# A Simulation Tool Chain for Investigating Future V2X-based Automotive E/E Architectures

Christoph Roth*, Harald Bucher*, Alisson Brito+, Oliver Sander*, Juergen Becker*
Karlsruhe Institute of Technology (KIT)*
Federal University of Paraiba (UFPB)+
Email: {christoph.roth, harald.bucher}@kit.edu

***Keywords—Co-Simulation, Vehicle-to-X, E/E-Architecture***

## I. Introduction

Today, the electric/electronic (E/E) architecture of modern cars is a distributed network of embedded systems, consisting of several bus systems, dozens of electronic control units (ECUs) and hundreds of sensors and actuators. To this system, high requirements regarding determinism are imposed which form the basis for safe operation in any conceivable driving situation. Because of that, the timing behavior of single components as well as communication within the vehicle is widely statically determined a priori and the E/E architecture is realized as a nearly fully closed system.

Due to the evermore rising number of functions, current E/E architectures are more and more a vulnerable source for faults and a barrier to innovation [1]. This situation is aggravated by the integration of new technologies like Vehicle-to-X Communication (V2XC) which form the basis for a large number of future services and applications. For these, it is no longer sufficient to only consider data that is available locally. The majority of applications rather will rely on information that originates from most diverse data sources. E/E architectures in the first approximation need to be ”opened” by means of an additional radio interface located e.g. at an ECU, which enables communication with other vehicles or infrastructure. This not least increases potential for non-deterministic disturbance of safety-critical functions or malicious attacks of the internal communication network [2].

In order to overcome the limitations of current E/E architectures, application of new design principles like more encapsulation, standardization and centralization as well as a fundamental reconsideration of the fragmented development process is necessary [1]. Up to the present day, both, function and architecture development are most often separately running tasks. As a result, errors are often only discovered in the integration phase. This fact can become an extremely time consuming and expensive endavor. Principles of platform-based design (PBD) are a promising solution in order to cope with this problem [3] since function and architecture are separated. This enables a flexible mapping of function to architecture and its validation already in early phases of development. The result is an increase of reliability and a reduction of development cost by avoiding additional design cycles [3]. Within this context, we propose a novel extensible tool chain that targets facilitation of exploration, validation and verification of future V2X-based automotive E/E architectures. Such systems are heterogeneous by nature. Hence, a design framework is necessary that supports managing heterogeneous model composition for representing data as well as control flow between models. The proposed tool chain is made up of a heterogeneous design tool called *Ptolemy II* (PtII) [4] and a simulation middleware based on the *High Level Architecture (HLA)* [5]. A possible framework architecture that can be developed by the tool chain is illustrated in Fig. 1.

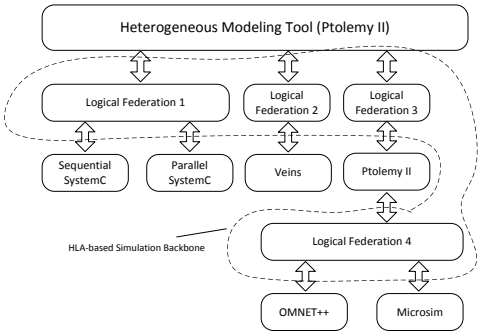


Fig. 1. Possible Framework Architecture

The HLA enables distributed co-simulation with domain-specific simulators like (parallel) SystemC [6][7] (hardware/-software), OMNET++ [8] (network), Veins/SUMO [9], Microsim [10] (traffic) or co-emulation with real hardware/software components [11]. PtII is the starting point of the design process. Its task within the tool chain is twofold: I) The capabilities for explicit meta-modeling using an abstract syntax of clustered graphs [12] serve as user interface for support of tool integration and configuration of control and data flow interaction via HLA, II) due to its inherent heterogeneity and model composition properties PtII serves as central design tool for performing architectural exploration and validation and verification.

## II. Fundamentals

In PtII, the basic building block of a system description is an *actor*. Actors are concurrent components that communicate through *ports* and *relations*. They can be *atomic* or *composite*. An atomic actor is at the bottom of the hierarchy. A composite actor allows hierarchical nesting of actors. Both, atomic and composite actors are executable following specific execution semantics, also known as *models of computation* (MoC) [4]. The MoC within a composite actor is determined by a *director*. In the *discrete event* (DE) MoC, interaction between actors is modeled by *events*, representing some instantaneous action

during simulation time. DE simulations are particularly suitable for modeling discrete systems like e.g. digital hardware or communication networks. SystemC, OMNET++, SUMO or Microsim are examples of DE simulators. Beside DE there exist various other MoCs like continuous time (CT) which is suitable for modeling analogue components like sensors, or process networks (PN) which are often used for modeling data flow applications.

The HLA is an IEEE standard [5] since 2000. It was originally defined by the Defense Modeling and Simulation Office (DMSO) for the U.S. Department of Defense. Its original field of application are military training simulations. The HLA is a generic software architecture combining all the components necessary for *Parallel Discrete Event Simulation* (PDES). In HLA terminology the logical representation of an interconnection of different simulators is called a *federation* and includes multiple simulators called *federates*. Federates connect via *ambassadors* to a *runtime infrastructure* (RTI). The RTI implements services defined by the HLA standard like time management or data distribution management. A RTI can possibly run several independent logical federations in parallel. Also part of the HLA standard is the so called *Object Model Template* (OMT) which defines the format and syntax of HLA object models including object/interaction classes attributes, parameters and datatypes but not their content. The OMT allows to define Federation Object Models (FOM) and Simulation Object Models (SOM). The FOM contains properties of a whole federation. The SOM contains properties of a single federate.The HLA implementation used in this work is [13].

## III. Simulation Platform Library

In order to equip PtII with HLA interface configuration and generation capabilities, a new version of the so called *Simulation Platform Library* (SPLib) [11] is used. The new version enables model-based federation development and is incorporated by PtII for construction kit like composition of distributed simulation tools. Therefore, a set of classes is provided that abstract from the HLA ambassador interfaces and that allows to define a contract between HLA and simulation tools in terms of data flow and interaction behavior. Fig. 2 shows an extract of the newly developed SPLib classes.

