Integrating MBD and CBD Workflows for Automotive Control Software

V. B. Singh, Ajinkya Bhave, Dhvinay P V, Dilli Atturu
Siemens Industry Software (India) Private Limited.,
SKCL Central Square - 1, Guindy Industrial Estate,
Chennai, 600 032, India

I. ABSTRACT

The design of modern automotive software systems requires support capable of properly dealing with their ever-increasing complexity. In order to account for such a complexity, the whole software development process needs to be rethought. Many approaches have been proposed to tackle this complexity including combining model and component based development approach. This paper proposes a different approach to combine MBD and CBD approach, reviews the different approach in light of the development process and discusses the advantages of the proposed approach to reduce the software development time and complexity.

II. INTRODUCTION

The progress in the automotive market is no longer dominated by mechanical features and power specifications, but increasingly on but how efficient, safe and feature rich the vehicle is. This had resulted in - the extensive use of electronics and software as part of the traditional automobile industry. 90% of the upcoming innovations in the automotive industry are achieved with electronics and the associated control, monitoring, and diagnostics software[1]. The embedded systems deployed in a conventional vehicle can range between 35-45% of the cost of the vehicle[2]. With increase in features come increased code complexity, and potentially more space for errors. With about 75 million new vehicles expected to be sold in the year 2016[3], the economic ramifications of creating non-robust, low quality embedded systems will be enormous.

Hence the need for transition from monolithic to open and flexible systems has emerged due to problems in traditional software development, such as high development costs, inadequate support for long term maintenance and system evolution, and often unsatisfactory quality of software [5]. Model Based Development (MBD) and Component-Based Development (CBD) can be considered as two independent approach of reducing software development complexity: MBD shifts the focus of software development from source code to models in order to bring design and system reasoning closer to domain-specific concepts; whereas CBD aims to organize software into encapsulated independent components with well-defined interfaces, from which complex application can be built and incrementally enhanced.

When exploiting these development approaches, several different modelling notations and consequently several software models are involved during the development life cycle. On the one hand, effectively dealing with all the involved models and heterogeneous modelling notations that describe software systems needs to bring component-based principles at the level of the software model landscape hence supporting, e.g., the specification of model interdependencies, and their retrieval, as well as enabling interoperability between the different notations used for specifying the software. On the other hand, MBD techniques must become part of the CBD process to effectively reuse of third party software entities and their integration as well as, generally, to boost automation in development process.

A constructive interplay of CBD and MBD approach could help in handling the intricacy of modern software systems and thus reducing costs and risks by:

(i) Enabling efficient modeling and analysis of extra functional properties,
(ii) Improving reusability through the definition and implementation of components loosely coupled into assemblies,
(iii) Providing automation where applicable in the development process,
(iv) Exploiting software components encapsulation property.

Thus, software engineers can decide to realize some parts of a system programmatically in code, models or component. This concept yields flexible well-performing software that can be easily and systematically monitored, analyzed and modified.

---

[1] Automotive SW Workshop, San Diego, USA, Hans Frishkorn, January 2004.
[2] http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code
[3] http://www.statista.com/statistics/200002/international-car-sales-since-1990/

In the last decade, such integration has been recognized as extremely promising; tools and frameworks have been developed for supporting this kind of integrated development process. When exploiting interplay of MBD and CBD, clashes arises not only due to misalignments in the related terminology but also, and more importantly, due to differences in some of their basic assumptions.

The objective of this paper is to present a holistic approach to software development process which allows collaboration between MBD, CBD and legacy C code, use of different abstractions and fully developed verification and validation to increase safety and security for the developed software.

III. BACKGROUND AND RELATED WORK

Here, we give an overview of CBD and MBD and their respective development workflow using V model.

A. Model Based Development of Automotive Software Systems

Modeling and models are strongly associated with engineering domain. They are used for communicating information, for analyzing and synthesizing a system. One traditional definition of model is: "A model is a simplified representation of a system intended to enhance our ability to understand, predict and possibly control the behavior of the system". For example, consider a body sliding on the ground and affected by an external force *F.* Based on Newtonian laws and certain assumptions we can obtain a behavioral model in the form of a differential equation: *d2x/dt2=F-kdx/dt.* The resulting model can be used to predict the body's motion and for synthesizing a feedback control system, as long as the assumptions are valid with respect to the design tasks at hand.

