Developing a DSL Design Methodology for CPS Diagnostics

ALIAKSEI KOUZEL, University of Twente, The Netherlands

Diagnostics play a crucial role in ensuring the reliability and efficiency of Cyber-Physical Systems (CPSs). By promptly detecting system anomalies and their root causes, it becomes possible to ensure maximum uptime through preventive or corrective measures. However, developing an effective diagnostic system is challenging, requiring comprehensive knowledge about the expected system behavior. One way to achieve this is by specifying a model of the system using a Domain-Specific Language (DSL) that incorporates knowledge of the system's components and processes. In this research, we define a methodology for developing such a DSL. We begin by identifying a general methodology for designing a DSL applicable to the domain of CPS diagnostics. Then, we explore how the knowledge about CPS diagnostics can be formalized within a model. Finally, we investigate the methodology utilized by an existing DSL in this domain. This research is expected to contribute to the field of DSL development and the diagnostic formalization of complex systems.

Additional Key Words and Phrases: Cyber-Physical System, Domain-Specific Language, Diagnostics

1 INTRODUCTION

Cyber-Physical Systems (CPSs) represent complex structures, integrating software, networking, computation, and physical operations [37]. They are found in various fields such as transportation, healthcare, smart housing, and agriculture, accounting for a large part of the global economy [4]. For instance, the market size of CPSs was around $8.45 billion in 2023 and is anticipated to grow to $14.2 billion by 2031 [3]. Therefore, failures in such systems are critical and can potentially lead to significant financial losses due to reduced productivity and customer dissatisfaction. Additionally, system malfunctions could lead to serious injuries or even fatalities, as is the case with medical systems [37]. Hence, to ensure the reliability and efficiency of such systems, it is necessary to incorporate efficient diagnostics that allow the timely application of preventive or corrective measures.

However, various diagnostic techniques, such as Model-Based Diagnosis (MBD), require the formalization of the system's components and processes within a model, along with indicators of potential failures [32]. One possible solution to achieve this is by implementing a Domain-Specific Language (DSL), which is a computer programming language designed for a specific application domain [17]. In the scope of this research, the domain is CPS diagnostics. Thus, the DSL defines formal rules by which the system behaves under normal conditions. If the system functions differently from what is specified in the model, it could indicate a potential failure. Additionally, such models could provide insights into the specific causes of failures, allowing for a more orderly approach to their resolution.

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

Another alternative is to use a General-Purpose Language (GPL) such as Java, Python, or C++. However, DSLs are more appropriate for diagnostics as they offer reduced expressiveness, ensuring lower complexity and better maintainability [17, 27, 37]. DSLs also improve communication between programmers and domain experts, as the latter are not always familiar with GPL concepts [17].

Nevertheless, despite extensive research conducted on designing and evolving DSLs in general [17, 19, 21, 24, 27, 28, 38], modeling complex systems [6, 7, 10, 22, 26, 33, 37], and formalizing knowledge about system anomalies [8, 15, 29, 30, 32], there remains a gap in defining the methodology for DSL design specifically in the domain of CPS diagnostics. Such a methodology can provide an overview of things to take into consideration when developing a DSL for this domain. This knowledge can improve CPS diagnosability, thereby reducing downtime and the risks of critical failures. Hence, we express the purpose of this research as follows:

Define a methodology for DSL design in the domain of CPS diagnostics.

We will further explore this by considering the following research questions:

(1) How can a DSL be designed specifically for the domain of CPS diagnostics?
(2) How can the knowledge about CPS diagnostics be formalized within a model?
(3) What is the methodology used by CML, an existing DSL in the domain of CPS diagnostics?

This paper is organized as follows: Section 2 describes the methodology, Section 3 reviews related literature to establish the context and background for the research, Section 4 presents findings on the DSL design in the domain of CPS diagnostics, and Section 5 concludes the paper and suggests options for further research.

2 METHODOLOGY

In this section, we describe the steps used to achieve the research objectives, including an extensive literature review and an interview with an expert in language engineering.

2.1 Literature Review

To collect related literature within the research domain, we used Google Scholar. By utilizing combinations of search terms, such as "Model-Based Diagnosis", "Fault Diagnosis", "Domain-Specific Language", "Anomaly Detection", and "Cyber-Physical System", we identified a range of documents addressing these topics.

For the literature review, we combined the knowledge from existing work in the fields of domain-specific languages, model-based diagnosis, and model-driven engineering, and applied it to derive a methodology for DSL development specifically in the domain of CPS diagnostics.

We began by studying DSL design in general and identifying relevant information specific to the domain. In particular, we determined the purpose of the DSL and its usage for CPS diagnostics. We also defined the potentially supported features by the DSL based on existing applications. Next, we chose an appropriate category of the DSL applicable to the domain. Additionally, we derived design principles and computational models that are suitable for the domain of CPS diagnostics. Finally, we formulated development phases and their corresponding patterns based on the domain.

