MDE-based Sensor Management and Verification for a Self-Driving Miniature Vehicle

Md Abdullah Al Mamun, Christian Berger, and Jörgen Hansson
Chalmers | University of Gothenburg, Sweden
Department of Computer Science and Engineering
[abdullah.mamun, christian.berger, jorgen.hansson]@chalmers.se

ABSTRACT
Innovations for today's vehicle functions are mainly driven by software. They realize comfort systems like automated parking but also safety systems where sensors are continuously monitoring the vehicle's surroundings to brake autonomously for avoiding collisions with cars, pedestrians, or bicyclists. In simulation environments, various traffic situations with alternative sensor setups are imitated before testing them on prototypical cars. In this paper, we are presenting an MDE approach for managing different sensor setups in a cyber-physical system development environment to leverage automated model verification, support system testing, and enable code generation. For example, the models are used as the single point of truth to configure and generate sensor setups for system validations in a 3D simulation environment. After their validation, a considered sensor configuration is transformed into a constraint-satisfaction model to be solved by the logical programming language Prolog. Based on this transformation, the conformance to the embedded system specification is formally verified and possible pin assignments, for how to connect the required sensors are calculated. The approach was validated during the development of a self-driving miniature vehicle using an STM32F4-based embedded system running the real-time operating system ChibiOS as the software/hardware interface to the sensors and actors.

Categories and Subject Descriptors
D.2.4 [**Software/Program Verification**]: Model Checking; D.1.6 [**Logic Programming**]: Prolog

General Terms
Formal Verification for Sensor Layout Configurations with Prolog in a Cyber-Physical Development Environment

Keywords
Model-Driven Engineering (MDE), Sensor, Simulation, Formal Verification, Prolog, Cyber-Physical Systems

1. INTRODUCTION AND MOTIVATION
Comfort and safety systems from today's vehicles are powered by software, which continuously process data from the vehicle's surroundings perceived by various sensors. However, identifying, experimenting, and validating various sensor configurations to find the layout, which serves the intended use cases in the best way, is a challenging, time-consuming, and error-prone task for engineers. During the realization of a possible sensor layout, different sensor setups need to be evaluated regarding their mounting position, orientation, and detection characteristics like opening angle or viewing distance within a virtual environment before testing them in prototypical cars [6, 9, 10].

Interfacing restrictions from an embedded system, which is processing the incoming sensor readings, like pin assignment must be kept in mind. As a running example in this article, we use an STM32F4 Discovery Board as the software's gateway to the sensors and actors of our self-driving miniature vehicle [7]. The board's microprocessor is an ARM-based system [18], which has a low power consumption profile, 80 connection pins, and is powerful enough to interface with various sensors as shown in Fig. 1. Different pin assignments are possible but constrained by the target hardware environment (STM32F4 Discovery Board in our case).

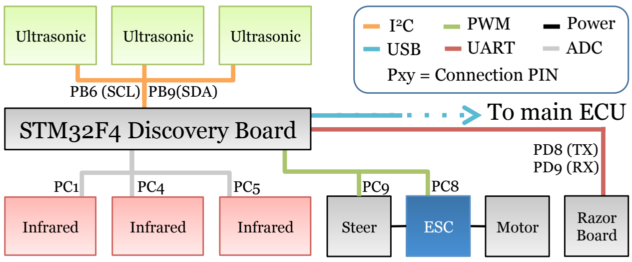

Figure 1: Possible pin assignment to interface with sensors and actors for the STM32F4 Discovery Board.

In this paper, we outline an MDE-based sensor management approach, which relies on a domain-specific language (DSL) to describe the domain model of possible sensor configurations. We use its instances for their validation in a 3D simulation environment before verifying them for the intended target hardware environment with Prolog.