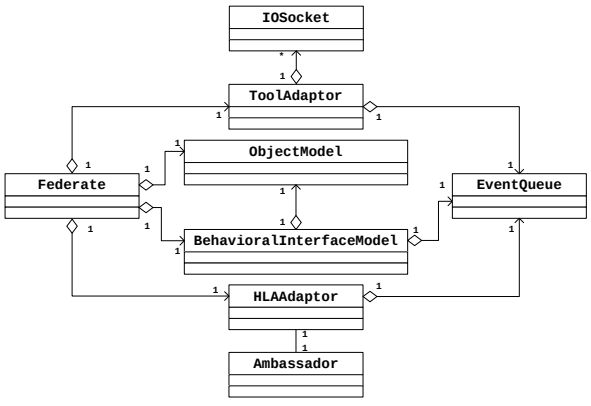


Fig. 2. SPLib Class Diagram

The `Federate` class aggregates all other core classes. The `ToolAdaptor` class establishes a link from a simulation tool to the data model and the behavioral metamodels of the SPLib. These are given by the `ObjectModel` (OM) and the `BehavioralInterfaceModel` (BIM) classes. The `ToolAdaptor` provides high-level control flow related methods that are to be called by the simulation tool. Beside that, it encapsulates instances of `IOSocket` which are used for establishing data flow connections to the simulation tool. `IOSockets` are equipped with a queue that allows buffering incoming HLA reflections. On the reverse side, the `Ambassador` class provides access to the RTI ambassador interfaces and implements the federate ambassador callback methods. Hence, it is the access point for data as well as control flow interaction with the RTI. The HLA specific interfaces given by the `Ambassador` class are encapsulated by the `HLAAdaptor` class.

### A. Object Model

The `ObjectModel` allows to store the SOM of a specific simulator. It provides capabilities for dynamic object model representation resulting in a high amount of flexibility. An excerpt of the overall class structure is illustrated in Fig. 3. The classes form an object-oriented meta-representation of a SOM. The structure has been developed according to the OMT. In the OMT specification, components are structured in tables. Object/interaction classes within the OMT can be hierarchically nested, representing inheritance. Components within one table can reference components within other tables. For instance, an attribute entry in the attribute table must reference a specific object class it belongs to. In addition, an attribute must be assigned a datatype of one of the datatype tables. Each datatype table contains types that share common characteristics. The OMT standard differntiates e.g. between basic data representations, simple datatypes or the array datatype table. More complex datatypes like arrays may reference to simple datatypes as element types.

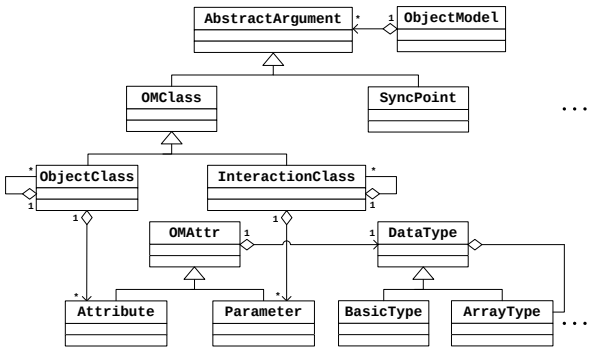


Fig. 3. Object Model Class Diagram

The nesting and referencing relationships between components in the OMT tables are mapped to object-oriented aggregation and/or inheritance relationships within the structure of Fig. 3. Generally, the `ObjectModel` class consists of `ObjectClass` and `InteractionClass` members. The hierarchical nesting between object and interaction classes is modelled using self-aggregation. Also, attribute and parameter ownership is modelled using aggregation between `Object/InteractionClass` types and `Attribute/Parameter` types. The latter two are owner of

any of the defined datatypes. For datatype representation inheritance has been applied in order to exploit polymorphism in case of compound datatypes like arrays. In the current implementation SOM tables can be specified in a easy readable form by means of (nested) C++ method calls as shown in Listing 1. Therefore, the `ObjectModel` class needs to be inherited for implementation of the `generateObjectModel()` method. Methods applied within `generateObjectModel()` are implemented in the base class. An alternative is to provide the SOM as XML representation.

Listing 1. OM Instantiation Code

```
virtual void generateObjectModel()
{
 //Synchronization table
 SYNCPOINT("SyncPoint","NA", "Register/Achieve","...");
 ...
 //Basic data representation table
 BASICTYPE("HLAinteger16BE","16","...","Big","...");
 ...
 //Simple datatype table
 SIMPLETYPE("HLAASCIIchar","HLAoctet","NA","NA", "NA","...");
 ...
 //Array datatype table
 ARRAYTYPE("HLAASCIIstring","HLAASCIIchar","Dynamic","HLAvariableArray","...");
 ...
 //Object class structure table
 OBJECTCLASS("ObjectRoot","N",
         OBJECTCLASSL("ObjClass1","PS"),
         OBJECTCLASS("ObjClass2","PS",
                 OBJECTCLASSL("ObjClass3","PS"))
 );
 //Attribute table
 OBJECT("ObjectRoot.ObjClass1",
         ATTRIBUTE("attr1","HLAASCIIchar","Conditional","UpdateCondition","DA","PS",
         "HLAReliable","TimeStamp"),
         ... );
 ...
}
```

### B. Behavioral Interface Model

By inheriting from the BIM class (see Fig. 4) it is possible to define a finite state machine (FSM) by which data and control flow interactions between simulator and HLA and vice versa can be controlled. The FSM defines a valid calling sequence of simulator interface and ambassador methods. This allows to realize/verify different general interaction behaviors and/or synchronization schemes on top of the HLA services. The latter can be advantageous for accelerating DE co-simulation/emulation [11]. Also, for MoCs different from DE other synchronization schemes may be more suitable. BIM descriptions can be annotated to respective actors as C++ code. When neglecting hierarchical states and abstract arguments, BIM FSMs can mathematically be described as a tuple $(S, \Sigma, \Omega, \delta, s_0, s_e, S_{int})$, with $S$ being the set of states, $\Sigma$ the input alphabet, $\Omega$ the output alphabet, $\delta : S \times \Sigma \rightarrow S \times \Omega$ the transition function, $s_0$ the start state, $s_e$ the end state and $S_{int}$ the set of *interaction states*.

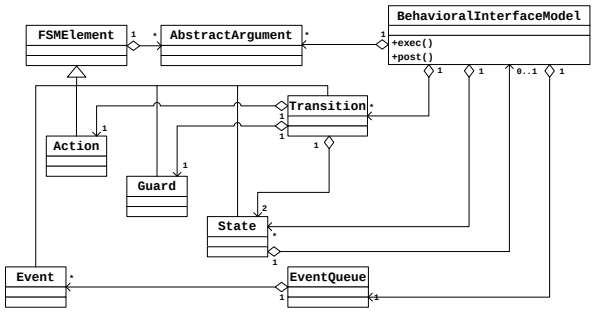


Fig. 4. Behavioral Interface Model Class Diagram

States are realized by the `State` class. During execution, the input within a certain state $s$ is given by the topmost `Event` within the `EventQueue`. Events are either generated by actions or incoming method calls from the `Ambassador` or the simulation tool respectively `IOSocket`. In the latter case, method calls are converted by the respective adaptor into events. The transition function relates inputs and source states to outputs and target states. In the `BehavioralInterfaceModel` class it is realized by a transition table which holds elements of type `Transition`. The traversal of transitions must be protected by `Guard`s. Outputs are generated by `Action`(s) which can be annotated to transitions. An output corresponds to the generation of a new event into the `EventQueue`, the call of a ToolAdaptor/HLAAdaptor method or the write access to elements of type `AbstractArgument` which are part of a FSM description.

