CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

A Domain Specific Language for Automotive Systems Integration

Renato Oliveira
David Pereira
Cláudio Maia
Pedro José Santos

CISTER-TR-190806

2019/10/14

A Domain Specific Language for Automotive Systems Integration

Renato Oliveira, David Pereira, Cláudio Maia, Pedro José Santos

CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: prmol@isep.ipp.pt, drp@isep.ipp.pt, clrrm@isep.ipp.pt, pjsol@isep.ipp.pt

https://www.cister-labs.pt

Abstract

Developing complex safe and secure Cyber-Physical Systems(CPS)applications for the automotive domain is typically a complex task, due to the criticality inherent to this domain. Considering such known complexity of the development process, we propose a novel solution that aims to provide a quasi automatic integration process between the different components of such CPS systems via the support of a Domain Speci?c Language (DSL) that provides several views of the system, abstracting away the more technical implementation details, while imposing system properties and rest rictions that have the potential to be formally veri?ed (either statically or at run-time) during design, and facilitates the process of customization and quasi-automatic build and deployment processes. In this paper, we brie?y analyze the tools that are available and that cover partially the characteristics of our envisioned DSL, describe its building blocks, and show how it can be applied in a small, yet suf?ciently complex CPS application whose architecture is very close to what we may expect for the modern and future generation of CPS application in the automotive domain.

A Domain Specific Language for Automotive Systems Integration

Renato Oliveira, David Pereira, Cláudio Maia, Pedro Santos
CISTER Research Centre – ISEP, P.Porto, Porto, Portugal
{prmol, drp, clrrm, pjsol}@isep.ipp.pt

***Abstract*—Developing complex safe and secure Cyber-Physical Systems (CPS) applications for the automotive domain is typically a complex task, due to the criticality inherent to this domain. Considering such known complexity of the development process, we propose a novel solution that aims to provide a quasi-automatic[1] integration process between the different components of such CPS systems via the support of a Domain Specific Language (DSL) that provides several views of the system, abstracting away the more technical implementation details, while imposing system properties and restrictions that have the potential to be formally verified (either statically or at run-time) during design, and facilitates the process of customization and quasi-automatic build and deployment processes. In this paper, we briefly analyze the tools that are available and that cover partially the characteristics of our envisioned DSL, describe its building blocks, and show how it can be applied in a small, yet sufficiently complex CPS application whose architecture is very close to what we may expect for the modern and future generation of CPS application in the automotive domain.**

***Index Terms*—system verification, CPS application development, system integration, domain specific languages**

I. Introduction

Developing complex CPS applications for the automotive domain is a complex task, due to the criticality inherent to this domain and the need for increased functionality, safety, and security. A considerable part of the development effort relies on the integration of third-party and vendor specific software components, thus involving the usage of plethora of different technologies, typically developed in isolation (and by different entities) and, later in the process, integration and verification efforts that delay the time to market of new solutions, and higher production costs. The different technologies typically involved in the development of automotive applications range from distinct modeling frameworks, code editors and programming languages, development frameworks, operating systems, COTS platforms, support for virtualization via hypervisors, among others. Such diversity of technologies leads to an integration that can be very complex, tedious, error prone, and time-, money- and effort-consuming [1].

As the complexity of these CPSs increases, our inability to rigorously capture the interactions between the physical and the computation components paves the way to software errors that can lead to serious vulnerabilities during operations. Ultimately, systems become unsafe, with disastrous failures that could not have been properly identified during the several stages of the application's development cycle [2].

[1]By quasi-automatic we envision a scenario where consistency checking is mostly made by the solution itself, automatizing the integration process

In order to properly address the complexity of the design and development process, we propose the development of a DSL that aims to provide a quasi-automatic integration process between the different entities or components of a CPS application, abstracting away technical details of these components and enable rigorous methods of verification, customization, and deployment. To this end, the DSL provides different *views* of the system, where properties or constraints imposed in a view propagate to other views, increasing the set of system-wide properties that need to be ensured correct in order for the system to be considered safe. By providing distinct views, our envisioned solution allows different system intervenients to seamlessly interact during the development process, simultaneously ensuring that the underlying infrastructure used for this interaction is verified, and that CPS applications and systems can be easily built, customized and deployed.