From the engineering point of view the ability of MBD representation to look closer with the problem is what makes it advantageous to the users. Below a traditional software development V model is presented for building a system using model based development

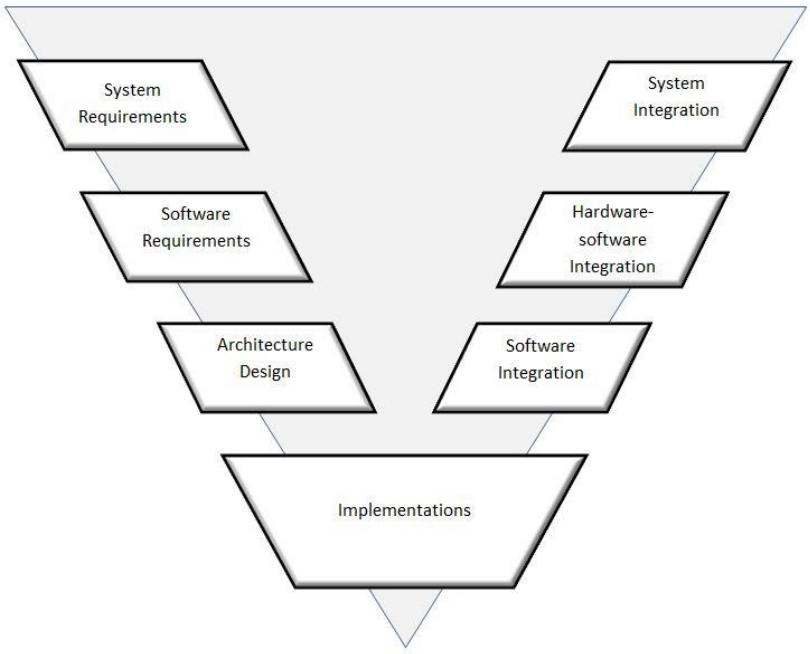

Figure 1 - A detailed V development process for MBD

B. Component Based Development of Automotive Software Systems

CBD methodology relies on the notion of component that is "*a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party*". A basic idea in component based software development is building systems from existing components as opposed to building the entire application from the scratch.

From the engineering point of view the advantages of CBD are based on standardization and reusability. Standardization plays a crucial role as it enables independent development and seamless integration of components. By reusing the same entities the confidence of their behavior and properties increases. Similar to other engineering domains, CBD aims for targeting complexity: By reusing existing solution not only on the component level but also on the system structure level CBD enables better understanding of complexity; the implementation details of components are hidden and only component services that are exposed through component interfaces are visible. In this way the abstraction level is increased which is a key factor in managing complexity. Further, in the CBD approach the maintenance is focused on replacement similar to replacement of spare parts of components rather than on re implementation of specific parts.

Below a traditional software development V model is presented for building a system using component based development

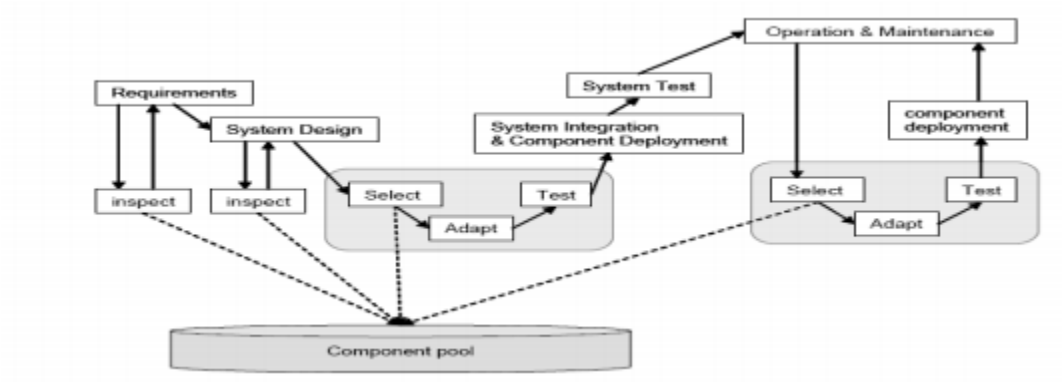

Figure 2 - A detailed V development process for CBD [1]

A challenge related to a component specification is the verification; How to verify that a component will behave correctly in a new environment? The example of Ariane 5 illustrates the case in which a component has been reused in a new environment, resulting in a major system failure. The question of verification is related to testing and formal verification. In both cases it is a challenge – what specification is needed and which methods can be applied.

IV. PROPOSED MODELING CONCEPTS

In this section, we explain how software is developed in Embedded Software Designer[4] from top down and bottom up. While discussing top down approach we start from model based approach and combine it with the component. We are mainly going to talk about paper objective with regard to top down approach. We also talk briefly about the bottom up development in Embedded Software Designer.

A. Top Down – From an architecture model to a Software Component

In the top down workflow system engineers can develop component/block architecture design by providing interfaces descriptions, data types, contracts etc. Starting from this skeleton model, design engineers can develop controller model using the existing blocks or custom subsystems in Embedded Software Designer and use these blocks inside runnables of the software component.

One important thing to note here about this approach is that the blocks are not converted to their respective components, rather – they are directly used as runnables in the top level component. The workflow of this integration is presented in the Figure 3.

[4] LMS Imagine.Lab Embedded Software Designer is a tool developed by Siemens capable of modeling embedded controls for mechatronic products, using Domain Specific Languages. The tool is built on top of mbeddr [7]: an extensible set of integrated languages for embedded software development. [2] presents a use case developed in Embedded Software Designer.