The notion of *interaction state* as it is introduced within this work becomes more clear when considering the execution semantics of a BIM FSM in more detail. These are based on iterative execution of the two methods `exec()` and `post()` which are inspired by the `fire()` and `postfire()` methods of Ptolemy II actors (since the FSM is always ready to fire, an equivalent to `prefire()` is not necessary). An iteration consists of a single call to `exec()` and a subsequent single call to `post()`. Therewith, the following happpens:

- The `exec()` method takes the front event from the `EventQueue` and selects the transition for which the guard evaluates to true. This transition is called *active transition*. Thereby, a runtime check is performed in order to ensure that there exists only one active transition. If the runtime check is passed, the associated action(s) is/are executed. If the exist more than one or no active transitions, an exception is thrown.
- The `post()` method changes the current state to the target state of the chosen transition. The method either returns $0$ oder $+1$. If the target state is the toplevel end state or an interaction state the method returns $0$ in order to signal termination or that control should temporally be passed to the environment. For all other states the method returns $+1$ in order to signal that a further iteration of `exec()` and `post()` should be performed.

Therewith, an interaction state forms a kind of label at which the execution of the BIM FSM is suspended and control is given to the surrounding execution environment (i.e. the simulator to be connected). The state in which the FSM has been suspended last, corresponds to the entry point for further execution. Instantiation of a BIM FSM works similar to the instantiation of the object model, namely by using (nested) C++ function calls within the `generateBIM()` method. Applied methods within `generateBIM()` like `STATE()` or `TRANSITION()` are again part of the `BehavioralInterfaceModel` base class. Listing 2 illustrates an example. At the top instantiation of an abstract argument, specifically a synchronization point named "'READY_TO_RUN"' is exemplarily shown. Afterwards, four states `S_INIT`, `S_GRANT`, `S_ADVANCE` and `S_END` are instantiated within the state table. `S_INIT` and `S_END` are marked as start respectively end states. In order to avoid state explosion, BIM FSMs can be defined hierarchically: `S_INIT` is a hierarchical state that is refined with a complete state machine called `Init_FSM`. Thereby, refinements are always executed first. Only if a refinement cannot

execute, then outer transitions are evaluated. The S_GRANT state is marked as interaction state. Both, S_GRANT and S_ADVANCE are referenced by the subsequent transition definition, where S_GRANT is the source state and S_ADVANCE the destination state of the transition. Traversal of the transition is protected by an event guard (a derived type of the Guard class) which limits passage of the transition to the occurence of the *Sim_Event_SetNextBarrier* event. The passage of the transition is coupled with the execution of an action called *HLA13_A5_11_Action_NextEventReq*.

Listing 2. BIM FSM Instantiation Code

```
virtual void generateBIM()
{
 //Argument table
 ARG_SYNCPOINT("ARG_SP","READY_TO_RUN");
 ...
 //State table
 STATE(S_INIT);
 STATE("S_GRANT");
 STATE("S_ADVANCE");
 STATE(S_END);
 ...
 //special states
 START_STATE("S_INIT");
 INTERACTION_STATE("S_GRANT");
 END_STATE("S_END");
 REFINED_STATE(STATEREF(S_INIT),FSM("Init_FSM"));
 ...
 //Transition table
 TRANSITION(
        STATEREF("S_GRANT"),
        STATEREF("S_ADVANCE"),
        ACTION("HLA13_A5_11_Action_NextEventReq"),
        EG("Sim_Event_SetNextBarrier")
 ),
 ...
}
```

## IV. Tool Integration

Tool integration means, making simulation models or tool capable of communicating with the rest of the overall simulation. The process of tool integration can be separated into the three steps *Federation Model Definition*, *Interface Generation* and *Interface Integration*. The proposed overall supporting process is illustrated in Fig. 5. In the first step one or several *federation models* (FMs) are defined in PtII. A single PtII model can contain several FMs, each representing a separate federation. In the second step, interfaces for each tool are generated automatically based on the FM definition(s). Finally, interfaces are integrated into simulation tools. Depending on the tool, this step can occur automatically or must occur manually.

### A. Federation Model Definition

An FM specified in PtII syntax includes all the necessary information for generating interfaces and configuration files. This information is extracted by parsing the FM. Generally, a FM is specified within a PtII HlaComposite actor. Beside the HLAComposite actor, further different novel types of PtII actors like HLAComposite, HLAFederate, HLAObjectClass or HLAInteractionClass are introduced. HLAFederate actors represent simulators to be co-simulated, HLAObjectClass or HLAInteractionClass actors represent HLA object respectively interaction classes (in the following only HLAObjectClass actors are mentioned for simplicity reasons). By means of these basic elements the user can define I) data flows and data representations within a distributed co-simulation using actors, ports and relations. This includes datatypes, object/interaction classes and publish/subscribe relationships. II) control flow between

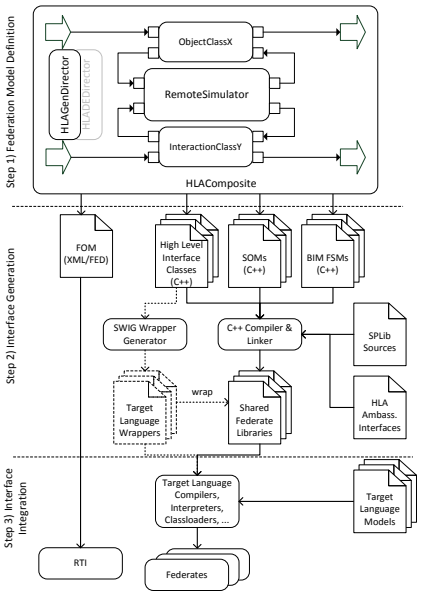


Fig. 5. Interface Generation and Tool Integration

simulators by means of annotating behavioral descriptions to actors.

*1) Definition of Data Flows and Datatypes:* Data flows between the mentioned HLA entities (i.e. the newly introduced actors) are modelled by PtII relations. Thereby, the names of the actors correspond to the names of the respective entities within an HLA object model. HLAFederate actors are only allowed to connect to HLAObjectClass actors and communicate with each other via these. Beside that, federation models can but need not include PtII as federate. PtII can solely be used for FM definition and interface generation. In turn, a HLAFederate actor can represent any type of simulator, including another PtII instance. In order to allow a PtII instance to take part in a federation, ports need to be added to the composite actor that allow connecting to the residual PtII simulation model to HLAObjectClass actors. The structure of HLA object classes and the datatypes of attributes and parameters are determined by the PtII datatypes of the actor ports. The direction of communication between federate actors and object class actors through ports and relations determines the publish/subscribe relationships. In the example of Fig. 5, a single external simulator represented by a federate actor named "'RemoteSimulator"' is co-simulated with a local PtII model. Connection to a local PtII model is achieved by introducing PtII input and output ports (large arrows in corners of the composite) that connect the FM to the HLA composite. The local PtII model and the remote simulator solely communicate via the ObjectClassX and InteractionClassY actors.

