CPSL: A Domain-Specific Language for Modelling the Behaviour of Cyber-Physical Systems

HAROUN MANGAL, University of Twente, The Netherlands

ABSTRACT

Cyber-physical systems (CPS) are complex machines consisting of many physical and computational parts. These machines have numerous challenges when it comes to their maintenance due to the many disciplines of expertise needed for their understanding. Project Zorro, a multi-institutional research initiative, has been set up with the task of reducing downtime in CPS. In this paper, we propose a novel Domain Specific Language (DSL) for modelling the behaviour of a CPS called Cyber-Physical Systems Language (CSPL). CSPL infers which computational tasks are affected by component failures, thus facilitating intelligent diagnostics and reducing downtime. This is done by specifying the behaviour of the CPS in terms of the parts needed to perform computational tasks performed by the system. We primarily focus on the design of CPSL and showcase its functionality via an example of a smart traffic light system.

KEYWORDS

Cyber-physical systems, domain-specific language, model-based system engineering, ontology

1 MOTIVATION

Cyber-physical systems (CPS) are integrations of computation with physical processes. Embedded computers and networks monitor and control the physical processes with sensors and actuators, with feedback loops where values received from physical processes affect computations and vice versa. Sensors are parts of the system which monitor conditions and signals when a change occurs, while actuators receive signals and perform actions. For example, a pressure sensor is a sensor since it measures pressure differences and an LED screen is an actuator since it displays the values it receives. CPS can perform computational tasks or communicate with other CPS via the cloud. Thus, CPS are an integration of the physical and the cyber world [14]. Examples of CPS are smart grids, autonomous vehicles, and medical monitoring [12].

For a CPS to enable seamless integration between the physical and cyber worlds, the physical events must be reflected in the cyber world, and the computational decisions taken by the cyber world need to be communicated to the physical world. Both these actions need to be accurately performed and in a timely manner. Thus, CPS must coordinate between embedded systems which themselves are heterogeneous systems, consisting of computing devices and distributed sensors and actuators [12]. Therefore, modelling the behaviour of a CPS is not simple.

Due to the diversity of the components of a CPS, maintenance is complex, and reducing downtime is not a trivial task [11]. Project Zorro, a multi-institutional research project, was set up aimed at reducing the downtime of complex systems, specifically CPS [2]. By utilising intelligent diagnostics, as opposed to traditional human-based diagnosis, anomalies can be detected and related to potential root causes more precisely and faster. Zorro is composed of multiple work packages where this research will focus on work package 4, model-based systems engineering [2]. This research will do so by creating a grammar of a Domain Specific Language (DSL).

In this paper, we propose Cyber-Physical Systems Language (CPSL), a DSL for modelling the behaviour of a CPS. CSPL is used to measure the impact of failure of a part of the system by looking at how the behaviour of the system changes. Firstly, the necessary background is introduced. Then, the relevant parts of CPS are captured into an ontology model. Finally, the grammar and syntax of CSPL are shown and the functionality of the proposed DSL is showcased via an example with a smart traffic light system.

2 BACKGROUND

2.1 Maintenance

Buksh and Stipovanic mention four types of maintenance in their paper: preventive maintenance (PM), condition-based maintenance (CMB), predictive maintenance (PdM), and reactive maintenance (RM) which can be seen in Figure 1 [4]. PM is a type of maintenance that is done too early leading to parts of the machine not reaching their full economic potential while RM is a run-to-failure type of policy where the system will incur downtime. CMB or PdM are preferred strategies due to the substitution of a part being done close to its point of failure while still being early enough to not incur downtime. CMB is similar to PdM, however, they differ in two key ways. With PdM continuous monitoring via edge computing and sensors is performed while with CMB regular inspections are done. With the PdM strategy, sensor data is analyzed with machine learning models while with the CMB approach, structural-specific models are used.