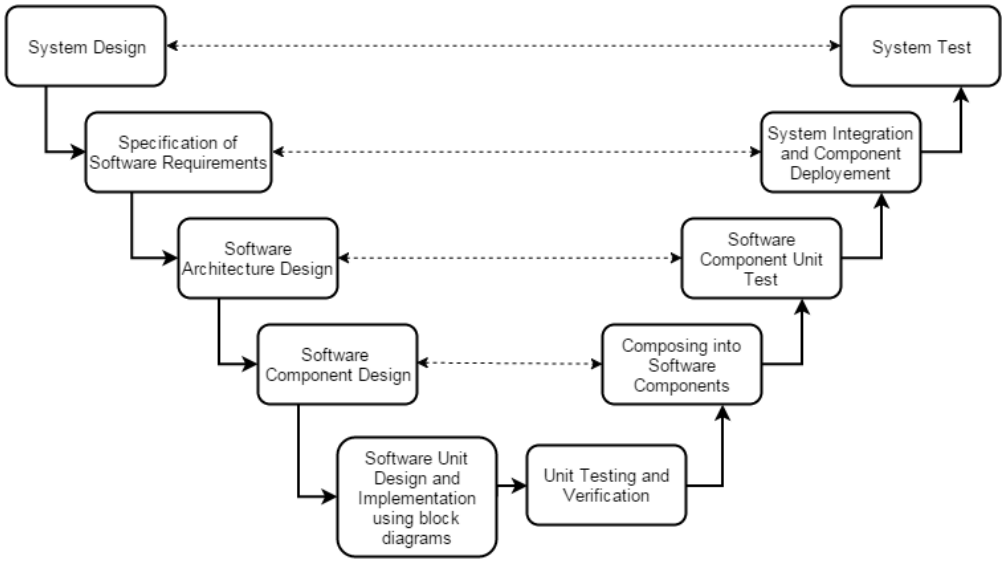

Figure 3 - A typical workflow proposed to combine MBD and CBD

This requires the following process:

1. A separate Client-Server interface is created for the operations where block can be used. Same operations can be used for multiple blocks together. If following top down approach these interfaces will already be available so they can be used in the components.

```
exported cs interface compute_ssd_mode {
  void compute_ssd_mode_function(double/m/ d_safe, double/m/ d_lead)
    pre(0) d_safe < 1000 m && d_lead > 0 m
}
```

Figure 4 - Shows a Client-Server interface having one operation - Note the use of physical units and pre-condition in the development

2. For this approach we needed the support to make use of a Data Flow block inside the runnable – Hence we came up with the approach where a data flow block can be used a type (<Block Type> not to be confused with C data type).

```
block<                > <no name>;
      ssd_mode_compute contents (Components_Refactoring.ACC_SSD_ATOMIC)
```

Figure 5 - Picture shows a data flow block ssd_mode_compute being used as a block type

3. Another type named <block result> is introduced, which stores the block's output as structure. This block result can be used to pass data from the block to component interfaces.

```
blockresult<                > <no name>;
            ssd_mode_compute contents (Components_Refactoring.ACC_SSD_ATOMIC)
            Abs                       contents (ClaStdLib.Cla_Functions)
```

Figure 6 - Shows the block result type - this can be used to trigger the execution or terminate the instantiated block

```
blockresult<ssd_mode_compute> Data_Flow_Block_result = Data_Flow_Block.  ;
                                                        execute()   (BaseConcept in c.l.a.blocks.c)
                                                        terminate() (BaseConcept in c.l.a.blocks.c)
```

Figure 7 - A data flow block can be executed or terminated (In case the block uses states) - and the result can be assigned to the same block result type

```
blockresult<ssd_mode_compute> Data_Flow_Block_result = Data_Flow_Block.execute(d_lead = <value>, d_safe = <value>);
```

Figure 8 - The values of the input ports of the data flow block can be provided using interfaces created in step 1 or using local variables in components or a source block which has to be again instantiated inside the runnable

This has several advantages –

(i) Lots of MBD advantages can be leveraged when algorithm are modeled using blocks,
(ii) the model thus developed could be taken to Simulink/Amesim to perform closed loop simulation with the plant,
(iii) If any further changes are required, it can easily be done by simply modifying the algorithm in blocks and without touching the component part. Needless to mention the user has full flexibility between choosing pure MBD or legacy development approach

B. Bottom-Up: Reusing Legacy Models and Legacy C code in the Development process

Because of extensive early use of legacy C files and since Model-Based Design is a well-established development process in the automotive industry, companies usually have an extensive library of mature and extensively tested C files and models. It is important that these models can be reused to reduce the development costs and time.

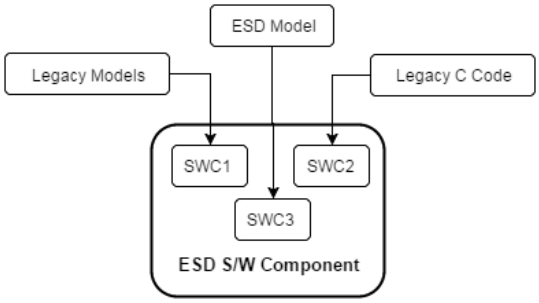

Figure 9 – Different implementation approach can be used and combined with components in Embedded Software Designer

These legacy C-files and Simulink models can be imported into Embedded Software Designer and used in the further development process. Hence, the bottom-up workflow requires the component interfaces to be configured to ensure that these imported models can be imported properly.