*2) Definition of Control Flow:* The kind of interaction and control flow between simulators that are part of a federation is specified by their BIM FSMs. The BIM FSMs limit the number of interaction patterns between simulators that interact via HLA to a subset of all interaction patterns that are allowed by HLA. In the current version of the PtII extension, BIM FSM descriptions must be annotated to actors as C++ code. In case PtII taking part as federate, the BIM FSM must be annotated to the HLAComposite in case of any other simulator it needs

to be annotated to the respective HLAFederate actor.

### B. Interface Generation

Tool interfaces can be generated automatically. A generated interface serves as a separate layer between a simulator and HLA. From the overall FM four types of artifacts are generated:

- FOM related artifacts: Currently it is possible to automatically generate the .fed file which is necessary for configuring the RTI.
- SOM: The SOM is exported as C++ code. SOM specifications follow the syntax as it has been defined in section III-A (i.e. an inherited class of the ObjectModel class with respective OM instantiation code).
- Behavioral Interface Model FSM: As already mentioned above, BIM FSMs currently need to be annotated as PtII parameter to federate actors and/or the HLA composite actor. BIM FSMs follow the syntax defined in section III-B. From the annotated FSM description an inherited class of the BehavioralInterfaceModel class is generated for each federate.
- High level interface classes: These are generated for each federate and set on top of the SPLib classes. They include derived types of IOSocket and a C++ wrapper class. The latter provides a function-based interface to the Adaptor and IOSocket classes. This greatly simplifies usage. Beside that, SWIG [14] interface files (.i files) can be generated. SWIG is incorporated in order to wrap the C++ wrapper class once again for integration into simulators that are based other languages than C++. Interface files are used to configure SWIG with special type mappings between C++ and a target language. The resulting component is called *interface wrapper* in the following.

For interface generation there exists a special director called HLAGenDirector. This director must be added to the HLAComposite. The HLAGenDirector has access to different ModelInterpreter classes as shown in Fig. 6. These provide concrete (language specific) interpretations of the syntactical artifacts provided by the FM. Each of the TypeInterpreter classes contains tables with which PtII datatypes are mapped to language specific types like C++, Java or HLA datatypes. Hence, the PtII type system serves as reference type system from which language specific types are derived. The SOMInterpreter is used to fill the datatype tables of the type interpreters with SOM specific content. Both, SOMInterpreter and FOMInterpreter classes provide the capabilities for generating the above mentioned FOM and SOM related artifacts. Finally, the JavaInterfaceInterpreter and CPPInterfaceInterpreter classes inherit from SOMInterpreter and provide additional capabilities for deriving the signature of the data flow related Java and C++ interface functions by which the interface wrapper is accessed from outside. CPPInterfaceInterpreter is capable of generating the high level interface classes, the JavaInterfaceInterpreter is capable for generating the Java specific SWIG interface file.

The resulting artifacts are compiled with the SPLib classes into a shared C++ library. If the target simulator is written in C++ (like e.g. the open source SystemC or OMNET++ simulators) the shared library can directly be integrated and linked to the simulator/model. If the target simulator is written in another language like Java, C# or a scripting language like Python, SWIG must be called for target language wrapper generation.

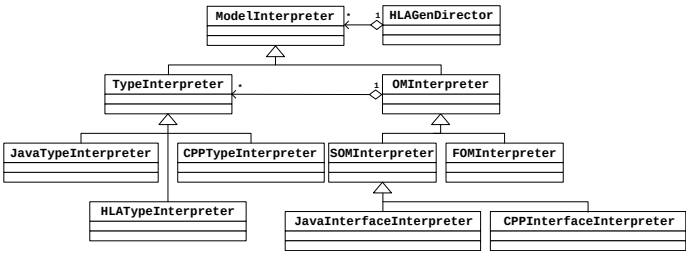


Fig. 6. Model Interpreters used within the HLAGenDirector

### C. Interface Integration

The interface wrapper appears to a simulator as a single class that provides a fixed number of control flow and a variable number of data flow related high level methods. The control flow related methods are provided by the ToolAdaptor class, the data flow related ones by the IOSocket classes (see also section III). They have a fixed signature. In the current implementation, control flow related methods are

- getState(): Return the current state of the BIM FSM.
- setExpState($s_{exp}$): The expected $s_{exp}$ state value may be used by the FSM to verify that it matches the actual next interaction state.
- setNextBarrier(): Set the next time barrier that the simulator wishes to advance.
- getNextBarrier(): Get the last time barrier that has been granted.
- iterate(): Iterate the BIM FSM. Executes sequences of exec() and post() until post() returns 0. This method must regularly be called by the environment simulator in order to advance the BIM FSM state.
- end(): Generate a Sim_Event_End event for the BIM FSM.

Generally, interface integration can be viewed as a mapping of BIM FSM states into the execution phases of the target simulator. Exemplarily transferred to a PtII director these phases are preinitialize(), initialize(), prefire(), fire(), postfire() and wrapup(). For PtII integration, a special HLADEDirector has been developed which derives from the original DEDirector and which integrates an interface wrapper following the described approach (a prestudy has been done in [15]). Since PtII is written in Java, a SWIG based wrapper is used and loaded dynamically. The mentioned control flow related calls can theoretically occur in any of the execution phases of the HLADEDirector. The applied BIM FSM (the same has been used for SystemC and Veins) is illustrated in Fig. 7. It consists of a toplevel FSM with four hierarchical states and an atomic state. In S_INIT HLA initialization like federation creation, publication and subscription as well as instance registration is performed (for the latter, S_INIT internally contains an interaction state). The S_SYNC_READY_TO_RUN and S_SYNC_SHUTDOWN states

are refined by FSMs that implement synchronization procedures by means of HLA synchronization points which ensure that simulators start and stop execution (the latter includes instance deletion and the shutdown procedure) at the same time. The `S_EXECUTE` state is responsible for time advancement (federates are time regulating and time constrained). It is an interaction state and a refined state at the same time. Interaction with the environment simulator is only performed if there does not exist an event in the FSM event queue (symbolized by `Event_Absent`). Otherwise, the refinement states ensure that only valid events can occur during the time advancement procedure. Finally, if the toplevel end state becomes the next state, the FSM terminates. The current implementation matches the `S_INIT` and `S_SYNC_READY_TO_RUN` states to the `initialize()` phase, the `S_EXECUTE` state to the `fire()` phase and the `S_SYNC_SHUTDOWN` and `S_END` states to the `wrapup()` phase of the `HLADEDirector`. If the `fire()` method of the `HLADEDirector` is called and the timestamp of the next local event (requested by `getModelNextIterationTime()`) is larger than the previously granted time, the director proposes an advance, iterates the BIM FSM and sets the next fire time via `fireContainerAt()` to the granted time. Afterwards reflections are served. Due to the microstep semantics zero lookahead was chosen. In order to be able to achieve deterministic distributed execution by means of the HLA 1.3 NER service, a priority field method similar to the one described in [16] can be applied. In this case, also the microstep needs to be passed when requesting advancement.