This paper is organized as follows: in Section II we briefly review languages that have common features to the DSL we are proposing; in Section III we analyze the development environments that facilitate the efforts to implement a new DSL; in Section IV we introduce our DSL and show how it can be used in the development of a CPS application with complex structure and involving different types of components (system-wide); finally, in Section V we draw some conclusions and point to future work.

II. Related Work

There are a number of languages that help the development of complex CPS applications by enabling the integration between different components and systems. In this section we briefly overview some of such languages.

A. SysML

One of such languages is the *System Modelling Language* (SysML), which aims to provide cross-domain technology and system integration. SysML addresses issues associated with *multi-view modeling* [3] and offers a unifying language between the various views of the system [3]. SysML is based in the *Unified Modelling Language* (UML) and allows for the creation of a plethora of diagrams that output models, which can then be used to generate code.

SysML allows basic operation signatures to be defined at interfaces, and pre- and post-conditions to be specified textually [4]. This feature is tightly connected with the concept of *Design by Contract* (DbC) [4]. Contract verification, in the case of SysML, requires the translation of these contracts into the formal notation of the *COMPASS Modelling Language*

(CML) [4]. As this is a somewhat manual process, and since it relies on non-formal, natural language for a textual description of the contracts, it does not enable the rigorous specification of system properties or/and constraints that need to be verified in order to ensure system correctness.

SysML cannot transform models into target code by itself, which introduces other tool dependencies, like *Enterprise Architect* [4]. Additionally, SysML is by design a diagram oriented language, which is not always the more convenient way of satisfying the ever increasing necessity of incorporating different applications in a single solution that typically relies on tweaking code to achieve such integration.

B. AADL

Another language that provides integration support among different components of a system is the *Architecture Analysis and Design Language* (AADL). AADL was designed for the specification, analysis, integration and code generation, based on requirements that refer to performance-critical systems.

AADL allows the analysis of CPS systems prior to their development, supporting model driven development approaches throughout the system life cycle. Due to its focus on the embedded systems domain, it can be used to produce both software and hardware, making it a very versatile tool for designing systems.

AADL can be used to manage both system and software aspects within the same model, offering a single notation mechanism for both. The existence of a single model eases the analysis because there is only one representation of the system, eliminating possible ambiguous notions that may emerge. Therefore, in order to properly produce a model, any AADL user can extend the language by defining properties, which in turn are system specific characteristics.

It is possible to extend the AADL language for a specific domain, via annexes. Language annexes enhance the core AADL language in order to provide an enrichment to the architecture description and definition. At the present time, there are four defined annexes: the (1) behavior annex, which adds behavior to components with state machines; the (2) errormodel annex which specifies fault and propagation concerns; the (3) ARINC653 annex which defines modeling patterns for avionics systems; and the (4) data-model annex which describes the modeling of specific data constraints within AADL. With that being said, it would be possible to design a DSL using the annex definition capabilities of AADL. We chose not to pursue this path due to its inherent focus on real-time performance-critical (timing, safety, schedulability, fault tolerant, security, etc.) systems. These properties, while useful in some domains, might not fit the scope of the solution we are trying to envision. Additionally, the reason for not using AADL stems from the difficulty of representing different views using this language, at least in the way we envision them.

III. Tools for DSL development

In this section we will describe two candidate DSL development tools, namely the Xtext tool and the Jetbrains Meta Programming System (MPS).

A. Xtext

Xtext is a framework that allows one to quickly develop tools for a textual language [5]. It can be used to develop DSLs or General Purpose Languages (GPL). Additionally, it has support for configuration files and requirement documents.