As part of this project, we have interviewed a product manager from Philips, one of the industry partners of Zorro. In this interview, the product manager shared details about the machine learning models used at Philips to predict the failure of a part of a CPS. These models get various data about each part of the CPS such as the temperature, voltage, software logs, etc. The machine learning model uses these inputs to determine with what probability the part goes down. These data points are monitored continuously. Thus, Philips

TScIT 41, July 5, 2024, Enschede, The Netherlands

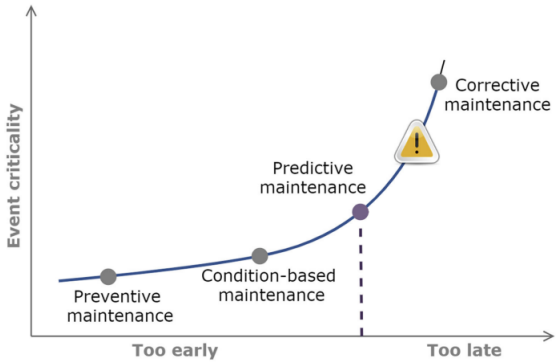

Fig. 1. Buksh and Stiponvic's four types of maintenance [4]. The yellow triangle denotes a point of failure of a part in a system.

uses a PdM strategy. The goal of the DSL proposed in the paper is, therefore, to measure the impact of the failure of a part after describing the behaviour of the system. The language should infer whether the behaviour of the system changes based on input given by the user.

2.2 Domain Specific Languages

A Domain Specific Language (DSL) is a computer programming language of limited expressiveness focused on a particular domain, rather than a general-purpose language like Java [6]. Examples of domain-specific languages are Graphviz [7] for producing graphical renderings of graphs, CSS [3] for styling web pages, and SQL [5] for database queries. The benefit of using a DSL over a general-purpose language is that it boosts the productivity of engineers by providing a clearer intent of the part of the system that it is modelled [6, 16, 24]. This clear intent can, for example, be captured in a declarative model where it is specified how things are, rather than what to do as is the case with imperative languages.

A distinction can be made between an internal DSL and an external DSL [6]. An internal DSL or embedded DSL is as an extension of another programming language [20]. This allows the DSL to integrate seamlessly within the host language. This gives the benefit of having the familiarity and tooling of the host language, such as IDE support, while still working with a language that is tailored to the needs of the specific task. The downside of internal DSL is that the syntax of the DSL is constrained to that of the host language. An example of an internal DSL is JMock, a mocking library in Java primarily used for unit testing [6]. An external DSL is a stand-alone language which allows for complete freedom when designing the syntax of the language. The downside of an external language is that it comes with the cost of having to build and maintain a new language. In addition, the developers using the language also need to learn this new language. An example of an external DSL is Graphiz [7].

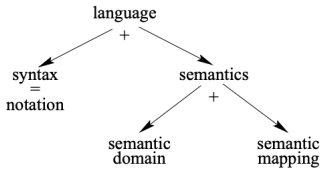

Fig. 2. Harel and Rumpe's structure of a language [10].

The development of a DSL can be broken down into 5 phases: decision, analysis, design, implementation, and deployment [16]. In the decision phases, the decision to make the time, effort, and financial investment into making a DSL is made. The analysis part focuses on gathering domain knowledge in addition to identifying the problem domain. In the design part, the DSL syntax needs to be determined. The implementation part is about transforming the grammar and syntax made in the previous phase into an actual executable DSL using a compiler, interpreter, etc. Although the paper mentions the deployment phase, it does not offer a concrete definition of it. The paper by Zaystev [29] does mention including proper tooling like IDE support. In this research paper, the focus will be only on the analysis, and design of a DSL for modelling the behaviour of a CPS due to the scope of the project.