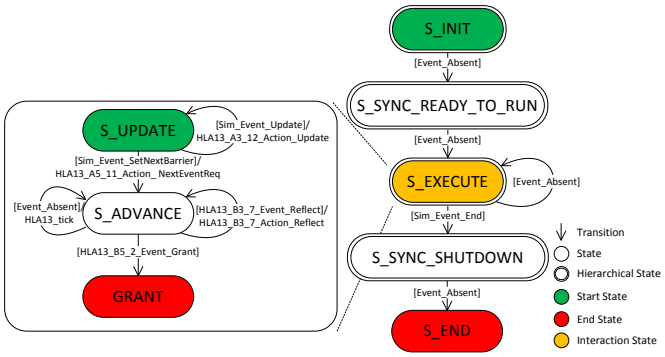


Fig. 7. Behavioral Interface Model FSM

In contrast to the control flow methods, data flow related methods do not necessarily have a fixed signature. Methods for reading and writing start with `read` or `write` followed by the name of the object class that is accessed. The instance name is a method parameter. Number and types of other parameters depend on number and types of attributes that are stored within the respective object class. Data flow related methods are

- `readX()`: Read topmost reflection of `IOSocket`/object class "X".
- `writeX()`: Write an update into `IOSocket`/object class "X". This effectively generates a `Sim_Event_Update` event for the BIM FSM. In addition, a timestamp must be passed to the `write` call.
- `addRemInst()`: Generate event for adding/removing object class instances.
- `getSocketQueueSize()`: Return the fill level of a specified `IOSocket`.
- `popSocket()`: Pop the topmost reflection from a specified `IOSocket`.

The `HLADEDirector` manages data flow between a local PtII model and the interface wrapper with support of the `HLAObjectClass`. The `HLADEDirector` is equipped with a `reflectDir()` and `updateDir(token)` method. The `reflectDir()` method is called by the `HLADEDirector` itself at any point in simulation time when a new time barrier has been read by calling `getNextBarrier()`. The method checks `IOSocket` buffers for available reflections by calling `getSocketQueueSize()`. If a reflection is available the full signature of the `read` method is dynamically derived from the object class structure of the SOM using the `JavaInterfaceInterpreter` class. The reflection is then converted into a PtII token. Afterwards, the token is transferred to the `HLAObjectClass` actor that belongs to the `IOSocket` which provided the reflection. For that reason, each `HLAObjectClass` actor is equipped with a `reflectAct(token,time)` method. In turn, the `updateDir(token)` method is called by `HLAObjectClass` actors. The method converts the passed token by the help of the `JavaInterfaceInterpreter` class into a `write` method call on the interface wrapper. The timestamp of the generated update event corresponds to the current local simulation time. The semi-automatic integration by dynamic signature derivation makes sense since PtII plays a central role within the overall tool chain and is probably applied in many different and extensible co-simulation scenarios.

During execution, only `HLAObjectClass` actors that connect to the `HLAComposite` ports play an active role. They represent the HLA object classes (to) which the residual probably heterogeneous PtII model subscribes/publishes. They are necessary for transferring data between the residual PtII model and the federation and vice versa. All other `HLAObjectClass` and `HLAFederate` actors remain passive and will never fire during execution. The `reflectAct(token,time)` method of a `HLAObjectClass` actor stores the passed token in an internal timed queue. The method internally calls `fireAt(time)`. This makes sure, that the actor is fired at the point in time, when a reflected token should be forwarded to the residual PtII model through the corresponding output port. Therefore, the time parameter must correspond to the time of the grant. The opposite direction works as follows: As soon as an `HLAObjectClass` actor is fired it checks its input ports for available tokens. if a token is found it is passed to the `HLADEDirector` by calling `updateDir(token)`.

Overall, by means of the combination of PtII and HLA in the described manner, the following types of execution are conceivable:

- *Standard Mode*: All simulators are part of a single federation. This mode is best applied if all simulators follow related execution semantics. E.g. integrating several identical DE simulators into a single federation is a straight forward issue since both, RTI and the simulators are basically event-based. A typical application is parallel simulation for improving performance.
- *PtII Managed Mode*: Generally, a "brute-force" composition of heterogeneous simulators following different

MoCs may result in so called *emergent behavior* [4]. Hence, in this mode, the RTI is assisted by PtII for managing heterogeneity. This enables structured composition of federations and modification of data and control flows between them. Single simulators are co-simulated with PtII within separate federations using separate `HLAComposite` actors and appropriate SOMs and BIM FSMs. PtII serves as federation gateway and mediates between these federations. A possible drawback is decreased performance.

- *Mixed Mode*: In this mode, the previously described modes are executed together, i.e. not only PtII and single simulators form separate federations but a co-simulation federation can contain more simulators. It combines advantages of both modes, namely better execution performance and flexibility in model composition.

## V. Case Studies

Applicability of the previously described tool chain is now demonstrated by means of a framework that has been developed with the tool chain. Thereby, validation of a V2X based ACC application running on a future automotive E/E architecture has been chosen as example. Pursued goals of the case studies are I) demonstration of the basic capability of heterogeneous distributed co-simulation II) demonstration of the applicability of PBD within the tool chain III) demonstration of applicability for verification using several federations in parallel.

### A. Simulation Setup

The overall framework is shown in Fig. 8. It consists of PtII, Veins and SystemC federates. The simulation setup has been designed according to the PtII managed mode described in section IV-C, i.e. single simulators are co-simulated with PtII within separate logical federations using separate `HLAComposite` actors. All federates are integrated by means of the interface wrapper described above. In case of Veins and SystemC the interface wrapper has been integrated manually.

Within the PtII federate the E/E architecture of the vehicles of interest is modelled. These intra-vehicle models are connected to an inter vehicle model that is provided by the Veins composite. Veins is an open source vehicular network simulator that integrates the OMNET++ network simulator with the SUMO traffic simulator bidirectionally using a TCP/IP proxy. It allows simulating IEEE 802.11p wireless networks including node mobility. Within the framework Veins simulates surrounding vehicles and the inter vehicle communication via V2X messages. We've modified the application and mobility modules of Veins by the introduction of additional parameters that allow indicating which vehicles are remotely controlled by PtII or if WiFi messages are received/sent remotely from/to PtII. This is a prerequisite for validation of the ACC functionality. Regarding synchronization a parameter can be set for each vehicle representing the time interval for updating messages exchanged with PtII. Analogueous, there is a parameter which specifies the update time interval between SUMO and OMNET++ via TCP/IP. In the different considered scenarios these are set to a value of $0.01s$. Finally, by means of a SystemC federate selected ECUs within the E/E architecture model can be refined down to fully cycle accurate descriptions. As an example a detailed SystemC model of a multi-core architecure called HeMPS [17] which represents a refined V2X ECU has been integrated. The model consists of a configurable number of processing elements (PEs) which are interconnected by a Network-on-Chip (NoC) called HERMES [18]. The model is equipped with a HLA-based virtual internal/external network interface that connects the model to the interface wrapper.