Xtext takes advantage of the *Eclipse Modeling Framework (EMF)* and allows easy integration with tools from the Eclipse Modeling ecosystem, such as Model-to-Model or Model-to-Text transformation languages [5]. Starting with a grammar definition, Xtext generates a parser, serializer and a smart editor for the language. All these generated artifacts can be configured or customized via dependency injection [5].

Unfortunately, considering the domain approached in the context of this paper, a need for the validation of various concerns emerges, those concerns being timing, safety, schedulability, fault tolerance, security, etc. As such, the default behaviour that Xtext provides does not meet the needs of our envisioned solution, as we would need to customize this behaviour to close this gap. In order to perform this customization, some proficiency with Xtend is required [2]. Xtend is used to aid Xtext on processing models and expressions built under the latter, and they are normally used together. With that being said, we conclude that, although using a combination of these tools is a possible strategy, it requires proficiency with both of them, which possibly implies increased staff costs and development time.

B. MPS

MPS is a metaprogramming system and also known a language workbench that allows the development of DSLs, simultaneously creating models for such DSLs, generating code, perform model verification, define annotations and create documentation. MPS relies on a projectional editor, meaning that the Abstract Syntax Tree (AST) is changed after every input instead of having a parser that transforms a character sequence in that same tree. This brings some advantages, namely, language modularity and flexible notation capabilities.

In the scope of the present work, we will use plugins developed for the MBEDDR [3] project due to their focus in embedded system development, even though the usage of C as the only translated code is not the objective. Although this is the case, MBEDDR can be used as a set of plugins of MPS, which introduces features that bring value to our project, e.g., state machine definition, formal analysis capabilities, among others.

The plugins that compose MBEDDR contain language definitions, namely MBEDDR C and its extensions, which are languages in terms of MPS, in addition to the many libraries, utilities, views, editors, etc.

With that being said, it is pertinent to briefly describe MBEDDR architecture[4].

[2]Xtend is a flexible and expressive dialect of Java

[3]MBEDDR is an extensible C-based language and IDE for embedded software development [6]

[4]http://mbeddr.com/userguide/images/concept/architecture.png

MBEDDRs structure is comprised of 5 layers: (1) user extensions, which is where users can define languages and other domain specific features; (2) default extensions, which are the extensions already supported by MBEDDR; (3) core, which contains the bulk of MBEDDRs features like the C99 language standard, model checking features, requirement specification, documentation, etc.; (4) platform, which is the JetBrains MPS framework; (5) backend tool, which contains the basis (independent tools) for the entire tool like the C compiler, new symbolic model checker, etc.

Additionally, MBEDDRs architecture focuses on three distinct concerns, namely the implementation concern, the analysis concern and the process concern. The implementation concern addresses the development of applications based on C with the advantage that the user can extend the language and any of the existing extensions.

The analysis concern contemplates static analysis (formal verification) for some of the default extensions provided by the implementation concern. These analysis (based on symbolic model checking, SMT solving and C-level model checking) are performed by external tools, integrated in MBEDDR.

The process concern focuses on the facilities of integrating MBEDDR into development processes. These features apply to default and user-defined C extensions. It contains features such as requirements support which provides a language to describe requirements; traces that can be attached to any program element, additional arbitrary data that can be added to a requirement, among other features.

IV. A New DSL for CPS development

In this section, we introduce the ideas supporting the novel DSL that we are developing, and illustrate how it can be used to support the development of a simplified CPS application that contains the main characteristics one can expect in the modern and future generation of automotive CPS applications. It is crucial to mention that the DSL is still in a very early development phase, which means it is still subject to significant changes.

A. Example Application

With the ever growing demand for cars to be equipped with more advanced, safe, and secure Advanced Driver-Assistance Systems (ADAS), whose development costs and time-to-market have reduced, we foresee that many CPS applications will make use of high-performance computing hardware platforms equipped with several, potentially heterogeneous computing cores, as well as with GPUs specially tailored to improve the performance and energy efficiency of advanced computation features, like neural networks or image processing algorithms.