3 DOMAIN ANALYSIS

The creation of a DSL starts by analysing the relevant domain. This domain must be captured by the language vocabulary of the DSL such that all domain constructs can be expressed. We can split up a language into its syntax and its semantics. The syntax is the symbols used in expressions of the language. The semantics of the language is the meaning behind the syntax [10]. For example, consider the syntax of the SQL statement: `SELECT * FROM USERS;` The semantics, the meaning, behind the statement is: "select all attributes of all rows from the users table"[21]. The semantics of a language can in turn be split into the semantic domain and the semantic mapping from the syntax to the semantic domain. The semantics domain denotes all different kinds of meanings that can be created with a language. Thus, the semantic domain is an abstraction of reality describing all important aspects of the system that will be modelled in the DSL. The full structural tree of a language as defined by Harel and Rumpe can be seen in Figure 2. This meaning gets mapped to by the semantic mapping via expression created in the language's syntax [10].

The semantic domain can be modelled using an ontology. Within the context of model-based engineering, an ontology is a representation of domain knowledge [22]. Generally speaking, an ontology is denoted with (domain) concepts and the relationships between these concepts [9]. Numerous papers have been written about ssing an ontology for the development of a DSL. To name a few, Lyado et al. describe a framework for developing DSLs by letting domain experts develop an ontology upon which DSL developers will base

the language on [15]. In the paper by Tairas et al., an ontology for air traffic communication is constructed and a subsequent context-free grammar is proposed for a DSL design [23]. Utilin and Babkin discuss the evolution of an ontology and its DSL by adding new rules to the DSL which subsequently also adds new concepts and relationships to the ontology [25].

Four different kinds of anomalies can occur when mapping an ontology to a construct (which in this paper is a DSL): construct deficiency, construct overload, construct redundancy, and construct excess [17].

- *Construct deficiency* means there is no construct for an ontological concept.
- *Construct overload* is when a single notation maps to multiple ontological concepts.
- *Construct redundancy* is when multiple notations map to the same concept.
- *Construct excess* is when a notation construct does not map to an ontological construct.

In case there is a construct deficiency, then the DSL is said to be ontologically incomplete. If any of the three other cases occur, then the DSL is ontologically unclear [17]. The goal of a good DSL is to have a one-to-one mapping from ontological concepts to the language vocabulary.

There are different kinds of formal notations to describe ontologies. For example, the Ontology Web Language (OWL) is a formal ontology for which Pereira et al. created OWL2DSL, an OWL to DSL converter [19]. However, OWL is mostly used in the context of the Semantic Web. Bunge-Wand-Weber (BWW) is a different kind of formal notation for an ontology and is one of the leading ontology frameworks used [17, 27]. The proposed ontology in this paper uses the following concepts of the BWW ontology to describe CPS: *Thing* (an elementary unit), *Properties* (attributes belonging to a thing), *State* (the values of all attributes of a thing), *Event* (a change of one or multiple properties, i.e. a change in state), *History* (all events of a thing), *Coupling* (whether the history of two things are independent or not), *System* (things which are connected to each other and have dependent histories), *Composition* (all things inside a system), *Environment* (all things outside a system that interact with things inside the system), *Structure* (the coupling among the components of the systems and the Environment), *Subsystem* (a system whose Composition and Structure are a subset of another system), *Input* (a thing in a system acted upon by an environmental thing), and *Output* (a thing in a system acting on an environmental thing) [8, 27].

For defining an ontology, we need a rigorous definition of a CPS. Embedded computers in CPS measure and interact with their environment via sensors and actuators. These embedded computers can communicate with cloud computers which can communicate with other CPS. However, we can describe a CPS more formally. Research conducted by Morozov et al. describes a possible modelling of CPS, where an unambiguous division is made between the physical and cyber parts of the system [18]. There is also a separation between the physical connections of the system and the cyber connection which are components connected via connected via software.

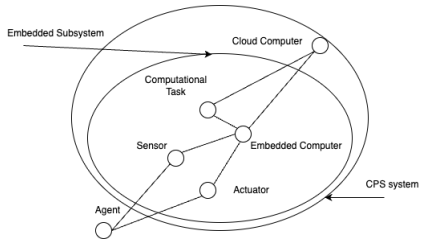

Fig. 3. A graphical view of a CPS where the nodes represent BWW things [27], lines show BWW Couplings, and the inner circle represents the BWW subsystem of the outer circle which is the BWW system.