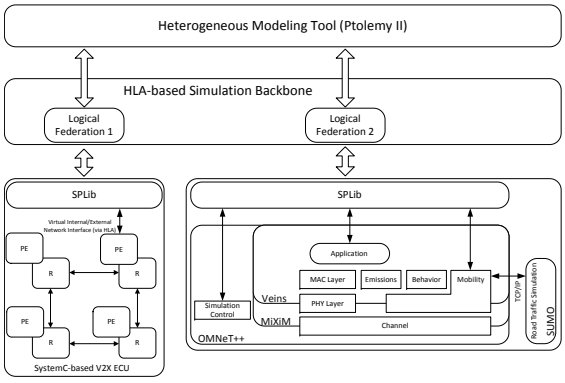


Fig. 8. Case Study Simulation Framework

### B. Adaptive Cruise Control Application

Current ACC systems monitor the distance to the vehicle in front using radar sensors. Based on the monitored values speed and distance are automatically adjusted by motor and brake intervention. However, conventional ACC monitoring is limited to the line of sight. This can be a safety critical factor e.g. in case of a sudden traffic congestion within a curve. By extending the ACC with V2XC capabilities, an automatic reaction to such situations is possible.

The PtII model of the considered scenario is shown in Fig. 9. On the top level, the overall simulation model is instantiated, containing Veins and separate composite actors for selected vehicles. Vehicles can be refined to E/E architectures. Here, we orient ourselves by design principles suggested for future E/E architectures [1] like a centralized computer architecture and interconnection by a standardized communication backbone. We assume, that the basic structure of the target E/E architecture is predefined while implementation of components like ECUs may vary. Inside the E/E architecture data is routed through a network model represented by an `Internal Network` composite. V2X ECU and Central ECU are assumed to be multi-core architectures which execute the ACC application. Radar ECU and GPS ECU are responsible for data acquisition and preprocessing of radar sensor and GPS data where the Radar ECU provides position and velocity of the vehicle ahead and the GPS ECU the own position. The four Wheel ECUs take the acceleration calculated by the ACC as an input and drive the corresponding wheels to the desired velocity. The determined velocity value is fed back to the top level model and forwarded to Veins where the velocity of the corresponding remotely controlled SUMO vehicle is adjusted. The Wheel ECU composites, representing the control path of the ACC application, are directed by a `ContinuousDirector`. The velocity provided by the Wheel ECUs is discretized in order to be reused for transmission to connected ECUs. The sampling rate is set to $0.01s$

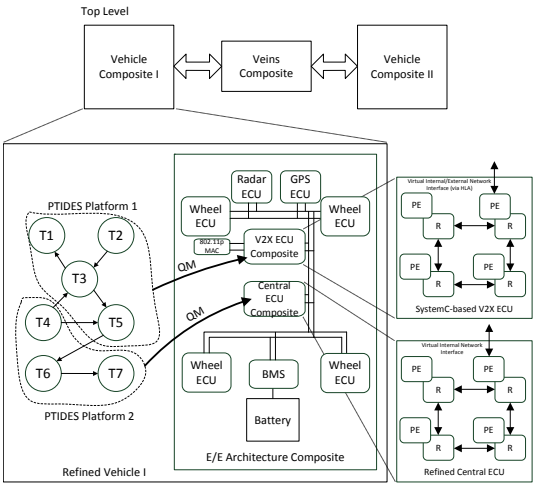


Fig. 9. V2X-based ACC Validation Scenario

### C. Heterogeneous Distributed Co-Simulation

For the first example scenario we've constructed an urban traffic scenario in SUMO where a leading vehicle is periodically approaching crosses with a distance of 200 m. The mobility model of SUMO [19] results in regular decceleration and acceleration cycles. A second follower vehicle is controlled remotely by PtII, whose velocity is set with the value calculated by the ACC application. The latter is modelled as an abstract DE model located within the V2X ECU composite. The traffic simulation is executed with a forerun of $10s$. In this section ACC scenarios are considered that either use radar or V2X communication. In both cases, velocity and position values of the leading vehicle are transferred to PtII, either directly or via the 802.11p wireless channel model.

In Fig. 10 the velocity measurements of radar only controlled ACC is shown. The red and blue curves belong to the ahead and refined vehicle respectively. As we can see, the refined vehicle whose velocity is calculated in PtII follows the leading vehicle controlled by SUMO with a certain distance that depends on ACC parameters such as a safety time gap (set to $1.7s$). The calculated acceleration is plotted in Fig. 10. On the right hand side in Fig. 10 we can see the position of the leading and the following vehicle. The difference between both curves represents the distance between both vehicles and is also reflected by the distance plot in Fig. 10.

To analyse the influence of the V2X based ACC, several simulations with different V2X message beacon rates of $0.1s$, $1.0s$ and $10.0s$ have been conducted. With a beacon rate of $0.1s$ the curves are identical to the radar ACC. When increasing the beacon rate to $1.0s$ and $10.0s$ (low beacon rates could arise e.g. in lossy or jammed channels) the progress of the velocity of the following vehicle gets less comfortable in terms of maneuver with high deceleration rates or higher velocities. This gets more obvious the lower the beacon rates get since the ACC algorithm gets updates about the actual position and velocity of the leading vehicle less frequently. This can be observed in Fig. 11 but even with a beacon rate of $10.0s$ the scenario still remains crash free. However, if the beacon rate gets too low, the ACC cannot react properly anymore and crashes into the leading vehicle. This has been

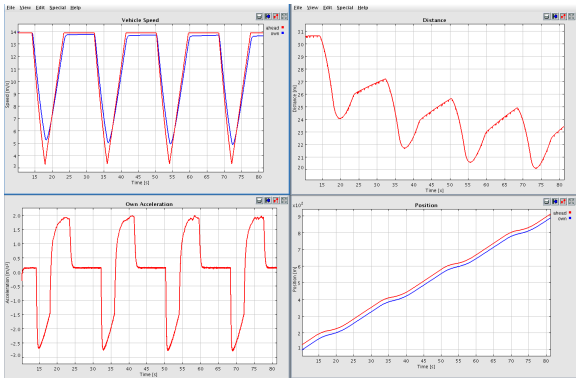


Fig. 10. Velocities (top left), Own Acceleration (bottom left), Distance (top right) and Position (bottom right) of radar controlled ACC

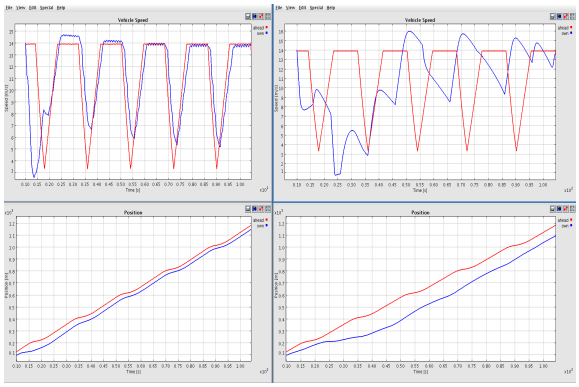


Fig. 11. Velocity and Position of Ahead and V2X Controlled Vehicle with Left: Beacon Rate = 1.0s, Right: Beacon Rate = 10.0s

observed with a very low beacon rate of $50.0s$ where a crash occured after around $49s$.