Also importantly, single OS solutions are also becoming quite limited, as complex CPS applications for the automotive sector clearly require higher degrees of isolation between components of the system since these components have different levels of criticality and security. Therefore, hypervisors that are able to manage several OSs within the same platform are gaining relevance, as they naturally enforce temporal and spacial isolation among components, and thus allow interference mitigation and provide a natural way to have components executing in an OS environment that provides support for predictable computation and operation over the physical environment (e.g., real-time operating systems satisfying AUTOSAR requirements), and for these to send data for being processed in standard Linux OSs where frameworks like the Robot Operating System (ROS) are used for coordination of components that are responsible for providing the ”intelligence” required for vehicles with advanced ADAS features, for example.

In the remainder of the paper, notably when we start introducing the concepts associated with the novel DSL that we are developing, we will use as working example an abstract instance of a system with the characteristics that we have described in the previous paragraph. In particular, we consider a CPS application comprised of three partitions (each corresponding to a guest operating system, two of which are standard Linux and the remaining one hosting a RTOS), one *hypervisor* managing those partitions, all of which running on a high-performance computing platform Nvidia Jetson TX2. One of the Linux partitions will serve as the interface with the driver, the second Linux partition hosts ROS as its most outstanding application, and the RTOS running in the third partition is assumed to be in charge of managing the sensing of, and the actuation over the physical instruments of the vehicle. The high-level of this CPS system's architecture is illustrated in Figure 1.

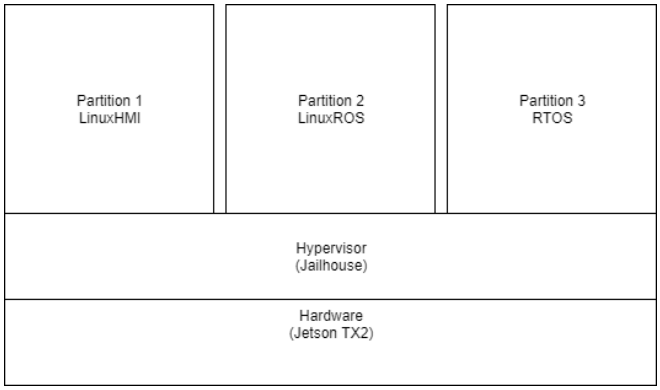

Fig. 1. Example CPS application.

B. DSL characteristics and structure

From the architecture of the above described example CPS application, we can immediately identify four different views of that application: (1) a *platform view* (PV), where the hardware platform is chosen and the set of resources that have to be used in the application; a (ii) *hypervisor view* (HV), where one needs to configure how many partitions have to be defined and how they communicate with each other (if they communicate at all); an (iii) *operating system view* (OV) where the OS to be deployed in each partition is selected, and customized to reduce resource usage and potential interference

with the applications that will be running on them; and an (iv) *application view* (AV), where the actual computational components responsible for the functionality of the system are identified, their interactions are defined, and from which verification conditions can be derived.

Each of these views contains specific information which is necessary to be known by other views in order to establish strict boundaries on the support that each view can have or can provide to the other view to which it is directly connected. For instance, the HV must provide information to the OV indicating that only a subset of operating systems is supported, which in turn impacts on the set of components that can be deployed in each of the operating systems that may be chosen. Next, we provide more details about the role of each view in our idealized DSL, as well as the associated syntax of the language while showing how it can be applied to capture the properties of our running CPS application example.

(1) – PLATFORM VIEW. The scope of the PV is on defining the hardware platform to be adopted in for the target application, as well as specifying which of its resources are allowed for usage in the upper-level views. In terms of syntax, this represents a `view` block that is an instance of a `Views.Platform` template. Since, in our working example, we are assuming a NVidia Jetson TX2 as the target hardware platform, we define a `refine` statement whose argument is the name of this embedded board. In the general case, we assume that in order to use some hardware platform, some previous work on defining the corresponding stub must be done.