More formally, a CPS is denoted as a tuple $CPS = \langle P, C, CPS, R^p, R^c \rangle$. P stands for the physical parts of the CPS, C stands for the cyber part, CPS stands for a another CPS which is part of the total CPS, R^p stands for a relationship between components that are physically connected and R^c stands for the cyber connection of components. This leads us to the ontology which is in Table 1.

In Figure 3 all Couplings can be seen between *Things* of a CPS and what *Things* constitute a CPS. It can also be seen that there is no distinction between physical and cyber connection as the BWW ontology does not make a distinction. The difference between the embedded system and the whole CPS highlights the interconnected nature of CPS.

With the ontology in Table 1, a DSL will be constructed via an iterative forward approach as mentioned in physics-of-notation methodology [17].

4 DESIGN

4.1 Language and features

For describing the behaviour of the system, we propose CPSL. The grammar of CSPL is shown in Figure 4. The grammar allows for describing the system in terms of its parts. The parts are then further specified by their types which are defined by the user. This allows for the modelling of niche types of a CPS. Types are preceded with an underscore symbol so it is easier to visually distinguish them between identifiers. The parts are split into physical and cyber parts as has been defined by Morozov et al[18]. Each part has a mode which denotes whether the sensor is on or off. If no mode is specified, the sensor is assumed to be on. The distinction between the two is that the cyber parts also need to be specified in terms of what computational tasks they compute. These computational tasks are written down in terms of requirements. The requirements themselves are declared in terms of what and how many minimum types of parts of the system are needed. The grammar also includes storing parts, computational tasks, requirements, and CPS into variable names via a declaration. The program is a list of declarations.

After specifying the behaviour, the user can enter which sensor will go down by referring to the sensor by its identifier and declaring

BWW concept	**CPS concept**	Description
Class	**Part**	Represents a physical or cyber part of the CPS.
Thing	**Sensor**	Represents a physical sensor on the CPS.
Thing	**Actuator**	Represents a physical actuator on the CPS.
Thing	**Embedded Computer**	Represents a physical local computer embedded on the CPS itself.
Thing	**Cloud Computer**	Remote computer which a CPS communicates with, i.e. sends and receives data.
Thing	**Computational Task**	A computational task requires Requirements
Thing	**Agent**	Represents an external source to the CPS upon which the CPS acts or the CPS acts on.
Property	**Mode**	Whether a part is turned off or on.
Property	**Requirements**	A list of values needed to perform a computational task.
Event	**Turn On**	When the property Mode changes to On.
Event	**Turn Off**	When the property Mode changes to Off.

Table 1. BWW Ontology of CPS

it as off. This feature is similar to querying which is done in the programming language Prolog [13]. Prolog is a declarative language where a programmer specifies what the situation is via rules and facts. After specifying the relevant situation, one can conduct a query which is either true or false in the world that is specified. The truthfulness of these queries is determined by the logic inference engine of the program [13]. Similarly, CPSL can use an inference engine to check whether the failure of one system affects any of the computational tasks performed by the CPS. If so, then CPSL will print out which computational tasks are affected. Because of the niche syntax and the under-the-hood inference engine that will accompany the DSL, the DSL has been chosen to be an external DSL.

Ontological representation	**Grammar representation**
Part	part
Sensor	physical_part
Actuator	physical_part
Embedded Computer	cyber_part
Cloud Computer	cyber_part
Computational task	comp_task
Mode (property of Part)	mode
Requirements (property of Computational Task)	req

Table 2. Mapping between the ontological and grammar constructs

As can be seen in Figure 4 comments, indentation & white space, substitution, and a numeric data type are included in the grammar of CPSL. These features are taken from the 'DSL Core' described in the paper of Zaytsev [29]. Zaytsev details a grounded theory for software language design by creating 96 so-called cards based on 24 books about software language design which cover at least 7 categories of software language design, namely: Parsing Techniques, Compiler Construction, Compiler Design, Language Implementation, Language Documentation, Programming Languages, and Software. In this paper, a cluster of cards is made for a DSL, referred to as the 'DSL Core'. These cards have been used as a software language design toolkit.