V. USE CASE

In this Section we show the application of our modeling concepts as presented on an exemplary use case from the automotive domain. First we give an overview of the example and system. After that, we apply the above discussed modeling approach together. Finally, we discuss the use case presented in top down development workflow.

A. Example – Adaptive Cruise Control (ACC)

ACC is a convenience feature that enables driver to relax by adopting the vehicle cruise control to traffic conditions automatically without driver's intervention. ACC system composes of a sensor system that senses the traffic conditions. Based on the traffic conditions, ACC system commands the vehicle to move either at the set cruise speed or at a speed that is safe enough to avoid collision with the vehicles in the front. When there are no vehicles in the front, ACC commands the vehicle to cruise at the set cruise speed. If there is a leading vehicle in the front that is slowing, ACC system slowly reduces the vehicle speed to maintain a safe distance from the leading vehicle to avoid collision. As soon as the leading vehicle starts accelerating and moves away, ACC accelerates the vehicle back to its set cruise speed. ACC system controls the vehicle speed by manipulating the throttle and brake[2].

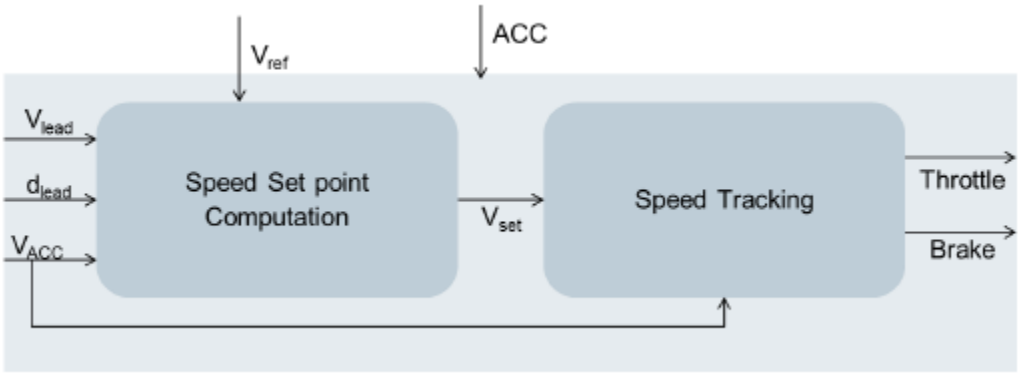

Figure 10 - A top level architecture design for Adaptive Cruise Control Algorithm [2]

B. System Design

One of the first steps is to develop the system architecture. The architecture is the topology of the system and describes the structural hierarchy of the subsystems and their interfaces and connections. Partitioning of the control algorithm is normally done either based on task scheduling or based on features. Since it is assumed that the ACC algorithm will be executing at a single task rate, control algorithm is architected into several features.

Functionalities are identified and individual block interfaces are created for them – these block interfaces contain the specification for that block. Specifications include the block skeleton (no of ports, parameters including types and units). Contract based design [6] can be used for both the architecture and implementation.

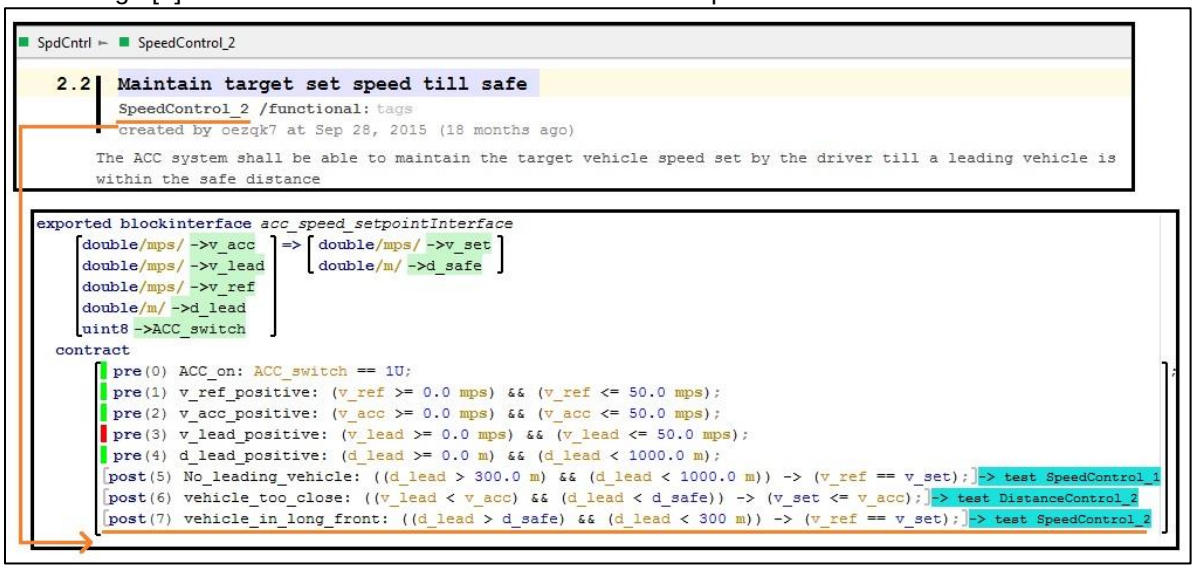

Figure 11 - The block interface acts as a container for the real subsystem implementation and can be linked to requirements created in Embedded Software Designer. Also note the use of contracts and physical units in the block interface.