The next step in the literature review involved formalizing CPS diagnostics within the MBD framework, which is explained in subsection 4.2. Specifically, we explored MBD techniques to formalize system diagnostics within a model and examined how CPS modeling differs from traditional embedded systems modeling by considering the notions of system-of-systems and interoperability. We paid particular attention to the approach illustrated by Barbini et al. (2021) [8], which was utilized by the ADiA project [2] and discussed during the expert interview.

2.2 Expert Interview

An essential part of this study was an interview with a specialist at TNO-ESI [5], a research center specializing in systems design and engineering for the high-tech equipment sector. This interview provided valuable insights into the ADiA (Assisted Diagnostics in Action) project, which developed a DSL named CML (Component Modeling Language) used to model CPSs for computing diagnosability. We analyzed the methodology utilized in CML, involving aspects such as the framework, language concepts, functionality, and development phases.

3 RELATED WORK

In this section, we provide an overview of the existing work done in the fields of domain-specific languages, model-based diagnosis, and model-driven engineering. This knowledge is further used in the literature review from subsection 2.1 as part of the research methodology.

3.1 Domain-Specific Languages (DSLs)

The field of DSLs is a widely researched topic, covering aspects of real-world applications, development methodologies, and Model-to-Model (M2M) transformations. Fowler and Parsons (2010) [17] as well as Wąsowski and Berger (2023) [38] provide general overviews of DSLs, which discuss terminologies in DSL development, their categories, design guidelines, language aspects, and M2M transformations. Mernik et al. (2005) [28] analyze development phases and their recurring patterns utilized by DSLs. Mengerink et al. (2018) [27] explore the evolution of DSLs by examining (semi-)automated processes and identifying the most volatile components in DSL development. Hermans et al. (2009) [21] explore the success factors of DSLs, focusing on a DSL used for creating web services. Van den Berg et al. (2018) [37] developed and described a DSL for CPSs with the ability to compute the Pareto optimal system designs and explore all possible operation modes. In industrial contexts, Gray and Karsai (2004) [19] provide an overview of three DSLs, focusing on tool integration through model transformations, and quantitative analysis of DSLs with their generated code in C++. Lastly, Karsai et al. (2014) [24] detail 26 general principles for DSL design, providing practical guidelines for developers.

This research also builds on the methodologies and patterns of existing DSLs. For instance, Hawk [26] is a DSL embedded in Haskell, using a functional approach to define and verify micro-architectures. Verilog [7] is used for designing and verifying digital circuits at the register transfer level. An extension of Verilog, Verischemelog [22], embedded in Scheme [16], offers greater flexibility and maintainability for hardware design simulations. Facile [33] specifies micro-architecture simulators using fast-forwarding techniques. SHIFT [6] uses state-based descriptions to model complex systems such as automated highways and air traffic control. LINQ (Language Integrated Query) [14] is a DSL internal to C# that provides a user-friendly solution for querying data from sources like SQL or XML. UPPAAL [10] represents real-time systems as networks of timed automata. ATL [23] and VMT [35] focus on describing M2M transformations. mCRL2 [20] is used for modeling, validation, and verification of concurrent systems. Overall, these DSLs demonstrate a range of approaches and techniques, providing insights into design patterns for various domains.

3.2 Model-Based Diagnosis (MBD)

Understanding the domain of the DSL is crucial to its design. In the context of CPS diagnostics, this involves the concept of MBD. Within this area, Chandola et al. (2009) [15] provide different mathematical techniques for anomaly detection. Pietersma et al. (2005) [32] illustrate a model-based approach for deriving test sequences for fault diagnosis using the modeling DSL called Lydia. Barbini et al. (2021) [8] demonstrate an application of the MBD methodology to compute system diagnosability and identify hypothetical sensors needed to find root causes of system anomalies. Kurien and R-Moreno (2008) [25] as well as Pietersma and van Gemund (2007) [31] analyze the costs and benefits of the MBD approach. Munirathinam and Balakrishnan (2016) [30] also consider an alternative to the MBD approach in the form of Data-Driven Diagnostics (DDD), proposing machine learning techniques for predicting equipment faults in the semiconductor manufacturing process.

3.3 Model-Driven Engineering (MDE)

In the scope of this research, the DSL methodology is used within the context of MDE, and this paper considers general principles from this framework. For instance, Bézivin (2004) [13] explores metamodeling, model transformations, and automated code generation from models. Similarly, Brown (2004) [12] provides an introduction to MDE theory and its practical applications, offering examples of various models, extensions of modeling languages, and Rapid Application Development (RAD) solutions.