4.2 Ontological Analysis

In Table 2 the ontological representation is mapped to their grammar counterparts. At first glance, the grammar seems to be ontologically redundant since `physical_part` maps to both **Sensor** and **Actuator**, and `cyber_part` maps to both **Embedded Computer** and **Cloud Computer**. However, both non-terminals allow for `type` specification, allowing for distinction to be made by the end user. The grammar also shows that the *Property* **Mode** cannot exist outside of the *Thing* **Part** to which it is a property. However, a **Requirement** can exist as a standalone declaration outside of the **Computational Task**. This indicates that there is ontological excess. However, the requirement can only be part of a computational task and the decision to allow for stand-alone declarations was for better in-lining all requirements may lead to large variables being created. In addition, the *Events* **Turn On** and **Turn Off** are also incorporated into the language via `part_mode`.

4.3 Example

To illustrate the functionality of CPSL, the behaviour of the CPS which controls traffic flow in Enschede, a city in the East of the Netherlands, is constructed in CSPL. In Enschede, there are a select few traffic lights that will send a signal to a central computer that a biker is approaching when that biker has installed the 'Enschede Fietst' app and the central computer can communicate with the embedded computer in the traffic light pole [1]. The biker can also use the button on the traffic light pole (sensor) which will make the embedded computer in the traffic light make the traffic light (actuator) light up green. Let us also suppose that there are two

pressure sensors which allow the biker to not press the button and make use of these sensors that are present at the traffic light. In Figure 5 the BWW representation of this example can be seen.

In Figure 6, the described scenario is expressed in the proposed DSL. We can see that the computational task *displayLight* gets defined in terms of the types of parts needed for the task (pressure sensor, button, etc.). The type of the physical parts and cyber parts needs to be specified. The cyber parts also need to have a computational task assigned to them. Finally, the whole system is an agglomeration of all parts together. After having defined the system, the user should be able to disable parts of the system and see which computational tasks are affected. For example, if *centralComputer* is turned off, then the system can still perform the task *displayLight* as the pressure sensor, button, and embedded computer are still operational. If in addition, the embedded computer on the traffic light is turned off, then DSL will display that the task *displayLight* will be affected.

5 RELATED WORK

In this paper, a model for the DSL was constructed via an ontology. However, there are alternative approaches to modelling complex systems such as the work of Zaytsev about megamodelling [30]. In this paper, systems are represented as *Nodes*, *Graphs*, and *Automata* (NGA) via a megamodel. A megamodel is a type of model which is an amalgamation of other models. Therefore, each element in a megamodel is a model. The *Node* view represents the most abstract view and, as the name implies, is visualised as a node in a megamodel. A *Graph* view shows more details by representing arcs between *Nodes*, thus specifying the relationship between Nodes. The *Automaton* view models the behaviour of the system by specifying what happens when the system is run and can be modelled by automata used in software engineering. Together, these three types of representation allow for the modelling of the components, the relationship between components, and the behaviour of the system relevant to the model that is being constructed. These models can then be mapped to syntax in the language instead of the ontology.