The benefits of using an MDE approach are that models are considered to be the single point of truth throughout the development. Furthermore, the models are defined and managed based on the actual domain needs; thus, the DSL remains in the best case very close to the actual use cases to be supported without unnecessary complications. Additionally, it defines sound engineering approaches to the definitions of the models and their accompanying transformations for model to model and model to code.

The rest of the paper is structured as follows: In Sec. 2, we introduce our sensor management language and its design rationales. Afterwards, we outline two use cases in Sec. 3, where an instance of the meta-model is transformed into a configuration for our simulation environment [8] to carry out experiments with the chosen sensor configuration. Once the experiments' results have been approved, the configuration is transformed into a constraint-satisfaction problem (CSP) for the logic programming language Prolog to formally verify its conformance to the possibilities of the target hardware environment. The article closes with a discussion of related work (Sec. 4) and a conclusion.

2. SENSOR MANAGEMENT LANGUAGE

We used the Eclipse Modeling Framework (EMF) [2] to capture the domain knowledge, abstract syntax, and static semantics for our DSL. Our main design drivers were simplicity and extensibility of the meta-model. Our meta-model is easily extensible in the sense that adding new sensors, sensor properties, and configuration properties would require minimal modification in the existing meta-model. For example, adding a new sensor would require just adding a new *EClass* with an inheritance link with the *Sensor* abstract *EClass* without requiring modifying any other references in the meta-model.

UnitType enumerations are introduced to avoid the occurrence of invalid/mismatched assumptions related to physical units(cf. [4]). Further enumerations like *SensorClass*, *ExecutionBoard*, and *OperatingSystem* are introduced to allow adding static semantics to verify the correct usage of COTS components like a concrete ultrasonic sensor or an OS.

2.1 Domain Meta-Model

The meta-model defining the abstract syntax of the DSL is shown in Fig. 2. Class *Vehicle* is the top-level semantic element of the meta-model that maintains relationships with the rest of the classes in the meta-model. The class *Sensor* represents physical sensors, which can be of type *Infrared* or *Ultrasonic*. Hereby, *SensorClass* captures the identity of a concrete sensor given by its manufacturer like "SRF08" from Devantech, which is used by our self-driving miniature vehicle. This per-sensor annotation introduces further restrictions that need to be considered during the code generation stage. A brief description of the available *Properties* is listed in the following.

- *rotZ*: Rotation of the sensor around the Z-axis.
- *translation*: Position of the sensor in 3-dimensional axis(X;Y;Z) with respect to the vehicle's center.
- *angleFOV*: The field of view of the sensor in *AngleUnit*.
- *distanceFOV*: The length of the field of view at its maximum viewable position in *DistanceUnit*.
- *clampDistance*: Effective distance of the sensor to which the physically possible viewing distance is reduced e.g. to reduce unwanted noise.
- *address*: Address used by a sensor, e.g. for the I^2C bus.
- *pinConnection*: Physical pin identifier of the execution platform to which the sensor is connected. This information is computed automatically depending on the target hardware environment (cf. Sec. 3.3.1).
- *connectionType*: Type of standard connection used by the sensor (e.g., ADC, I^2C, ...). This information is used to verify a desired configuration as described in Sec. 3.3.1.

PropertyCategory indicates whether a property or configuration property is related to the vehicle simulation (*Cyber*), its run-time realization (*Physical*), or both (*CyberPhysical*). This annotation is used during the code generation stage. *ExecutionPlatform* indicates the physical execution platform (in our case: STM32F4 DiscoveryBoard) to which a sensor shall be connected and the *ApplicationPlatform* captures the software/hardware interface (in our case: Chibi/OS).