4 FINDINGS

4.1 RQ 1: DSL Design

There are many different aspects when it comes to the DSL design of CPS diagnostics. Due to the limited scope of this research, we considered only the following: DSL's purpose, features, category, principles, computational models, and development phases. These

aspects were chosen based on those considered in other DSL-related papers [17, 21, 28, 37, 38].

4.1.1 Purpose. Before designing a DSL, it is important to understand its application in the domain. For CPS diagnostics, the DSL can be applied within the MBD framework, where the system's health is inferred by its compositional model in comparison with observed inputs and outputs [32]. Specifically, a DSL is used to model the system and transform it into the primary language of the application, which is then used for further development of diagnostic algorithms. More information on that is provided in subsection 4.2.

4.1.2 Features. Based on the DSL's purpose and existing approaches to MBD, the language can potentially support the following features:

(1) Detection of faults and their root causes [9, 32, 37].
(2) Choosing CPS sensors [8, 9].
(3) Evaluating diagnosis accuracy [8, 9, 37].
(4) Evaluating diagnosis costs [32].
(5) Model extraction from existing data [9].
(6) Model visualization [8, 37].

However, the exact features would depend on the specific requirements of CPS stakeholders. Depending on the chosen feature set, the design can significantly vary. For instance, determining the root causes of CPS faults would require the mappings between anomalies and fault indicators to be embedded in the DSL design.

4.1.3 Category. According to Wąsowski and Berger (2023) [38], all DSLs can be divided into the following two categories: external DSLs and internal DSLs.

(1) **External DSLs**: Languages that operate independently from the primary language of the application they work with. For example, database queries in SQL, software building in Make, typesetting in LaTeX, and hardware design in VHDL [28].
(2) **Internal DSLs**: The use of a GPL in a specific way, utilizing a subset of its features to manage a particular aspect of the system. For example, querying data in LINQ within C# [14].

In the domain of CPS diagnostics, an internal DSL might not be a suitable choice due to the system's complexity and the possibility that domain experts may not have adequate skills in GPLs [17]. However, an external DSL that mirrors the structure and semantics of CPSs is a potential option. In this case, the DSL provides an intuitive understanding of the system, thereby facilitating communication between stakeholders.

DSLs can also be categorized according to their development workflow patterns, which is described in more detail in subsubsection 4.1.6.

4.1.4 Principles. To determine the principles behind the DSL design, it is necessary to understand its intentions. In the context of CPS diagnostics, the DSLs are expected to improve productivity during the development, maintenance, and utilization of diagnostic software by facilitating error detection, system modification, and program understanding [17, 28]. The latter is especially important as code comprehension can be time-consuming and may require more than half of the time allocated for software maintenance [39]. Another principal intention behind DSLs is to enhance interaction with domain specialists, as they are not always familiar with GPL concepts such as algorithms and data structures [17]. Taking this into account, we postulate the following design principles based on the existing work of Fowler and Parsons (2010) [17], Wąsowski and Berger (2023) [38], Hermans et al. (2009) [21], and Karsai et al. (2014) [24]:

(1) Use concise and simple syntax to facilitate communication with stakeholders.
(2) Use domain-specific terminology in the syntax and the semantic model to improve understandability for domain experts.
(3) Use common conventions familiar to everyday coding practices (e.g., adopting "//" for commenting if Java is widely used).
(4) Avoid ambiguity in definitions and reasoning.
(5) Avoid making the DSL resemble a natural language, as this introduces syntactic sugar that can obscure the semantics.
(6) Separate the DSL's semantic model and syntax, allowing their independent evolution.
(7) Implement automatic migration between DSL versions.
(8) Implement testing of the DSL's parser, scripts, and the semantic model.

4.1.5 Computational Models. When designing a DSL, it is crucial to choose the right computational model. This model determines the framework used to describe the computational processes and define the language semantics [17]. Most popular GPLs, such as Java, Python, and C++, utilize an imperative approach where the program consists of statements executed step by step [34]. They employ selection statements, iterative statements, support for object-oriented programming, and other constructs [34]. This approach is not always suitable, especially for CPS diagnostics, due to its high complexity for domain experts [17]. Therefore, a more declarative approach, such as the decision table, state machine, or production rule system, can be used instead [17].

Let us consider each one using a simple example of diagnosing a malfunction in a thermostat system. In this case, the diagnosis is based on conditions such as temperature readings (T), sensor status (S), and error codes (E). Also, the system adheres to the following rules consecutively:

(1) If E = 1, then the output is "system failure".
(2) If S != OK, then the output is "sensor failure".
(3) If T > 75, then the output is "high temperature".
(4) otherwise the system functions normally.