The purpose of the DSL in this paper was to model the behaviour of the system. However, research conducted by van den Berg et al. has been made into the construction of CPS using DSL specifically focused on agricultural machines [26]. In their paper, van den Berg et al. propose a grammar in which all different parts of the CPS can be modelled together to form one whole system. Each part can model their requirements in terms of operations spaces. An operation space is a number of operation dimensions where an operation dimension is either a value or a range. For example, consider the requirement of a rotor needing to be a certain speed. We can specify this by saying the operation space 'speed' must have a value between '[10, 30]' rotations per second. Another example could be modes of light. The operation space 'light' can be modelled with valued '[on flicker off]'. The operation spaces get merged in a bottom-up fashion, i.e. system/parts at the top inherit operation spaces from their children. These top-level operations spaces can

```
program         : declaration* EOF;
cps             : part (AND part)*
                | IDENTIFIER (AND IDENTIFIER)*;
part            : physical_part | cyber_part
                | IDENTIFIER;
cyber_part      : physical_part COMPUTES comp_task
                  (AND comp_task)*;
physical_part   : type part_mode
                | type;
part_mode       : ON | OFF;
type            : '_'IDENTIFIER;
comp_task       : disjunctive_req;
disjunctive_req : conjuctive_req (OR conjuctive_req)+
                | conjuctive_req;
conjuctive_req  : req (AND req)+
                | req;
req             : MIN NUM type | IDENTIFIER;

declaration : IDENTIFIER ASSIGN expr;

expr    : cps_expr
        | part_expr
        | comp_task
        | req_expr;

cps_expr        : part (AND part)*;
part_expr       : physical_part | cyber_part;
req_expr        : MIN NUM type;

/** Fragments are type definitions to make the grammar
more readible, not used in the actual language */
fragment LETTER : [a-zA-Z];
fragment DIGIT  : [0-9];

ASSIGN      : '=';
COMPUTES    : '->';
MIN         : 'min';
ON          : 'on';
OFF         : 'off';
AND         : '&';
OR          : '|';
NUM         : DIGIT+;
IDENTIFIER  : LETTER (LETTER | DIGIT)*;

/** Comments with double slashes */
COMMENT     : '//' (~('\n'))* -> skip;

/** Ignore whitespace */
WS          : [ \n\t\r]+ -> skip;
```

Fig. 4. CPSL grammar in ANTLR

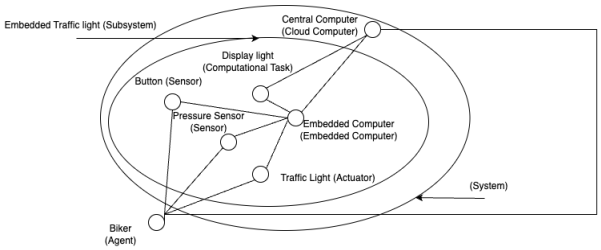

Fig. 5. A graphical view of the smart traffic light example where the CPS concepts are denoted in parentheses.

```
displayLight = min 1 _pressureSensor
             & min 1 _embeddedComputer
             | min 1 _button
             & min 1_embeddedComputer
             | min 1 _centralComputer
pressure1Traffic = _pressureSensor
pressure2Traffic = _pressureSensor
buttonTraffic    = _button
lightsTraffic    = _lightsActuator
computerTraffic  =
    _embeddedComputer -> displayLight
centralComputer  =
    _cloudComputer -> displayLight
TrafficSystem    = pressure1Traffic
                 & pressure2Traffic
                 & buttonTraffic
                 & lightsTraffic
                 & computerTraffic
                 & centralComputer
```

Fig. 6. Traffic light system expressed in CPSL.

then be used to generate a Pareto optimal system which satisfies all requirements, i.e. a system such that improving one part of the system would come at the cost of deteriorating another part.

Another DSL related to CPS is Triton, a high-level DSL that targets the Java Virtual Machine (JVM) [28]. This DSL was intended to program the behaviour of the DSL by specifying constraints within task blocks and what to do when these constraints are violated. It also allows for offloading, either synchronously or asynchronously (via an await-like structure), computationally expensive tasks to remote servers. The DSL was made specifically and allows for the integration of Remote Method Delegation (RMD), a grid computing platform, and MQ Telemetry Transport (MQTT), a machine network protocol. This DSL is closer to specifying the behaviour of DSL although it focuses primarily on the communication aspect of the DSL.

6 CONCLUSION

As part of work package 4 of project Zorro, we constructed a DSL, specifically for modelling the behaviour of a CPS. The DSL does so by showcasing what impact a component has on the system in terms of how many computational tasks are affected. This is done by specifying the computational tasks of the system in terms of what parts are needed for these tasks. The impact of a component failure is then measured in terms of how many computational tasks are affected. Combined with the output of the machine learning model used by Philips, the end user can make a more informed decision about replacing a component of a CPS. Therefore, CPSL aids in reducing the downtime of CPS.