Using these blocks interfaces – a complete system design can be built. Figure 12 shows the architecture modeled in Embedded Software Designer using Block interfaces and abstract composite blocks.

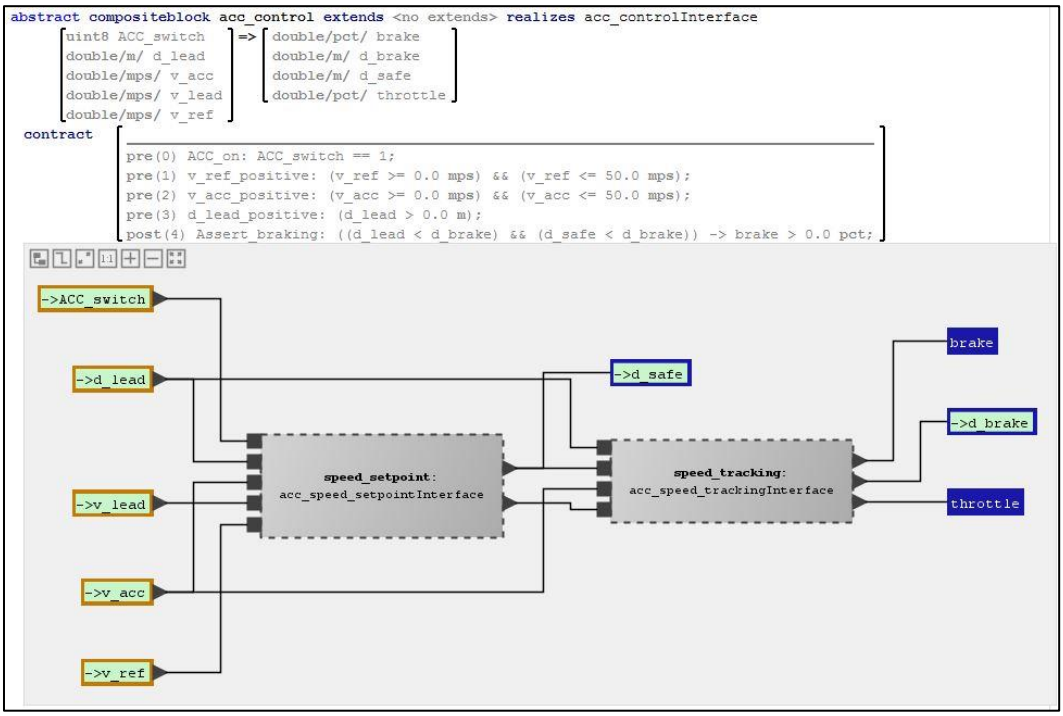

Figure 12 - A system architecture for the Adaptive Cruise Control system designed as an Abstract Composite block in Embedded Software Designer

From a workflow point of view, these system designs can be verified with respect to port compatibility, uniform data exchange between blocks using assume/guarantee analysis feature of Embedded Software Designer.

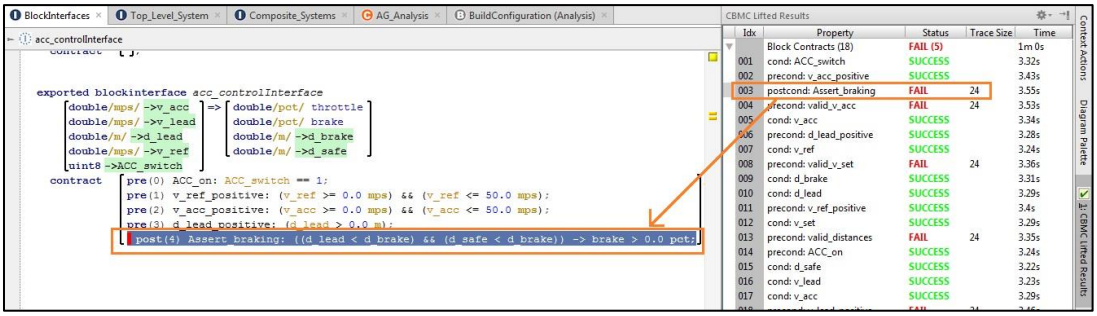

Figure 13 - Shows the contract based verification of an architecture created in Embedded Software Designer

In component based development approach, Interfaces such as Client-Server or Sender-Receiver can also be specified as part of the design process. Contracts can be specified for component specific interfaces. Figure 14 shows individual Client-Server interface with contracts associated with them.