The result of translating the aforementioned example to a decision table is shown in Listing 1. As can be observed, this approach is efficient in combining the outputs of multiple interacting conditions. It is also well understood by both software engineers and domain experts [17]. This model can be used to define the correct system behavior in diagnostics as a set of conditions leading to either fault/non-fault states or the probabilities of failures. An example of this model's usage can be found in the work by Barbini et al. (2021)

```
inputs:
    TemperatureReading T,
    SensorStatus S,
    ErrorCode E

outputs:
    Diagnosis D

table:
    T > 75 ; S = OK  ; E = 0 ; D = high_temperature
    T = _  ; S != OK ; E = 0 ; D = sensor_failure
    T = _  ; S = _   ; E = 1 ; D = system_failure
```

Listing 1. Thermostat example as a decision table, formalizing diagnostics as a table of inputs/outputs.

```
states:
    normal,
    high_temperature,
    sensor_failure,
    system_error

parameters:
    TemperatureReading T,
    SensorStatus S,
    ErrorCode E

transitions:
    normal -> sensor_failure
        when E = 0 and S != OK

    normal -> high_temperature
        when E = 0 and S = OK and T > 75

    normal -> system_error
        when E = 1
```

Listing 2. Thermostat example as a state machine, formalizing diagnostics as states and transitions between them.

[8], which introduces a model-based approach for computing the system's diagnosability by generating Bayesian networks. However, the drawback of this model is that defining input conditions can be time-consuming, especially for complex systems [17].

Another alternative is translating the thermostat example to a state machine, which defines the system as a set of states and transitions between them, as shown in Listing 2. This approach can be used to describe CPS diagnostics with "normal" states and transitions that lead to "faulty" states. Its application can be found in DSLs such as SHIFT [6], which focuses on describing complex systems, and Facile [33], which is used for micro-architecture simulations.

The last alternative is the production rule system. It is similar to the decision table, but the difference is that it focuses on the behavior of individual rules rather than the whole table [17]. Its usage is illustrated in Listing 3. This model is more compact than

```
inputs:
    TemperatureReading T,
    SensorStatus S,
    ErrorCode E

outputs:
    Diagnosis D

rules:
    E = 1   -> D = system_failure
    S != OK -> D = sensor_failure
    T > 75  -> D = high_temperature
```

Listing 3. Thermostat example as a production rule system, formalizing diagnostics as rules in a specific order.

the decision table, but engineers should also consider how rules interact with each other.

Nevertheless, other computational models exist besides those mentioned in this research that can also be suitable for the domain of CPS diagnostics. For example, Petri Nets [18] for the description and analysis of systems characterized by concurrency, synchronization, and resource sharing.

4.1.6 Development Phases. When it comes to DSL design, we can also standardize the patterns for each of the development phases. Even though they may differ from application to application, it is still possible to define the fundamental steps present in a typical development workflow [28]. This paper considers the following phases: *decision*, *analysis*, and *design*, which were inspired by Mernik et al. (2005) [28].

In the *decision* phase, the purpose of the DSL is identified based on the expected functionality for a given domain. In the case of CPS diagnostics, the following patterns can be applied:

(1) **Task Automation**: Automating diagnostics by converting DSL code to an appropriate GPL.
(2) **Notation**: Formalizing knowledge about CPS diagnostics in a textual format.
(3) **AVORT**: Analysis, verification, optimization, parallelization, and transformation of CPS diagnostic models.

In the *analysis* phase, information about the domain is collected, including its scope, terminology, description of concepts, their variability, and interdependencies. The available patterns are:

(1) **Informal**: Informally examining the domain.
(2) **Formal**: Applying structured methodology.
(3) **Extracted**: Automatically extracting domain knowledge from an existing GPL or technical documentation.

In the *design* phase, the formal nature of the DSL and its relationship with existing languages are characterized. For this purpose, the following patterns can be derived:

```
system AndGate (
    bool x1, x2, // inputs
    bool h,      // health
    bool y       // output
) {
    // explicit fault mode: stuck-at-zero
    y = (h ? (x1 and x2) : false);
}
system OrGate (
    bool x1, x2, // inputs
    bool h,      // health
    bool y       // output
) {
    // explicit fault mode: stuck-at-zero
    y = (h ? (x1 or x2) : false);
}
```

Listing 4. AND and OR gate models in Lydia [32].

Association with existing languages:

(1) **Exploitation**: The DSL partially/fully uses an existing GPL (same as an internal DSL).
(2) **Invention**: The DSL has no relationship with existing GPLs (same as an external DSL).

Formal nature of the DSL:

(1) **Informal**: Specifying a DSL informally by, for example, using a natural language.
(2) **Formal**: Specifying a DSL formally using techniques for semantic definition like grammars or state machines.