The model of CPSL was built via the BWW ontology [27] which was constructed from a literature review on the definition of CPS. For future work, domain experts should be involved in the process of formulating the ontology as they can provide experience and knowledge which is not written down in academic literature.

The functionality of CPSL was demonstrated via an example of simple CPS. For future work, CPSL should be tested by modelling more complex CPS such as the MRI machines of the industry partner Philips. Preferably, this complex CPS should be written by domain experts in CPSL as this would not only test the DSL in terms of being able to model the system, but also provide an opportunity to get feedback from the domain experts on the developer experience.

In addition, the implementation and deployment phases of CPSL need to be executed. For the implementation phase, that includes research into whether a compiler or interpreter is more suited as well as creating the inference engine needed for the language. Furthermore, for the deployment phase, the language also needs more features as has been outlined in Zaytsev's paper about language design. For example, the language needs to have code completion for popular IDEs such as VS Code [29].

Overall, CPSL serves as a basis for modelling the behaviour of CPS and is as a good step at reducing downtime, the goal of project Zorro [2], via model-based systems engineering. However, it still needs more features before it can become a full standalone language.

REFERENCES

[1] 2023. Enschede Fietst. https://enschedefietsstad.nl/enschede-fietst-app/.
[2] 2024. Zero Downtime in cyber physical systems. https://zorro-project.nl/.
[3] Bert Bos, Tantek Çelik, Ian Hickson, and Hakon Wium Lie. 2011. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. W3C Recommendation, http://www.w3.org/TR/2011/REC-CSS2-20110607.
[4] Zaharah Allah Bukhsh and Irina Stipanovic. 2020. Predictive Maintenance for Infrastructure Asset Management. *IT Professional* 22, 5 (Sept. 2020), 40–45. https://doi.org/10.1109/MITP.2020.2975736
[5] Donald D. Chamberlin. 2012. Early History of SQL. *IEEE Annals of the History of Computing* 34, 4 (2012), 78–82. https://doi.org/10.1109/MAHC.2012.61
[6] Martin Fowler and Rebecca Parsons. 2007. *Domain-specific languages.* Addison-Wesley Professional.
[7] Emden Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. 2003. Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools. https://www.graphviz.org/documentation/EGKNW03.pdf.
[8] Boryana Goncharenko and Vadim Zaytsev. 2016. Language design and implementation for the domain of coding conventions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering.* ACM, Amsterdam Netherlands, 90–104. https://doi.org/10.1145/2997364.2997386
[9] Nicola Guarino. 1998. Formal Ontologies and Information Systems.
[10] David Harel and Bernhard Rumpe. 2003. Modeling Languages: Syntax, Semantics and All That Stuff Part I: The Basic Stuff. (Sept. 2003).
[11] Erkki Jantunen, Urko Zurutuza, Luis Lino Ferreira, and Pal Varga. 2016. Optimising maintenance: What are the expectations for Cyber Physical Systems. In *2016 3rd International Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC).* 53–58. https://doi.org/10.1109/EITEC.2016.7503697

[12] Siddhartha Kumar Khaitan and James D. McCalley. 2015. Design Techniques and Applications of Cyberphysical Systems: A Survey. *IEEE Systems Journal* 9, 2 (2015), 350–365. https://doi.org/10.1109/JSYST.2014.2322503

[13] Feliks Kluzniak and Stanisław Szpakowicz. 1985. Prolog for programmers. (1985).

[14] Edward Ashford Lee and Sanjit Arunkumar Seshia. 2017. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach* (second edition ed.). MIT Press.