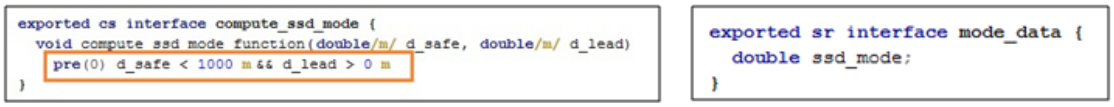

Figure 14- A client server interface (left) and a sender receiver interface created in Embedded Software Designer

Next section describes the development of a control algorithm used in ACC using MBD and refactored in components.

c. Development using MBD and CBD

In this section we will try to apply the above discussed approach for integrating data flow blocks and components. We will take part of ACC system, model it using Data Flow Blocks and integrate it with a larger component.

For implementing the algorithm, a subsystem can be designed by extending the block interfaces similar to Figure 11. User can use already existing library blocks or create their custom blocks using C code. Using library blocks, the algorithm can be modeled in drag and drop fashion.

Already existing algorithm in legacy C code and legacy models can also be integrated with the workflow as discussed in section IV.B.

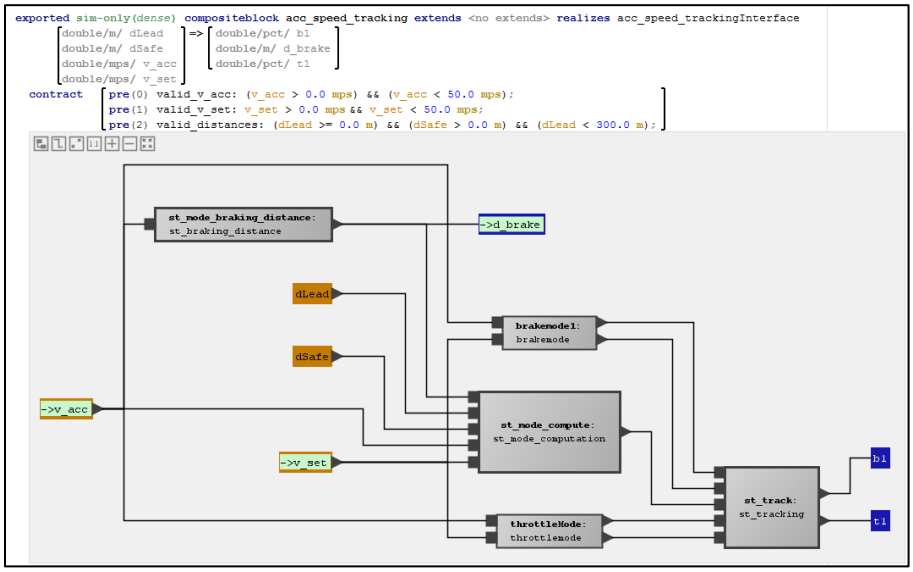

Figure 15 - Figure on the left shows a composite Block implementation in graphical view. Completed controller subsystem can also be simulated in closed loop with the plant model in tools like Amesim/Simulink

After the block is tested and formally verified (discussed in forthcoming section **VERIFICATION AND VALIDATION**), the approach of integrating this block with component is followed. For this purpose a component is created which provides the CS interface *compute_ssd_mode* and requires a Sender-Receiver interface *mode_data* to write the output value.

```
exported component ssd_mode_compute_refactored extends nothing {

  provides compute_ssd_mode provide_compute_ssd_mode

  requires mode_data mode_data

  void provide_compute_ssd_mode_compute_data_flow_block_in_component(double/m/ d_safe, double/m/ d_lead)
      <= op provide_compute_ssd_mode.compute_data_flow_block_in_component {

  } runnable provide_compute_ssd_mode_compute_data_flow_block_in_component

} component ssd_mode_compute_refactored
```

Figure 16 - The component has one client-server and one sender-receiver interface. The runnable implements the operation defined in client-server interface.

To use the data flow block in the runnable of a component, we follow the steps as shown in images from Figures 5 – 8.

```
exported component ssd_mode_compute_refactored extends nothing {

  provides compute_ssd_mode provide_compute_ssd_mode

  requires mode_data mode_data

  void provide_compute_ssd_mode_compute_data_flow_block_in_component(double/m/ d_safe, double/m/ d_lead)
      <= op provide_compute_ssd_mode.compute_data_flow_block_in_component {

    block<ssd_mode_compute> mode_compute;
    blockresult<ssd_mode_compute> ssd_mode_compute_result = mode_compute.execute(d_lead = d_lead, d_safe = d_safe);

    mode_data.ssd_mode = ssd_mode_compute_result.ssd_mode;

  } runnable provide_compute_ssd_mode_compute_data_flow_block_in_component

} component ssd_mode_compute_refactored
```

Figure 17 - Component with the block integrated.

D. Discussion of the Use Case

We have shown how our approach of modeling blocks and components and integrating MBD and CBD can be used as presented on a simplified use case. It is imaginable that this example can be further advanced to use in a larger system level applications and model extra-functional properties in more detail.

