR standard, like to any other customer specific standard.

Our next step will be to get this concept integrated into AUTOSAR, in order to guarantee standardization, to ease discussion between parties, and to improve the availability of development tool solutions.

References

[1] D. Claraz and M. Niemetz, “Engine management software dynamic architecture versus integration,” in *ERTS 2008*, 2008.

[2] D. Claraz, K. Eppinger, and L. Berentroth, “Reuse strategy at Siemens VDO Automotive: The EMS2 Powertrain platform architecture,” in *ERTS 2004*, 2004.

[3] (2010, 11) Specification of timing extentions. AUTOSAR Consortium. [Online]. Available: http://www.autosar.org/download/R4.0/AUTOSAR_TPS_TimingExtensions.pdf

[4] A. Frisch, *Essential System Administration*, 3rd ed. O’Reilly, 8 2002.

[5] M. Deubzer, U. Margull, J. Mottok, M. Niemetz, and G. Wirrer, “Partitionierungs-Scheduling von Automotive Restricted Tasksystemen auf Multiprozessorplattformen,” *Proceedings of the Second Embedded Software Engineering Congress*, December 2009.