4.2 RQ 2: Modeling CPS Diagnostics

This research considers MBD as a framework for incorporating knowledge about CPS diagnostics within a model. This approach is used to automatically determine faults in the system based on the differences between the actual and expected behavior [32]. If MBD is applied to the discrete domain, then the process can be formalized as follows [32]:

$$outputs = Model(inputs, health)$$
$$health = Model^{-1}(inputs, outputs)$$

where "health" is a collection of health states of system components, "inputs" is the inbound information to those components, and "outputs" is the outbound information. As can be observed, by "inverting" the specified model, it is possible to determine the system's health (diagnosis) based on the observed inputs and outputs. In this context, "inverting" is an abstract way to describe the use of the model in diagnostic algorithms.

4.2.1 Applications. One possibility for utilizing the MBD framework was introduced by Pietersma et al. (2005) [32]. They proposed automatically deriving optimal test sequences at the lowest computational cost, where the cost is defined as the average number

```
system Polycell (
    bool x1, x2, x3, x4, x5, // inputs
    bool h1, h2, h3, h4, h5, // healths
    bool y1, y2              // outputs
) {
    // declare intermediate outputs
    bool z1, z2, z3;

    // declare components
    system AndGate m1, m2, m3;
    system OrGate a1, a2;

    // connect components
    m1 (x1, x2, h1, z1);
    m2 (x3, x4, h2, z2);
    m3 (x2, x5, h3, z3);
    a1 (z1, z2, h4, y1);
    a2 (z2, z3, h5, y2);

    // define health probabilities
    probability (h1 = true) = 0.99;
    probability (h2 = true) = 0.99;
    probability (h3 = true) = 0.99;
    probability (h4 = true) = 0.99;
    probability (h5 = true) = 0.99;
}
```

Listing 5. Polycell model in Lydia [32].

Table 1. Partial polycell diagnosis table, where x indicates inputs, y indicates outputs, h indicates health states of components (diagnosis), and P(h) indicates the probability of diagnosis [32].

x	y	h	P(h)
(1, 1, 1, 1, 1)	(1, 0)	(1, 1, 1, 1, 0)	0.0096
	(0, 1)	(1, 1, 1, 0, 1)	0.0096
	(1, 1)	(1, 1, 1, 1, 1)	0.9510
		(0, 1, 1, 1, 1)	0.0096
		(1, 0, 1, 1, 1)	0.0096
		(1, 1, 0, 1, 1)	0.0096
(1, 1, 1, 0, 1)	(1, 0)	(1, 1, 1, 1, 0)	0.0096
		(1, 1, 0, 1, 1)	0.0096
	(0, 1)	(1, 1, 1, 0, 1)	0.0096
		(0, 1, 1, 1, 1)	0.0096
	(1, 1)	(1, 1, 1, 1, 1)	0.9510
		(1, 0, 1, 1, 1)	0.0096

of tests necessary to determine a diagnosis. Probabilities were assigned to the expected health states, and from all the variations of inputs/outputs, the information content or entropy was obtained. This calculation allowed them to determine the information gain for inputs, which was used to decide the sequences of tests forming the test decision tree. An example of formalizing the system for this approach is shown in Listing 4 [32], which specifies the behavior of AND and OR gates using the modeling language called Lydia. These gates are then combined to form a polycell system as illustrated in

Listing 5 [32]. Finally, the system is used to derive diagnostic data as in Table 1, which determines the probabilities of components' health states based on inputs/outputs.

Another application within the context of MBD was shown by Barbini et al. (2021) [8]. Even though their approach is not directly used for fault diagnosis, but rather for computing diagnosability at design time, we still considered it from the perspective of modeling system diagnostics. They proposed viewing the models as mappings from inputs/outputs to failure modes (root causes of failures). These mappings are then used to generate Bayesian Networks (BNs) for calculating the diagnosability of CPSs and determining the required sensors. This is discussed in more detail in subsection 4.3, which showcases the application of this approach based on CML, a DSL developed in the ADiA project [2].

4.2.2 System-of-Systems (SoS). CPSs are systems that integrate software, networking, computation, and physical operations [37]. This requires an approach known as SoS, where physical processes are observed and managed by multiple embedded systems and networks [37]. As a result, CPS models expand conventional embedded system models with additional support for network connectivity, temporal alignment (concurrency), and seamless integration between components (interoperability) [37]. Furthermore, if CPSs consist of systems requiring different model specifications, then their formalization can utilize the megamodeling approach [36]. The megamodels are models that combine other models and transformations between them and can be used as abstract languages for model operations [36]. Overall, when language engineers describe CPSs using a model-based approach, they need to consider aspects such as networking, concurrency, interoperability, and megamodeling.