In the next section we discuss about different aspects of tests and verification in Embedded Software Designer.

VI. VERIFICATION AND VALIDATION

Verification and Validation (V&V) are essential stages in the MBD and CBD workflows of automotive controllers to remove the gap between the design and implementation. Several approaches can be combined to make the implementation meet the safety requirements. In this section we discuss how an implementation developed using MBD and CBD can be tested. We also present the use of formal verification in this regard.

A. Testing Models Implementation

Test cases for a block interface can be created at the design time. After implementing the block interfaces, each implementation can be unit tested with respect to the test case created.

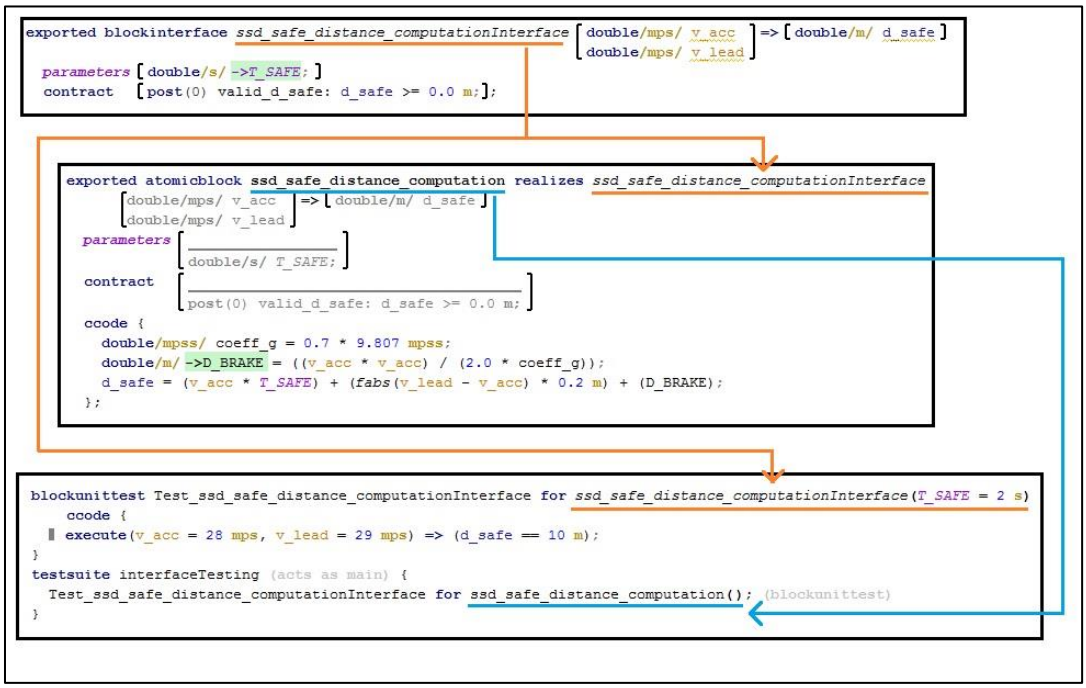

Figure 18 - It is important to note how interfaces, implementation, tests and testsuite are tightly linked with each other and can be traced easily

B. Component Verification

A component can be tested with the help of mock component and test case. A mock component is used to provide explicit and sequence of test data for the component under test.

```
mock component check_validity report messages: true {
  provides mode_data mode_data
}
```

Figure 19 - Mock component using an SR interface as a provider port

In this case a mock component is used to hold the output data from the above component. In this case the mock component is very simple, acting just as a data holder. However in a complex test scenario, sequence of steps can defined in mock component and is used in calling the component for every step.

After we have defined the mock component we need to connect the component under test with the mock and create adapter for the provided port of the component.

Figure 20 - Components connections made in textual and graphical view. In graphical view instances are created using drag and drop method.

An adapter makes a provided port of a component instance available in a test case which can be used to provide test data and to verify the output results.

```
exported testcase test_block_in_component {
  initialize connection;
  data_input.compute_data_flow_block_in_component(12 m, -5 m);
  assert-equals(0) block_algorithm_calc.ssd_mode == 5;
} test_block_in_component(test case)
```

C. Formal Verification

As systems become more complex, both in architecture and dynamics, exhaustively testing each property does not scale up economically and with satisfactory coverage to uncover all bugs. Because of this fundamental drawback, automated formal verification has emerged as a promising useful complement for the automotive industry. The roots of formal methods date back to the early days of computer science, including the works of Floyd [3] and Hoare [4]. The core idea is to devise automatic proofs at compile-time to verify runtime properties of the program under analysis. Because computing power has increased continuously in the last decade, formal methods have started becoming applicable to large scale industrial problems.

One of the problems is how to specify which runtime properties of the program are to be verified. All formal methods are also limited by the decidability problem. This is due to the fact that it is not possible for a finite memory, discrete device like a computer to comprehensively prove non-trivial properties of the extremely large (theoretically infinite) runtime behaviors of computer programs. In addition, in many cases, it is impractical or impossible for computers to solve decidable questions within a specified amount of time and memory for large input sequences, such as the behavior of control programs observed over long time periods.