2.2 Ensuring Static Semantics

We use the Object Constraint Language (OCL) to verify static semantics within the meta-model. General constraints of the meta-model are checked no matter which *SensorClass*, *ExecutionPlatform*, etc. is used in the model. An example of such a constraint is "sensors of a certain *SensorClass* must have exactly one *SensorConfiguration*". Other constraints which are related to specific COTS components used in our miniature vehicle are listed in the following:

- *SRF08*: *SensorClass* "SRF08" can only be associated with an *Ultrasonic* sensor.
- *SRF08*: The *connectionType* property of "SRF08" must use I^2C bus as *ConnType*.
- *STM32F4*: The maximum number of *Ultrasonic* sensors is 48 in case of using *STM32F4* as the target hardware environment because three I^2C buses are available each hosting up to 16 devices.

3. MODEL TRANSFORMATIONS

In the following, we describe how model transformations of our meta-model is applied to serve our two subsequent use cases.

Use Case A: This use case involves experiments with different sensor layouts within our 3D simulation environment. It requires modeling of sensors with associated properties like *rotZ, clampDistance, angleFOV, distanceFOV*, and *translation* to validate a sensor layout. Use case *A* does not require adding target information like concrete pin assignment.

Use Case B: This use case verifies if the selected sensor layout from use case A can be realized with the regarded target hardware environment. More specifically, we want to find a possible pin assignment for all sensors in a given layout.

3.1 Model-to-Text Transformation

To enable our use cases, the Model-To-Text (M2T) transformation is realized with Acceleo [1], an open source code generation framework based on the MOF Model to Text Language (MTL) standard. It can be used with any EMF based models to generate any type of code. Fig. 3 gives an overview of the M2T process for an instance model.

As depicted by Fig. 3, we generate configurations for the simulation environment from an instance model as described in use case A. Requests as required for our Prolog-based verification approach (cf. Sec. 3.3.1) for use case B are also derived from an instance.