4.2.3 Interoperability. According to IEEE, interoperability is defined as the capability of multiple systems or components to share data and make use of the shared data [1]. It is essential in modeling CPSs, as these systems can be composed of networks from various manufacturers and vendors [37]. One approach to address interoperability was demonstrated in the DSL called aDSL, where operation spaces and their constraints are clearly defined for each subsystem, allowing automatic validation for interoperability during the design phase [37]. Embedding interoperability validation is beneficial for CPS diagnostics, as conflicts in operational spaces might cause system failures. For example, if an autonomous car's collision avoidance system requires a minimum distance of 2 meters from obstacles to function correctly, while its parking assist system needs to operate within 1 meter of surrounding objects, this creates an interoperability conflict.

4.3 RQ 3: ADiA project

One of the notable instances of fault formalization in complex systems was illustrated in the project called ADiA [2]. It is an initiative by TNO-ESI [5], a Dutch research institute specializing in the development of advanced methodologies and tools for the design and maintenance of complex high-tech systems. The project focused on developing advanced diagnostic tools for complex systems to enhance the efficiency and accuracy of troubleshooting system failures. It uses a technique suggested by Barbini et al. (2021) [8], which involves generating BNs used to identify the root causes of faults based on the system's inputs/outputs. An example of generating a BN can be observed in the valve model shown in Figure 1 [8], where the valve's behaviors were initially specified in a table. ADiA, on the other hand, formalized the system's components and processes using a DSL called CML (Component Modelling Language).

In the scope of this research, an interview was conducted with one of the CML developers to analyze its design methodology. The interview determined the development framework, language concepts, functionality, and development phases.

4.3.1 Framework. CML was developed using the Xtext framework, an open-source project for developing DSLs and other programming languages [11]. This approach is often referred to as using a language workbench [17, 38] for developing an *external DSL*, a category from subsubsection 4.1.3 of the DSL design. The motivation for choosing this framework was the CML developer's familiarity with the Eclipse IDE, an environment supported by Xtext.

4.3.2 Language Concepts. The DSL utilizes the following elements (or concepts) to construct models:

(1) **Components**: Basic building blocks of the system.
(2) **Composites**: Aggregates of components.
(3) **Inputs/Outputs**: Defined explicitly for each component.
(4) **Failure Modes**: Categories of failure root causes.
(5) **Failure Maps**: Mappings from inputs/outputs to failure modes.

The overall structure resembles an object-oriented approach, where components and composites can be defined as "objects", and inputs/outputs as "attributes". Failure maps, in turn, are an instance of a "decision table" computational model from subsubsection 4.1.5, where combinations of attributes and failure models are specified in a matrix-like format. This choice of elements and semantics was largely made by mirroring the concepts by Barbini et al. (2021) [8], with an addition of "composites" for combining multiple components.

4.3.3 Functionality. Regarding the DSL's functionality, the following features were mentioned during the interview:

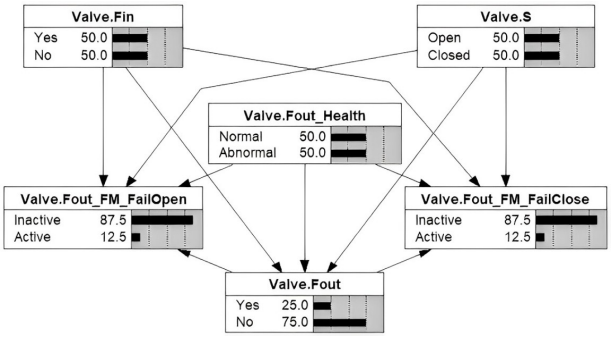
(1) **Validation**: Automatically validates the code syntax.
(2) **Autocompletion**: Autocompletes code for faster development. For example, entries in failure maps.
(3) **Generation**: Generates the target language, which involves BNs for diagnostic reasoning.

Where "validation" and "autocompletion" are supported to facilitate the development and maintenance of the DSL, and "generation" is included as part of the mandatory features.

4.3.4 Development Phases. The interview also captured the development phases of CML. It was mentioned that the methodology utilized an incremental development approach, following these steps in cycles:

Inputs		Output	Health	Failure Mode
F_{in}	S	F_{out}		
Yes	Open	Yes	Normal	
Yes	Closed	No	Normal	
No	Open	No	Normal	
No	Closed	No	Normal	
Yes	Open	No	Abnormal	FailOpen
Yes	Closed	Yes	Abnormal	FailClose
No	Open	Yes	Impossible	
No	Closed	Yes	Impossible	

(a) Behaviour mappings.



(b) Generated BN from behavior mappings.

Fig. 1. BN generation for diagnosability of a valve model [8].

(1) **Define Backbone**: Core concepts and their dependencies.
(2) **Write Grammar**: Syntax rules for the language.
(3) **Build Generator(s)**: Generate the target language.