Three main approaches have been considered for formal verification. All of them are approximations of the program semantics (formally defining the possible executions in all possible environments) due to the complexity of modern-day programs:

a. Deductive methods produce mathematical correctness proofs using theorem provers or proof assistants. The main limitation is that they need a human in the loop to simplify the induction arguments and to guide the proof strategy by supplying hints at different stages. This is not scalable for complex automotive software that is continuously being modified over a product's lifecycle;
b. Model-checking exhaustively explores finite-state models of program executions. The limitations are that increasing model complexity can lead to combinatorial explosion, and the approach requires a human to create the models;
c. Static analysis automates the abstraction of the program execution and always terminates. The limitation is that the approach can be subject to false alarms. This occurs when the analysis warns that the specification may not be satisfied although no actual execution of the program can violate this specification.

Software Implementation can be verified using formal methods. This helps in identifying issues such as divide by zero, array out of bounds, memory leak, pointer dereference, assertions check etc. In case of statemachine it is used to check the reachability of states and whether all transitions can be fired or not.

For a dynamic system where time varies and affects variables used in implementation, simulation and step time can be set to check robustness issues.

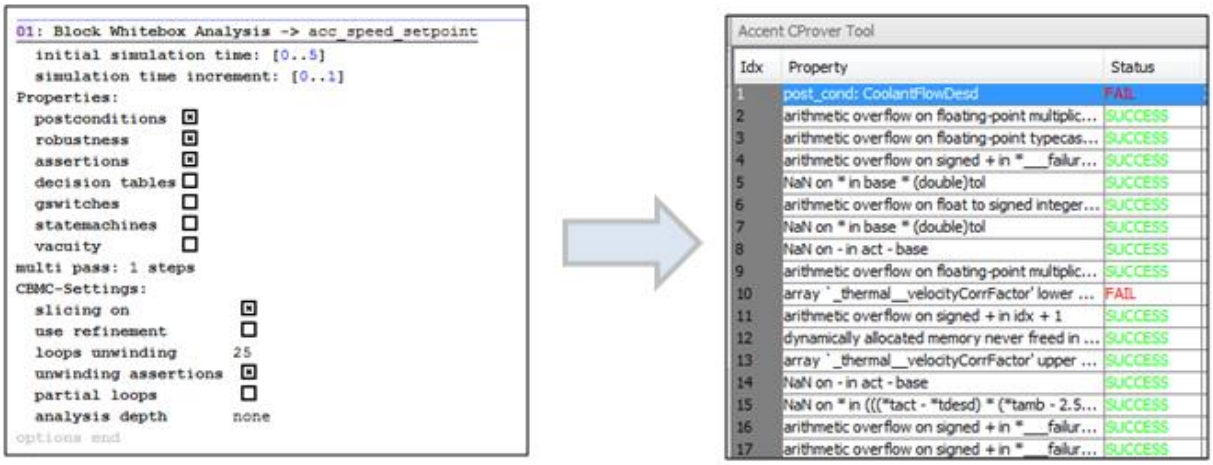

Figure 21 - The implementation can also be verified using formal methods. This helps in identifying the robustness issues, assertions check etc.

A component can also be formally verified in the same way as blocks are verified for robustness properties.

VII. CONCLUSION

In this paper, we presented concepts for modeling and integrating blocks and components together and showed in a use case how these concepts can be applied. We introduced the concept of declaring a block as a type and storing the result using another type called <block result>.

The vision is to have a generic modeling framework where MBD and CBD approaches can be combined and provide the user the best of both the world. The discussed approach can also be used holistically with the legacy C code or legacy models.

Concerning our future work, we are currently working on partitioning the components based on sample time. This will enable to model complex time and event dependent system. We are also working on developing a framework where the complete system thus modeled using components can be taken to Amesim and closed loop simulation can be performed with the plant. Currently this is only possible if system is developed using data flow blocks.

VIII. REFERENCES

[1] Crnkovic, I., Larsson, S., & Chaudron, M. (2012, 05 10). *Component-based Development Process and Component Lifecycle*. Retrieved 04 20, 2016, from http://www.mrtc.mdh.se/publications/0953.pdf

[2] Mori, H., Zhao, Q., Yokojima, Y., Akella, S., Sundaresan, S.A., Dhvinay, P.V. (May 2015). *An Advanced Approach for the Design of the Adaptive Cruise Control*. 4th Controls, Measurement and Calibration Congress. Pune.

[3] Floyd, R.W., *Assigning meaning to programs*. Proc. Symposium in Applied Mathematics, vol. 19, pages 19-32. AMS 1967

[4] Hoare, C. A. R., *An axiomatic basis for computer programming*. Comm. ACM 12(10): 576-580, October 1969

[5] Bosch, J., *Design and Use of Software Architectures*. ACM Press in collaboration with Addison-Wesley, 2000

[6] Rajan, A., and Wahl, T., Eds., *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*. Vienna: Springer Vienna, 2013

[7] http://mbeddr.com/