### D. Application of Platform-based Design Principles

As has been shown, the framework allows modelling of functional and architectural artifacts as well as environmental components, that are necessary for a holistic analysis and system validation. In the following it is shown that by replacing abstract artifacts through more detailed entities, an iterative refinement of an abstract specification towards the final implementation is possible. Moreover, if the process of refinement is based on *quantity managers* (QM) [20] a mapping of functions to co-simulated architecture models is possible. PtII provides capabilities for computation and communication refinement using so called *aspects* which are a synonym for QMs [21].

As mentioned above, messages exchanged within an E/E architecture model are routed through an `Internal Network` composite. For communication refinement, the ports at that composite can e.g. be annotated with a bus communication aspect allowing investigation of bus delay influences. Furthermore, the vehicle composites can be refined towards functional and architectural models which is the basis for computation refinement. The function (ACC application) is specified by means of a task graph consisting of Tasks T1-T7. A task is represented by a (composite) actor which models a specific sub-functionality of the overall V2X-ACC application. Hereby, the single tasks fulfill the following sub-functionality:

1) V2X data transmission 2) V2X data reception 3) V2X data security measures 4) radar sensor and GPS data processing 5) data aggregration 6) V2X-ACC algorithm based on the *Intelligent Driver Model* (IDM) 7) actuator control.

Within this case study, Central ECU and V2X ECU are responsible for the tasks T1-T7. The latter can be mapped on the former by means of a newly developed quantity manager actor called `QMMapper`. The mapping and routing are specified by routing entries in a mapping/routing table used by the `QMMapper`. This is the basis for a later automization of design space exploration. For investigating task mapping influences, the following methodology is imaginable:

1) Tasks are clustered into task groups consolidated in further composite actors. These composites follow the PTIDES MoC [22] time-synchronized distributed real-time applications. A PTIDES composite is also called *PTIDES platform*.
2) PTIDES platforms are mapped onto V2X ECU resp. Central ECU which is accomplished by means of the `QMMapper`.
3) The specification of each of the ECUs is refined from an abstract model down to an individual cycle accurate SystemC implementation. Each PTIDES platform is equivalent to a specific PE within the abstract composite/co-simulated SystemC model.

In the following measurements a V2X based ACC has been configured with a beacon interval of $1.0s$. Two task groups, i.e. PTIDES platforms called `ACC` (T4-T7) and `V2X` (T1-T3) have been created. Central and V2X ECUs are modelled abstractly. Each is made up of four PEs interconnected by an abstract NoC. Scheduling issues are not considered in this use case but can be incorporated by annotating different kinds of so called execution aspects [21]. Table I summarizes the mapping variations of the platforms. In both cases the `ACC` platform is mapped onto the Central ECU and the `V2X` platform onto the V2X ECU but on different PEs inside the ECU. Thereby PE 0 is a computational core with additional access to the network interface, i.e. every message to be received/sent is processed in PE 0 first. It is obvious that in Mapping 1 no additional communication delay through the NoC emerges whereas in Mapping 2 all messages need to be received/sent from/to PE 3. Furthermore, in both mappings the received V2X messages have to pass the internal network since the `ACC` and `V2X` platforms are mapped onto Central and V2X ECU respectively. To see the influence of communication delay caused by the E/E internal network and the NoCs of the Central and V2X ECU, we generated synthetic congestion resulting in bus service times of the internal network of $2ms$ and in NoC delays per hop of $500ms$. Such high delays per hop can arise in simple NoCs in case of congestion when there are no quality of service countermeasures. Thus in Mapping 2 there are three hops necessary in the V2X ECU to forward received velocity and position values of the vehicle ahead to PE 0 before the packet can be sent out via the internal E/E bus to the Central ECU. In the latter again all incoming messages have to be forwarded from PE 0 to PE 3 before the ACC algorithm can calculate new acceleration values. This results in a communication delay of about $6 * 500ms = 3s$ where the internal bus delay is neglected. This can be obeserved in Fig. 12 where the braking maneveur of Mapping 2 is delayed by calculated value compared to Mapping 1 where only the internal bus delay has an influence.

TABLE I. PTIDES PLATFORM MAPPINGS

| Mapping 1 | | | |
|---|---|---|---|
| **Platform** | **Tasks** | **ECU** | **PE** |
| ACC | T4-T7 | Central ECU | 0 |
| V2X | T1-T3 | V2X ECU | 0 |
| Mapping 2 | | | |
| **Platform** | **Tasks** | **ECU** | **PE** |
| ACC | T4-T7 | Central ECU | 3 |
| V2X | T1-T3 | V2X ECU | 3 |

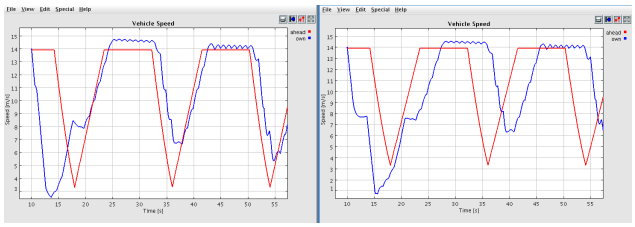


Fig. 12. Velocities depending on Mapping, Left: `ACC`=CentralECU PE0, `V2X`=V2XECU PE0, Right: `ACC`=CentralECU PE3, `V2X`=V2XECU PE3

### *E. Verification based on Multiple Federations*

The last case study serves as demonstration of the feasibility of PtII managed mode and mixed mode. In the case of discrete-event models a distinct execution order of actors is the basis for achieving causally correct and deterministic simulation results. For that case, PtII assigns priorities to events by sorting them according to model time, microstep and level [21]. The microstep specifies a zero delay at a certain point in time whereas the level is determined by a topological sort of a directed acyclic graph (DAG) of the actors. Therewith an ordering is assigned such that an upstream actor in the DAG executes earlier than a downstream actor. In order to be able to build a DAG, loops between actors must contain at least a microstep delay. The synchronization algorithm that has been presented above relies on the NER service of the HLA. An `HLADEDirector` proposes to advance to a certain point in time that is specified by the `setNextBarrier()` call. This includes passing the next time that is returned when calling `getModelNextIterationTime()`. In order to guarantee causal correct execution the time delta from the current point in time to the proposed time must not be larger than the minimum possible delay until the `HLAComposite` may possibly receive a new token. This requirement is always fulfilled for a single HLA composite and for multiple HLA composites without loops between them (due to the execution ordering). However, in case of the existence of loops between HLA composites the next time returned by `getModelNextIterationTime()` cannot unconditionally be taken for proposition via `setNextBarrier()`, even in the case of delays that are inserted. The reason is, that after the grant of a `HLADEDirector` $A$ within an upstream `HLAComposite` to time $t_A$, a `HLADEDirector` $B$ within an adjacent downstream `HLAComposite` that requests at the same time and microstep could be granted to a time $t_B < t_A$ and violate causal correctness by inserting a token that will be sent to federation $A$ at an already passed point in time. A simple solution is to derive a minimum lookahead time $> 0$ within the PtII model and regularly generate events having a