As can be observed, these steps are different from those specified in subsubsection 4.1.6 of the DSL design. This is because they refer more to the implementation part of the DSL, following the decision, analysis, and design phases. The motivation for the incremental approach was to lower the costs in case of changes in the requirements or design.

5 CONCLUSION

Throughout this research, we identified the key factors to consider when designing a DSL for CPS diagnostics and investigated the methodology of an existing DSL in this domain. To achieve our research objectives, we conducted an extensive literature review and interviewed a language engineer from TNO-ESI to understand the methodology of CML, a DSL in the domain of CPS diagnostics.

To study how a DSL can be designed specifically for the domain of CPS diagnostics, we considered the following aspects: its purpose, features, category, principles, computational models, and development phases. We determined that an external DSL is more appropriate than an internal DSL for this domain. Additionally, we found that the DSL should facilitate error detection, system modification, program understanding, and communication with domain experts, deriving eight corresponding principles. Regarding computational models, we examined the decision table, state machine, and production rule system, concluding that they all apply to system diagnostics based on a thermostat example. Lastly, we found that the development phases can be divided into decision, analysis, and design phases, each with its patterns for CPS diagnostics.

To understand how knowledge about CPS diagnostics can be formalized within a model, we explored the MBD framework and its possible applications, as well as how CPS modeling differs from embedded systems modeling. We demonstrated that a system can be modeled using a DSL to derive diagnosis probabilities given the system's inputs and outputs. Additionally, we showed that a system can be modeled in a table-like format to generate Bayesian networks for computing diagnosability. Finally, we indicated that CPS systems, in comparison with embedded systems, should consider networking, concurrency, interoperability, and megamodeling in order to formalize them in a model.

After studying the methodology of CML, we discovered that it follows an object-oriented approach with a "decision table" computational model. It comprises composites and the components they consist of, as well as the inputs and outputs that connect them. The primary features of the DSL are validation, autocompletion, and generation. Lastly, its development phases are iterative, involving defining the backbone model, writing grammar, and building the generator(s).

For future work, the research can be extended with additional computational models and their examples, besides a thermostat system, to better represent the differences between models. Additionally, more DSL principles can be introduced and described in more detail, such as offering examples when principles are violated. Last but not least, other methodologies from existing DSLs in the domain of CPS diagnostics can be studied and used to improve the methodology detailed in this paper.

6 ACKNOWLEDGMENTS

I would like to thank my supervisors, Vadim Zaytsev and Marcus Gerhold, for their guidance, valuable insights, and timely feedback throughout this research.

REFERENCES

[1] 1991. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. *IEEE Std 610* (1991), 1–217. https://doi.org/10.1109/IEEESTD.1991.106963
[2] 2024. *Building a digital diagnostic assistant on shared experiences - ESI.* https://esi.nl/news/articles/2024-digital-diagnostic-assistant
[3] 2024. *Cyber Physical System Market Size, Trends and Projections.* https://www.marketresearchintellect.com/product/global-cyber-physical-system-market-size-and-forecast/
[4] 2024. *ISO - Cyber-physical systems.* https://www.iso.org/foresight/cyber-physical-systems.html
[5] 2024. *TNO - ESI.* https://esi.nl/
[6] Marco Antoniotti and Aleks Gollu. 1997. SHIFT and SMART-AHS: a language for hybrid system engineering modeling and simulation. 171–182. https://www.researchgate.net/publication/221628247_SHIFT_and_SMART-AHS_a_language_for_hybrid_system_engineering_modeling_and_simulation
[7] Peter Ashenden. 2007. *Digital Design (Verilog): An Embedded Systems Approach Using Verilog.* Morgan Kaufmann.

[8] Leonardo Barbini, Carmen Bratosin, and Thomas Nägele. 2021. Embedding Diagnosability of Complex Industrial Systems Into the Design Process Using a Model-Based Methodology. *PHM Society European Conference* 6, 1 (2021). https://doi.org/10.36001/phme.2021.v6i1.2806

[9] Anup Barve. 2005. *Model-Based Diagnosis - An ASML Case Study.* Master's thesis. Delft University of Technology.

[10] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. 1996. UPPAAL — a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*. Springer Berlin Heidelberg, 232–243. https://doi.org/10.1007/BFb0020949

[11] Lorenzo Bettini. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing.

[12] Alan Brown. 2004. Model driven architecture: Principles and practice. *Software and System Modeling* 3 (2004), 314–327. https://doi.org/10.1007/s10270-004-0061-2

[13] Jean Bézivin. 2004. In Search of a Basic Principle for Model Driven Engineering. *Novatica/Upgrade* 5 (2004).

[14] Charlie Calvert and Dinesh Kulkarni. 2009. *Essential LINQ.* Addison-Wesley Professional.