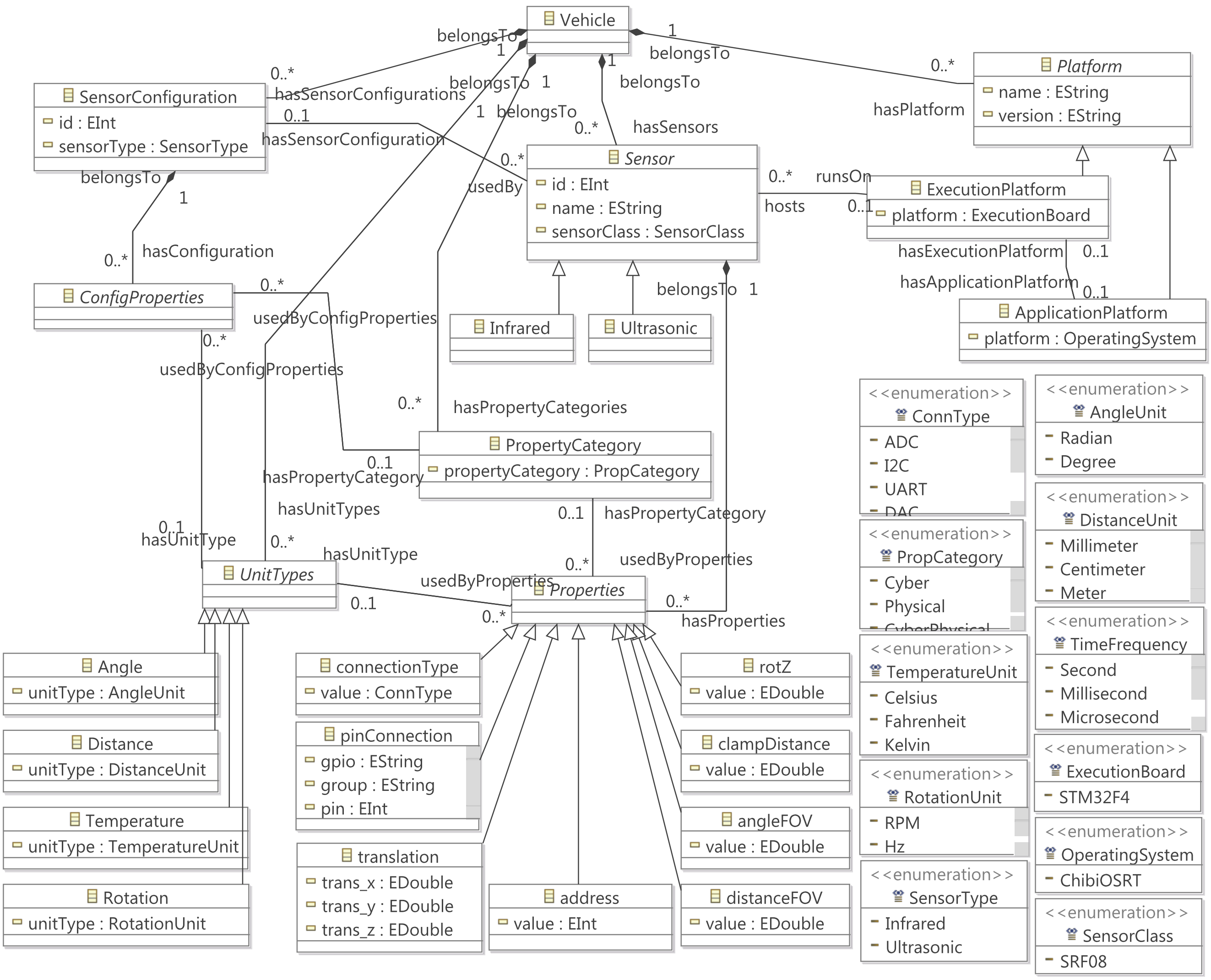

Figure 2: Meta-model of our DSL for managing sensor configurations.

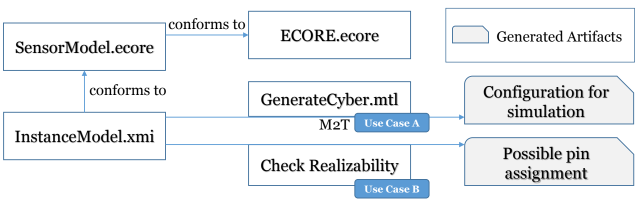

Figure 3: Overview of the M2T transformation process

3.2 Use Case A: Validating a Sensor Layout

We have developed a sensor layout with two infrared and three ultrasonic sensors to realize an automated parking scenario with the self-driving miniature vehicle. Once, the layout configuration is complete, we generate configuration code as shown in Fig. 4 for the 3D simulation environment. To save space, in Fig. 4, we have shown only one infrared sensor from the layout and removed the comments from the generated code. Fig. 5 visualizes the same sensor layout for the automated parking scenario in the simulation environment.

```
irus.numberOfSensors = 5
irus.showPolygons = 1
...
irus.sensor2.id = 2
irus.sensor2.name = Infrared_RearRight
irus.sensor2.rotZ = -90.0
irus.sensor2.translation = (-1.0;-1.0;0.0)
irus.sensor2.angleFOV = 5.0
irus.sensor2.distanceFOV = 3.0
irus.sensor2.clampDistance = 2.9
irus.sensor2.showFOV = 1
...
```

Figure 4: Generated sensor layout consisting one infrared sensor for the 3D simulation environment.

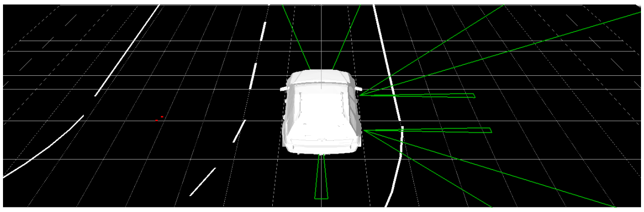

Figure 5: Configured sensor layout in a vehicle in the 3D simulation environment.