Next, we start to declare the platform resources that we want to use, and those which we do not want to use. To allow the usage of a resource, we use the statement `allow` followed by the name of the resource (which is defined in the corresponding stub, and is hidden from the developer. For instance, in the example we are using, we allow the access to the quad-core ARM A57 available in the Jetson TX2 board. We also allow the usage of ethernet, wireless, serial interfaces for communication, and hdmi support. Finally, we deny the usage of all the remaining available resources via the statement `deny all`, whose intended semantics is that of disallowing access to any resource that was not marked as "allowed" in the view specification.

```
view DemoPV is Views.Platform {
  # Refines existing datasheet based
    specification for Jetson TX2
  refines JetsonTX2;
  # Allow ARM A57 cores
  allow cores = [1, 2 ,3, 4];
  # Allow Ethernet and WLAN support
  allow ethernet;
  allow wlan;
  # Allow hdmi output
  allow hdmi;
  # Allow serial connection
  allow serial;
  # Block all the rest
  deny other;
  # Define usable memory region for partitions
  region 0x000000 .. 0xFFFFFF : main;
}
```

The relevance of the PV is that it can be used to facilitate the construction of hypervisor and OS distributions, by deselecting drivers for whose corresponding resources are denies of usage, as well as associated services, thus leading to leaner distributions that occupies less memory space (which is very useful when deploying in embedded hardware platforms, which provide less computational resources when compared to regular computers); also, this information can be used to check unwanted access to memory regions that may map hardware resources, for instance, by programmers.

(2) – HYPERVISOR VIEW. The goal of the HV is to allow the selection of the hypervisor and establish its configuration in terms of partitioning and sharing of resources, according to the requirements of the target application. In the example below, which is a view that implements the `Views.Hypervisor`, we select the Jailhouse [7] hypervisor via the statement `refine Jailhouse`. We then define three partitions using the `partition` keyword, and to each of these partitions, we specify which OS is expected to be deployed in it, what is the memory region in which the OS will be mapped to execute, and which resources each partition it will use. Since Jailhouse is a strong partitioning hypervisor, i.e., each hardware resource can be associated with just one partition, we can make use of our view to verify isolation either in terms of memory regions on also guarantee that resources are not wrongly shared among partitions.

```
view DemoHyp is Views.Hypervisor {
  # Bring to scope the selected hw platform
    and its usage constraints
  import DemoPV;
  # Refine existing Jailhouse hypervisor
    support
  refines Jailhouse;
  # Specify partition details
  partition LinuxHMI with {
   OS Linux;
   #Specify the main memory region of this
    partition
   region 0x000000 .. 0xAAAAA : main;
   use cores[0], wlan, hdmi;
  }
  # Specify partition details
  partition LinuxROS with {
   OS Linux;
   region 0xBBBBBB .. 0xCCCCCC : main;
   use cores[1..2], ethernet;
  }
  # Specify partition details
  partition RTOS with {
   OS FreeRTOS;
   region 0xDDDDDD .. 0xFFFFFF : main;
   use cores[4], serial;
  }
  # Finally, we set up a set of inter-
    partition communication memory zones
  channel inter_partition_1 with {
    connect LinuxHMI and LinuxROS;
    size 10MB;
    mode read write to all;
```

```
    protocol IVHSMEM;
  }
  channel inter_partition_2 with {
    connect LinuxROS and RTOS;
    size 10MB;
    mode write to RTOS;
    mode read to LinuxROS;
    protocol IVHSMEM;
  }
}
```

Another useful specification considered in the HV is that of specifying communication mechanisms between partitions. In the case of Jailhouse, that is possible via shared memory regions that the hypervisor manages. We specify those communication facilities via `channel` blocks, which relate partitions via the `connect` statement, after which we define the size of that communication channel, and the read/write modes associated with them. Also, we assume that there may be more than one kind of protocol being used to govern the inter-partition communication channel and for that we consider the `protocol` statement, which in the case of Jailhouse is IVHSMEM.