[15] Lyudmila N. Lyadova, Alexander O. Sukhov, and Marsel R. Nureev. 2021. An Ontology-Based Approach to the Domain Specific Languages Design. In *2021 IEEE 15th International Conference on Application of Information and Communication Technologies (AICT)*. IEEE, Baku, Azerbaijan, 1–6. https://doi.org/10.1109/AICT52784.2021.9620493

[16] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *ACM Comput. Surv.* 37, 4 (dec 2005), 316–344. https://doi.org/10.1145/1118890.1118892

[17] Daniel Moody. 2009. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering* 35, 6 (Nov. 2009), 756–779. https://doi.org/10.1109/TSE.2009.67

[18] Dmitry Morozov, Mario Lezoche, and Hervé Panetto. 2018. Multi-paradigm modelling of Cyber-Physical Systems. *IFAC-PapersOnLine* 51, 11 (2018), 1385–1390. https://doi.org/10.1016/j.ifacol.2018.08.334 16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018.

[19] Maria João Varanda Pereira, João Fonseca, and Pedro Rangel Henriques. 2016. Ontological approach for DSL development. *Computer Languages, Systems & Structures* 45 (April 2016), 35–52. https://doi.org/10.1016/j.cl.2015.12.004

[20] Lukas Renggli. 2010. Dynamic Language Embedding With Homogeneous Tool Support. (2010). https://doi.org/10.7892/boris.104713

[21] Larry Rockoff. 2021. *The language of SQL*. Addison-Wesley Professional.

[22] Michael Rosemann, Iris Vessey, Ron Weber, and Boris Wyssusek. 2004. On the Applicability of the Bunge-Wand-Weber Ontology to Enterprise Systems Requirements. (Jan. 2004).

[23] Robert Tairas, Marjan Mernik, and Jeff Gray. 2009. Using Ontologies in the Domain Analysis of Domain-Specific Languages. In *Models in Software Engineering*, Michel R. V. Chaudron (Ed.). Vol. 5421. Springer Berlin Heidelberg, Berlin, Heidelberg, 332–342. https://doi.org/10.1007/978-3-642-01648-6_35 Series Title: Lecture Notes in Computer Science.

[24] Federico Tomassetti and Vadim Zaytsev. 2020. Reflections on the Lack of Adoption of Domain Specific Languages. In *STAF Workshop Proceedings (STAF) (CEUR Workshop Proceedings, Vol. 2707)*, Loli Burgueño and Lars Michael Kristensen (Eds.). CEUR-WS.org, 85–94. http://ceur-ws.org/Vol-2707/oopslepaper5.pdf

[25] Boris Ulitin and Eduard Babkin. 2017. Ontology and DSL Co-evolution Using Graph Transformations Methods. In *Perspectives in Business Informatics Research*, Björn Johansson, Charles Møller, Atanu Chaudhuri, and Frantisek Sudzina (Eds.). Vol. 295. Springer International Publishing, Cham, 233–247. https://doi.org/10.1007/978-3-319-64930-6_17 Series Title: Lecture Notes in Business Information Processing.

[26] Freek van den Berg, Vahid Garousi, Bedir Tekinerdogan, and Boudewijn R. Haverkort. 2018. Designing Cyber-Physical Systems with aDSL: a Domain-Specific Language and Tool Support. In *2018 13th Annual Conference on System of Systems Engineering (SoSE)*. 225–232. https://doi.org/10.1109/SYSOSE.2018.8428770

[27] Yair Wand and Ron Weber. 1990. An ontological model of an information system. *IEEE Transactions on Software Engineering* 16, 11 (Nov. 1990), 1282–1292. https://doi.org/10.1109/32.60316

[28] Bradley Wood and Akramul Azim. 2021. Triton: a Domain Specific Language for Cyber-Physical Systems. In *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*, Vol. 1. 810–816. https://doi.org/10.1109/ICIT46573.2021.9453575

[29] Vadim Zaytsev. 2017. Language Design with Intent. In *Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Don Batory, Jeff Gray, and Vinay Kulkarni (Eds.). IEEE, 45–52. https://doi.org/10.1109/MODELS.2017.16

[30] Vadim Zaytsev. 2017. Megamodelling with NGA Multimodels. In *Proceedings of the Second International Workshop on Comprehension of Complex Systems (CoCoS)*, Christoph Bockisch and Michael L. Van De Vanter (Eds.). ACM, 1–6. https://doi.org/10.1145/3141842.3141843