3.3 Use Case B: Verifying a Sensor Layout for a Target Hardware

After validating a suitable sensor layout in the simulation environment, the developer needs to find a possible pin assignment for the embedded system of interest–in our case for the STM32F4 Discovery Board which is a constraint satisfaction problem (CSP) as elaborated in section 3.3.3. To solve this CSP, we use Prolog to verify a given sensor configuration. In the following, we describe our approach, code generation to Prolog, and results from our case study.

3.3.1 Verification Approach

Our verification approach utilizes a directed graph $G = (N, E, A)$ as shown in Fig. 6 to describe allowed configurations for a specific embedded system. Hereby, a node n denotes a specific pin of an embedded system and its incoming edge e with its annotation the specific usage for that pin (e.g. analog input). Set A contains all possible edge annotations, like $\{$analog, I^{2}C, $\ldots\}$.

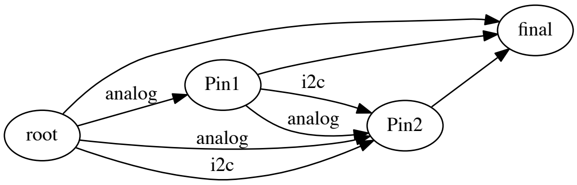

Figure 6: Simplified example for a possible configuration graph with six possible paths from n_0 to $n_{\mathbf{final}}$.

A path p from the root node n_{root} to the final node n_{final} describes a concrete configuration for the embedded system. In Fig. 6, the following six paths with their corresponding configurations are described:

1. $n_{\text{root}} \rightarrow n_{\text{final}}$: trivial path.
2. $n_{\text{root}} \rightarrow n_{\text{pin\_1}} \rightarrow n_{\text{final}}$: analog.
3. $n_{\text{root}} \rightarrow n_{\text{pin\_2}} \rightarrow n_{\text{final}}$: analog.
4. $n_{\text{root}} \rightarrow n_{\text{pin\_2}} \rightarrow n_{\text{final}}$: I^{2}C.
5. $n_{\text{root}} \rightarrow n_{\text{pin\_1}} \rightarrow n_{\text{pin\_2}} \rightarrow n_{\text{final}}$: analog, analog.
6. $n_{\text{root}} \rightarrow n_{\text{pin\_1}} \rightarrow n_{\text{pin\_2}} \rightarrow n_{\text{final}}$: analog, I^{2}C.

To match a desired sensor configuration from the simulation environment, we need to find a possible path p_{match} from n_{root} to n_{final}, which has the required annotations alongside its edges. Hereby, we use Prolog to model possible configuration specifications as *facts*, their graph-oriented traversal as *inference*, and a match request as a CSP *query* to be solved.

```
edge(root, pin1, analog).
edge(root, pin2, analog).
edge(root, pin2, i2c).
edge(pin1, pin2, analog).
edge(pin1, pin2, i2c).
edge(pin1, final, -).
edge(pin2, final, -).
```

Figure 7: Configuration graph in Prolog.

```
path(From, To, Path, RequiredProperties) :-
    traverse(From, To, [From], P, RequiredProperties),
    length(RequiredProperties, LenRequiredProperties),
    length(P, LenP),
    LenRequiredProperties =:= (LenP - 2),
    reverse(P, Path).

traverse(From, To, Path, [To|Path], _) :-
    edge(From, To, _).

traverse(From, To, Visited, P, RequiredProperties) :-
    edge(From, Other, Annotation),
    not(Other == To),
    not(member(Other, Visited)),
    member(Annotation, RequiredProperties),
    traverse(Other, To, [Other|Visited], P,
             RequiredProperties).
```

Figure 8: Graph traversal as Prolog source is the *inference* engine to find paths.

3.3.2 Code Generation to Prolog

In Fig. 7, the configuration space represented by the graph depicted by Fig. 6 is shown as *facts* in Prolog.