[15] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *Comput. Surveys* 41, 3 (2009), 1–58. https://doi.org/10.1145/1541880.1541882

[16] Kent Dybvig. 2009. *The Scheme Programming Language, fourth edition.* MIT Press.

[17] Martin Fowler and Rebecca Parsons. 2010. *Domain Specific Languages.* Addison-Wesley Professional.

[18] Claude Girault and Rüdiger Valk. 2003. *Petri Nets for Systems Engineering.* Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-05324-9

[19] Jeff Gray and Gábor Karsai. 2004. An Examination of DSLs for Concisely Representing Model Traversals and Transformations. (2004). https://doi.org/10.1109/HICSS.2003.1174892

[20] Jan F. Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Weerdenburg. 2006. The Formal Specification Language mCRL2. (2006). https://www.researchgate.net/publication/30815544_The_Formal_Specification_Language_mCRL2

[21] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2009. Domain-Specific Languages in Practice: A User Study on the Success Factors. 423–437. https://doi.org/10.1007/978-3-642-04425-0_33

[22] James Jennings and Eric Beuscher. 1999. Verischemelog: Verilog Embedded in Scheme. In *2nd Conference on Domain-Specific Languages (DSL 99)*. USENIX Association. https://www.usenix.org/conference/dsl-99/verischemelog-verilog-embedded-scheme

[23] Frédéric Jouault and Ivan Kurtev. 2005. Transforming models with ATL, Vol. 3844. https://doi.org/10.1007/11663430_14

[24] Gábor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2014. Design Guidelines for Domain Specific Languages. https://doi.org/10.48550/ARXIV.1409.2378

[25] James Kurien and María R-Moreno. 2008. Costs and Benefits of Model-based Diagnosis. *IEEE Aerospace Conference Proceedings*, 1–14. https://doi.org/10.1109/AERO.2008.4526647

[26] John Launchbury, Jeffrey R. Lewis, and Byron Cook. 1999. On embedding a microarchitectural design language within Haskell. 34, 9 (1999), 60–69. https://doi.org/10.1145/317765.317784

[27] Josh G. M. Mengerink, Bram van der Sanden, Bram C. M. Cappers, Alexander Serebrenik, Ramon R. H. Schiffelers, and Mark G. J. van den Brand. 2018. Exploring DSL Evolutionary Patterns in Practice - A Study of DSL Evolution in a Large-scale Industrial DSL Repository. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*. SCITEPRESS - Science and Technology Publications. https://doi.org/10.5220/0006605804460453

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

[29] Alexandros Mouzakitis. 2013. Classification of Fault Diagnosis Methods for Control Systems. *Measurement and Control* 46, 10 (2013), 303–308. https://doi.org/10.1177/0020294013510471

[30] Sathyan Munirathinam and Ramadoss Balakrishnan. 2016. Predictive Models for Equipment Fault Detection in the Semiconductor Manufacturing Process. *International Journal of Engineering and Technology* 8 (2016), 273–285. https://doi.org/10.7763/IJET.2016.V8.898

[31] Jurryt Pietersma and Arjan J.C. van Gemund. 2007. Benefits and Costs of Model-Based Fault Diagnosis for Semiconductor Manufacturing Equipment. *INCOSE International Symposium* 17, 1 (2007), 324–335. https://doi.org/10.1002/j.2334-5837.2007.tb02878.x

[32] Jurryt Pietersma, Arjan J.C. van Gemund, and Andre Bos. 2005. A model-based approach to sequential fault diagnosis. In *IEEE Autotestcon*. IEEE. https://doi.org/10.1109/autest.2005.1609208

[33] Eric Schnarr, Mark Hill, and James Larus. 2001. Facile: A Language and Compiler for High-Performance Processor Simulators. https://doi.org/10.1145/378795.378864

[34] Robert Sebesta. 2016. *Concepts of Programming Languages.* Pearson. https://books.google.nl/books?id=Wv6toQEACAAJ

[35] Shane Sendall, Gilles Perrouin, Nicolas Guelfi, and Olivier Biberstein. 2010. Supporting Model-to-Model Transformations: The VMT Approach. (2010).

[36] Jocelyn Simmonds, Daniel Perovich, María Cecilia Bastarrica, and Luis Silvestre. 2015. A megamodel for Software Process Line modeling and evolution. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 406–415. https://doi.org/10.1109/MODELS.2015.7338272

[37] 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)*. IEEE. https://doi.org/10.1109/sysose.2018.8428770

[38] Andrzej Wąsowski and Thorsten Berger. 2023. *Domain-Specific Languages.* Springer. https://doi.org/10.1007/978-3-031-23669-3

[39] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* 44, 10 (2018), 951–976. https://doi.org/10.1109/TSE.2017.2734091