time distance of the derived minimum lookahead. This solution has been chosen here. The path from Veins to HeMPS contains no delay. The path from HeMPS to Veins via the Wheel ECU hence must contain at least a microstep delay to create the DAG. Concerning lookahead, since the Wheel ECU generates events with the same frequency as Veins (i.e. 1/0.01s), there will always be an event in due time which avoids that Veins requests too far into the future. Hence, Veins will never receive tokens in the past and causal correctness is preserved. Fig. 13 illustrates the measured results for a simulated time interval of 10s when executing the radar ACC application as Kahn Process Network (KPN) on the cycle accurate HeMPS model. The model has been simulated with clock frequencies of 100 kHz respectively 50 kHz. As can be seen 100 kHz version reacts slightly faster which results in the velocity to start increasing again already at 19s whereas the 50 kHz version still remains at 0 m/s. The execution for 10s of simulation time was 25min at 50kHz and 52min at 100kHz on a core i5 dual-core at 2.5GHz.

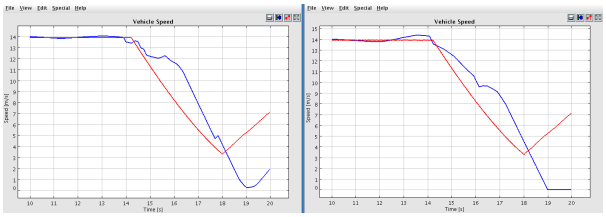


Fig. 13. Radar ACC as KPN Application on HeMPS (100 kHz left, 50 kHz right), red = ahead vehicle, blue = follower vehicle

## VI. Distinction to Related Works

In [23] a metamodel for federation architectures is presented. However, no solution for managing composition of heterogeneous simulation models is provided. In recent model-based approaches for investigating V2X communication like [24], [9] the emphasis lies on application analysis on a more coarse grained level like traffic management and efficiency. Because of that, in these works vehicles can be considered as a focused processing point. However, for validation of E/E architecture components, such a point-of-view is insufficient. There is rather the need for a comprehensive description of a complete V2X processing chain starting from the sensor of the source vehicle and ending up at the actor of the destination vehicle. The concept presented in [25] targets such a view but it is not mentioned, how heterogeneity could be managed or PBD principles could be integrated. In the area of hardware/software co-simulation [26] or [20] are prominent tools. However, they do not provide capabilities for structured integration of simulators.

## Acknowledgment

This work was supported by the *Sino-German Network on Electromobility*.

## References

[1] C. Buckl *et al.*, "The software car: Building ict architectures for future electric vehicles," in *Electric Vehicle Conference (IEVC), 2012 IEEE International*, 2012, pp. 1–8.

[2] B. Glas *et al.*, "Car-to-car communication security on reconfigurable hardware," in *VTC Spring'09*, 2009, pp. –1–1.

[3] A. Sangiovanni-Vincentelli and M. Di Natale, "Embedded system design for automotive applications," *Computer*, vol. 40, no. 10, pp. 42–51, 2007.

[4] J. Eker *et al.*, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[5] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)," *IEEE Std 1516.x-2010*, aug. 2010.

[6] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638.

[7] C. Roth *et al.*, "Improving parallel mpsoc simulation performance by exploiting dynamic routing delay prediction," in *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*, 2013, pp. 1–8.

[8] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," ser. Simutools '08. ICST, Brussels, Belgium, Belgium: ICST, 2008, pp. 60:1–60:10.

[9] C. Sommer, R. German, and F. Dressler, "Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis," *IEEE Transactions on Mobile Computing*, vol. 10, no. 1, pp. 3–15, January 2011.

[10] M. Treiber. (2013, 06) Microsimulation of road traffic flow. [Online]. Available: http://www.traffic-simulation.de/

[11] C. Roth *et al.*, "Efficient execution of networked mpsoc models by exploiting multiple platform levels," *Int. J. Reconfig. Comput.*, vol. 2012, pp. 27–39, Jan. 2012.

[12] X. Liu, Y. Xiong, and E. A. Lee, "The ptolemy ii framework for visual languages," in *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, ser. HCC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 50–.

[13] E. Noulard, J.-Y. Rousselot, and P. Siron, "CERTI, an Open Source RTI, why and how," in *Proceedings of the Spring Simulation Interoperability Workshop*, 2009.

[14] D. M. Beazley, "Automated scientific software scripting with swig," *Future Gener. Comput. Syst.*, vol. 19, no. 5, pp. 599–609, Jul. 2003.

[15] A. Brito *et al.*, "Development and evaluation of distributed simulation of embedded systems using ptolemy and hla," in *International Symposium on Distributed Simulation and Real Time Applications*, 2013.

[16] R. M. Fujimoto, "Zero lookahead and repeatability in the high level architecture," in *In Proceedings of the 1997 Spring Simulation Interoperability Workshop*, 1997, pp. 3–7.

[17] E. Carara *et al.*, "HeMPS - a framework for NoC-based MPSoC generation," in *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, 2009.

[18] F. Moraes *et al.*, "HERMES: an infrastructure for low area overhead packet-switching networks on chip," *Integr. VLSI J.*, vol. 38, 2004.

[19] M. Treiber, A. Hennecke, and D. Helbing, "Congested traffic states in empirical observations and microscopic simulations," *Rev. E 62, Issue*, vol. 62, p. 2000, 2000.

[20] A. Davare *et al.*, "A next-generation design framework for platform-based design," in *DVCon 2007*, February 2007. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/228.html

[21] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. [Online]. Available: http://ptolemy.org/books/Systems

[22] Y. Zhao, J. Liu, and E. A. Lee, "A programming model for time-synchronized distributed real-time systems," in *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, ser. RTAS '07. IEEE Computer Society, 2007, pp. 259–268.

[23] O. Topçu, M. Adak, and H. Oğuztüzün, "A metamodel for federation architectures," *ACM Trans. Model. Comput. Simul.*, Jul. 2008.

[24] T. Queck *et al.*, "Realistic simulation of v2x communication scenarios," in *Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE*, 2008, pp. 1623–1627.

[25] C. Roth *et al.*, "Car-to-X Simulation Environment for Comprehensive Design Space Exploration Verification and Test ," *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, vol. 3, no. 1, 2010.

[26] "Synopsis Platform Architect." [Online]. Available: www.synopsys.com/Systems/ArchitectureDesign/Pages/PlatformArchitect.aspx