The resulting graph is the basis for the traversal algorithm shown in Fig. 8. Hereby, `path` expects four arguments, where the last one contains a list of required edge annotations. To only return those paths, which fulfill entirely the required configuration, its length must match the length of the argument `RequiredProperties`. The second rule describes the case where a direct connection between the nodes n_{From} and n_{To} has been found while the last rule describes the transitive case to start the recursive graph traversal.

```
verify(RequiredProperties, Path) :-
    path(root, final, Path, RequiredProperties).
```

Figure 9: User interface to solve the verification *query* in Prolog.

In Fig. 9, the usage is shown. Hereby, the rule `verify` is called with a desired configuration set in the first parameter like `verify([analog, i2c], Path)`. Prolog replies either with `false` or with a potential pin assignment like `Path = [root, pin1, pin2, final]` .

3.3.3 Case Study Results

For our self-driving miniature vehicle, we modeled a configuration graph for the STM32F4 Discovery Board up to a graph with a height of six resulting in 1,388 *facts* for Prolog. Thus, it is possible to check configurations, which contain up to five different connection requirements. In Table 1, the results for different verification runs are depicted: In the first column, the actual length of the configuration to be verified is shown, the second column indicates whether the given configuration was valid or not, and the last column shows the actual execution time on a 1.8GHz Intel Core i7 with 4GB RAM running Mac OS X 10.8.4.

Configuration length	Valid	Execution time
1	true	0.00s
1	false	0.00s
2	true	0.01s
2	false	0.56s
3	true	0.57s
3	false	0.69s
4	true	112.61s
4	false	29.24s
5	true	36,044.65s
5	false	7,168.48s

Table 1: Results of verifying configurations in with a specification graph of depth six.

As shown by the table 1, Prolog detected all given invalid configurations. Interestingly, the longer the specification the relatively less time was spent to detect invalid configurations. However, the problem to find a path with certain characteristics (in this case, matching a given pre-defined configuration) is not solvable in polynomial time specifically for longer configuration lengths, which can be seen by the exponentially increasing execution times in the last column.

4. RELATED WORK

The use of sensors spans from simple water heater to space shuttles. The application areas of wireless sensor networks (WSN) e.g., sea observatory, weather forecasting, air/water monitoring etc. highly depend on different quality factors of the sensor nodes. Thus, there is a clear need for an advanced sensor management and these areas have advanced on this direction.

SensorML [12], a part of Open Geospatial Consortium (OGC), is a generic data model expressed using UML that captures classes and associations common to all sensors. OGC PUCK (Programmable Underwater Connector with Knowledge)[14] is a standard command protocol and focuses on the automated configuration and installation of sensors used in devices. When the device is connected to the host computer, PUCK protocol allows data transfer between the involved parties. IEEE 1451 TEDS (Transducer Electronic Data Sheet) [3] is a set of smart transducer standard, which has conceptual similarity with the OGC PUCK. IEEE 1451 TEDS provides a common set of interfaces so that transducers data can be accessed when they are connected to the system through wired or wireless networks.

However, the mentioned approaches do not precisely fulfill our needs. From an architectural point of view these approaches deal with distributed systems while we are focused on a shared memory system. Our approach does not require any standardized protocols or set of interfaces to read data from different sensor nodes since reading data in our case depends on the *connectionType* of sensor *Properties*. Moreover, we are focusing on the domain of self-driving vehicles inspired by [5] with a specific set of COTS components. Since our goal is to achieve a domain specific and semantically rich model that we can use to generate code for both simulation and execution environment, we have designed a DSL what would precisely fulfill our needs.

An approach to visually assist developers in managing configurations for middleware and simulations on the example of autonomous vehicles is presented by the authors of [16]. They focus on the "correct-by-construction" principle to configure both elements of a cyber-physical system development process. As an extension to their work, we focus additionally on the verification of a considered configuration for the target hardware environment by utilizing automatically generated models for CSP solving with Prolog.