(3) – OPERATING SYSTEM VIEW. The OV allows us to choose the concrete OS instances, according to the what has already been specified in the HV. Not only the concrete OSs are chosen, but also extra packages that need to be installed are selected and the set of services that need to start during boot time are defined. In the example below, we depict the specification of the OS that shall be deployed in the `LinuxROS` and `RTOS` partitions defined in the HV. Again, the advantages of having the OV is that of having the possibility to generate customized configuration and build scripts that lead to OS instances with reduced size, discard applications and services that are not needed, which consequently reduces the changes of unwanted interference among applications packages in the OS and the concrete applications we are targeting to deploy in the final CPS application.

```
view LinuxROSImage extends Views.OV {
 # Clone a pre-existing Ubuntu distribution
    with minimal software packages and
    services
 refines Linux.Ubuntu.Minimal;
 # State that is an instance for a particular
    partition
 deploys LinuxROS;
 # Set extra packages to be installed
 install ros,...;
 # Set application to be started at boot time
 boot { ... }
}

view RTOSImage extends Views.OV {

 refines RTOS.FreeRTOS;

 deploys RTOS;

 boot { ... }
}
```

In the above example we specify two OS images that shall be automatically built by the supporting tools of the DSL. The first one states that a minimal Ubuntu Linux distribution shall be the base, and that it will be deployed on the LinuxROS partition, installing extra packages related to ROS, and boot a set of services that are necessary. The second wrapper simply states that the base OS is FreeRTOS and that it shall be automatically built and deployed on the RTOS partition.

(4) – APPLICATION VIEW. The last view discussed in this paper is the AV. The goal of this view is similar to that of traditional development interfaces in model or DSL driven development environments. This view provides a set of predefined software components (or templates) to be refined and customized to the CPS application under development. Another aspect considered when specifying AVs is to establish verification conditions that need to be proved to ensure system correctness. To illustrate this, let us first look at the specification of two AVs which focus on a ROS component that needs to be deployed in the LinuxROS partition, and a task that shall be deployed in the RTOS partition. For the case of the ROS based application, we specify an AV that defines which topics are going to be used, and an AV that specifies the ROS application using that node.

```
view DemoTopic extends Views.AV {
  # Clone the stub for ROS topics
  refines ROS.Topics;
  # ROS bricks to be deployed on previously
  # specified LinuxROS image
  deploys LinuxROS;
  # Specify the set of topics to be considered
  topic T1 with {
    path = "path_to_ROS_topic_spec_file" ;
    type = float ;
  }
}

view DemoNode extends Views.AV {
  # Clone the stub for ROS nodes
  refines ROS.Nodes;
  # Set the target deployment OS
  deploys LinuxROS;
  # Specify the nodes
  node In_Node with {
    body "path_to_ROS_node_source_code_file";
    topics {
      T1 mode read
    };
  }
}
```

In terms of specifying topics, that is performed via the writing of `topic` blocks, where the `body` parameter defines the source file that contains the actual code that will provide the ROS specification, and the `type` is the type associated with that topic. We can use these two fields together to derive verification conditions that ensure that any access to the topic is performed using the correct data structures (a sort of static type checking at the DSL level). This is particularly useful when several developers are implementing different nodes accessing to common topics, and typing problems can

occur, so the DSL, when being processed, shall warn for those inconsistencies and deny the overall CPS application building process. In the case of the node specification, the approach is very similar, with the exception that we enforce reading/writing modes to associated topics, which can also be verified during analysis or build times of the application, using information already available in the topic's specification.