The use case as described in Sec. 3.3.1 is also supported by commercial tools. The manufacturer of the embedded system used on our self-driving miniature vehicles provides the tool MicroXplorer to support the pin configuration and code generation for their low-level system library [19]. However, that tool has two main drawbacks compared to our approach: Firstly, they require the user to select the pin layout to be used for the peripherals being handled by the STM32F4 microprocessor; thus, the user needs to work in a solution-driven instead of requirement-driven manner (i.e. specifying *what* to support and not *how* to use the microprocessor). Secondly and more importantly, the proprietary tool does not support our target realtime operating system Chibi/OS, which we aim to use to be platform-independent for future use cases and hardware environments.

An alternative to the aforementioned tool would be CooCox CoOS with their own toolchain to support the microprocessor configuration and code generation [13]. However, as at the time of writing of this article, they do not support our intended hardware environment. Furthermore, both alternatives do not support the merging, concatenation, or difference calculation of two or more configurations for the embedded system due to their nature of being a graphical tool. With our textual approach, those aspects can be realized easily.

An approach pointing in a related direction for applying logic programming to solve pin assignment and configuration is described by the authors of [15]. However, their work appears to be at a very early stage so far as they neither describe any successfully carried out experiments nor discuss any results.

The work by Berlier and McCollum [11] describes a self-implemented backtracking algorithm with a heuristic extension. Thus, they limit the state space, which needs to be explored to find a solution for a possible pin assignment. Compared to our approach, they do not discuss the challenge how to provide the formal hardware specification as input to their algorithm as well as how to merge, concatenate, or calculate the difference between several configurations.

In previous work, it was demonstrated how to systematically enumerate all potential system stimuli vectors as a formal approach to requirements engineering to generate test cases [17]. Therefore, it is required to define the required system boundaries in terms of input and output vectors. Our approach outlined here serves as an extension to assist an engineer to design a system, which exhibits just enough inputs and outputs to fulfill a given use case while considering hardware limitations already in the design space exploration phase.

5. CONCLUSION AND OUTLOOK

In this article, we have outlined a DSL to manage different sensor configurations for a self-driving miniature vehicle. The MDE approach bridges between the cyber and the physical system development environment. For the former, the meta-model supports the configuration process for using a 3D simulation environment to evaluate different sensor layouts. The latter deals with realizing a chosen sensor configuration within a prototypical hardware environment–a self-driving miniature vehicle in our case.

While the former case tries to preserve the design freedom during the evaluation process, the latter must consider constraints from the intended target hardware environment. We outlined an approach to derive a CSP model for Prolog from our meta-model, which we used to formally verify the chosen sensor configuration's conformance to the hardware specification of the target embedded system. The advantage of the outlined verification approach is the intuitive and highly compact representation; thus, concatenation, merging, or the difference calculation of two or more given configurations can be easily carried out. Though, future work needs to be done in the area of optimizing the graph-based representation of the hardware specification to reduce the computation time for verifying a given configuration. Our vision also spans toward finding an optimal sensor layout from a list of given scenarios and considering the cyber-physical aspects of the placement of the sensors.

Acknowledgments

The authors would like to thank Dr. Matthias Tichy for his comments and suggestions about the DSL meta-model.

6. REFERENCES

[1] Acceleo. http://projects.eclipse.org/projects/modeling.m2t.acceleo.

[2] Eclipse Modeling - EMF. http://www.eclipse.org/modeling/emf/?project=emf.

[3] IEEE standard for a smart transducer interface for sensors and actuators wireless communication protocols and transducer electronic data sheet (TEDS) formats. *IEEE Std 1451.5-2007*, pages C1–236, 2007.

[4] M. A. Al-Mamum and J. Hansson. Review and challenges of assumptions in software development. In *Proc. of the Second Analytic Virtual Integration of Cyber-Physical Systems Workshop (AVICPS)*, 2011.

[5] C. Basarke, C. Berger, K. Berger, K. Cornelsen, M. Doering, J. Effertz, T. Form, T. Gülke, F. Graefe, P. Hecker, K. Homeier, F. Klose, C. Lipski, M. Magnor, J. Morgenroth, T. Nothdurft, S. Ohl, F. W. Rauskolb, B. Rumpe, W. Schumacher, J. M. Wille, and L. Wolf. Team CarOLO - Technical Paper. Informatik-Bericht 2008-07, Technische Universität Braunschweig, Braunschweig, Germany, Oct. 2008.

[6] C. Berger. From Autonomous Vehicles to Safer Cars: Selected Challenges for the Software Engineering. In F. Ortmeier and P. Daniel, editors, *Proceedings of the SAFECOMP 2012 Workshops, LNCS 7613*, pages 180–189, Magdeburg, Germany, Sept. 2012. Springer-Verlag Berlin Heidelberg.

[7] C. Berger, M. A. Al Mamun, and J. Hansson. COTS-Architecture with a Real-Time OS for a Self-Driving Miniature Vehicle. In E. Schiller and H. Lönn, editors, *Proceedings of the 2nd Workshop on Architecting Safety in Collaborative Mobile Systems (ASCoMS)*, Toulouse, France, Sept. 2013.

[8] C. Berger, M. Chaudron, R. Heldal, O. Landsiedel, and E. M. Schiller. Model-based, Composable Simulation for the Development of Autonomous Miniature Vehicles. In *Proceedings of the SCS/IEEE Symposium on Theory of Modeling and Simulation*, San Diego, CA, USA, Apr. 2013.

[9] C. Berger and B. Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In U. Goltz, M. Magnor, H.-J. Appelrath, H. K. Matthies, W.-T. Balke, and L. Wolf, editors, *Proceedings of the INFORMATIK 2012*, pages 789–798, Braunschweig, Germany, Sept. 2012.

[10] C. Berger and B. Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer-Verlag, London, UK, 2012.

[11] J. A. Berlier and J. M. McCollum. A Constraint Satisfaction Algorithm for Microcontroller Selection and Pin Assignment. In *Proceedings of the 2010 IEEE SoutheastCon*, pages 348–351, Concord, NC, Mar. 2010.

[12] M. Botts and A. Robin. OpenGIS sensor model language (SensorML) implementation specification. OpenGIS Implementation Specification OGC 07-000, Open Geospatial Consortium Inc., 2007.

[13] CooCox. CoSmart. http://www.coocox.org/CoSmart.html, Aug. 2013.

[14] K. L. Headley, D. Davis, D. Edgington, L. McBride, T. C. O'Reilly, and M. Risi. Managing Sensor Network Configuration and Metadata in Ocean Observatories Using Instrument Puck. In *Proceedings of the 3rd International Workshop on Scientific Use of Submarine Cables and Related Technologies*, pages 67–70, 2003.

[15] B. Joshi, F. M. Rizwan, and R. Shettar. MICROCONTROLLER PIN CONFIGURATION TOOL. *International Journal on Computer Science and Engineering*, 4(05):886–891, 2012.

[16] A. Schuster and J. Sprinkle. Synthesizing Executable Simulations from Structural Models of Component-Based Systems. In *Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling*, volume 21, pages 1–10, 2009.

[17] S. Siegl, K.-S. Hielscher, R. German, and C. Berger. Formal Specification and Systematic Model-Driven Testing of Embedded Automotive Systems. In *Proceedings of the Conference on Design, Automation, and Test in Europe*, pages 1530–1591, Grenoble, France, Mar. 2011. European Design and Automation Association.

[18] STMicroelectronics. Discovery kit for STM32F407/417 line. http://goo.gl/hs7X28, Aug. 2013.

[19] STMicroelectronics. MicroXplorerMCU graphical configuration tool. http://goo.gl/3UUgdh, Aug. 2013.