For specifying a task in our DSL, the process is very similar to the specification of ROS nodes and topics. The difference relies on the `refine` argument, which in this case is a refinement of a task in FreeRTOS, and its associated scheduling parameters, which for now we consider only the task's priority and period. Like with the other AVs specified above, that task specification must also point to the concrete file where the code implementing its functional behavior is written.

```
view DemoTask extends Views.AV {
  # Clone the stub for FreeRTOS task nodes
  refines FreeRTOS.Tasks;
  # Set the target deployment OS
  deploys RTOS;
  # Task specification
  task ReadSensor {
    body "path_to_source_code_file" ;
    priority Pr;
    period Per;
  }
}
```

Next, we turn to the specification of verification aspects of the DSL. For that, we conceived the notion of an AV whose internal components are `check` and `monitor` blocks, where the former establishes formal specifications that shall be verified statically, whereas the latter considers a mechanism that generates monitors that will be coupled with the system and verify its properties during run-time. What we idealized is that the tools that will be built to support development of CPS application with our DSL shall have specific blocks that allow users to define formal specifications and these specifications are checked against the associated verification tools. In the example that follows, we assume some Linear Temporal Logic (LTL) verification tool to check one property, and a Restricted Metric Temporal Logic with Durations (RMTLD) monitor generation framework that will build a monitor that has the execution semantics of the formula specified. Intuitively, the LTL specification states that it is always true that when the ROS node `In_Node` reads from topic `T1` it satisfies property ϕ then, somewhere in the future it must write something to the standard output that satisfies the property φ. Similarly, the specification of the `monitor` block intuitively means that the node `In_Node` reading from topic `T1` cannot take more than 10 time units to finish.

```
view NodeVerification extends Views.AV {
  # Load all ROS verification stub
  refines ROS.Verification.*;
  # Include nodes and topics
  include DemoNode;
  include DemoTopic;
  # Intra Node Verification (static)
  check {
   module LTL {
    [G](if In_Node.Read.T1 sat φ
        then [F](In_Node.Write.stdout sat φ));
   }
  }
  # Property monitoring
  monitor {
   module RMTLD {
    # Check that a message publication takes
    no
    # more than 10 time units
    [G_{<10}](In_Node.Read.T1 sat true);
  }
}
```

V. Conclusions

In this paper we discussed current integration solutions and our approach to develop a DSL to smoothen the integration of different Cyber-Physical Systems into a single solution. We've also shown some key concept realizations in an example DSL definition, namely, inter and intra view communication concepts, deployment configuration concepts and formal verification concepts. For future work, we intend to fully develop and formally verify an integration tool using this approach, targeting the automotive industry.

Acknowledgment

This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UID/CEC/04234); also by the Norte Portugal Regional Operational Programme (NORTE 2020) under the Portugal 2020 Partnership Agreement, through the European Regional Development Fund (ERDF) and also by national funds through the FCT, within project NORTE-01-0145-FEDER-028550 (REASSURE).

References

[1] R. Land and I. Crnkovic, "Software systems integration and architectural analysis - a case study," 10 2003, pp. 338– 347.

[2] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming dr. frankenstein: Contract-based design for cyber-physical systems," *European journal of control*, vol. 18, no. 3, pp. 217–238, 2012.

[3] A. A. Shah, D. Schaefer, and C. J. Paredis, "Enabling multi-view modeling with sysml profiles and model transformations," in *International Conference on Product Lifecycle Management*. Citeseer, 2009, pp. 527–538.

[4] J. Bryans, J. Fitzgerald, R. Payne, A. Miyazawa, and K. Kristensen, "Sysml contracts for systems of systems," in *2014 9th International Conference on System of Systems Engineering (SOSE)*. IEEE, 2014, pp. 73–78.

[5] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2010, pp. 307–309.

[6] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "mbeddr: an extensible c-based programming language and ide for embedded systems," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 2012, pp. 121–140.

[7] M. Baryshnikov, "Jailhouse hypervisor," B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2016.