35th European Modeling & Simulation Symposium
20th International Multidisciplinary Modeling & Simulation Multiconference

2724-0029 © 2023 The Authors.
doi: 10.46354/i3m.2023.emss.015

A domain specific language for distributed modeling

Jan Zenisek[1,2*], Florian Bachinger[1], Erik Pitzer[1], Stefan Wagner[1], Dominik Falkner[3], Alfredo Lopez[4] and Michael Affenzeller[1,2]

[1]Heuristic and Evolutionary Algorithms Laboratory, University of Applied Sciences Upper Austria, Softwarepark 11, Hagenberg, 4232, Austria
[2]Institute for Symbolic Artificial Intelligence, Johannes Kepler University, Altenberger Straße 69, Linz, 4040, Austria
[3]RISC Software GmbH, Softwarepark 32a, Hagenberg, 4232, Austria
[4]Software Competence Center Hagenberg GmbH (SCCH), Softwarepark 32a, Hagenberg, 4232, Austria

[*]Corresponding author. Email address: jan.zenisek@fh-hagenberg.at

Abstract

As the digital transformation of industry continues, more and more data is being collected to gain insights into and further improve existing processes, known as *prescriptive analytics*. Among the enabling technologies for prescriptive analytics is simulation-based optimization. To accelerate the execution of simulations, the approach can be coupled with machine learning methods to create so-called *surrogate models*. However, this can lead to a loss of modeling accuracy if processes can only be inadequately mapped to such models. In this work, we present a new domain specific language, to model complex systems as a directed graph of smaller, communicating system components. With this language, surrogates may be developed more flexible, i. e., only for those parts, where it is meaningful. Further on, the execution of modeled components can be distributed to gain speedup. We provide an overview of the created language syntax, development process and support. We also show the applicability of the language in a case study: in terms of parsing speed, the language performs at the same level as comparable markup languages, while it outperforms them in terms of brevity, showing that it is more expressive. Finally, we outline additional features and the future application context of the language.

Keywords: Domain Specific Language, Modeling and Simulation Software, Surrogate Modeling, Prescriptive Analytics

1. Introduction

For decades, computer simulation is used in a multitude of domains to model complex, dynamic systems and facilitate experimenting with them. Inside the digital environment, simulation provides an inexpensive playground to test what-if scenarios without the risks and limitations of real-world setups. The research branch simulation-based optimization is dedicated to combine simulation with suitable algorithms for purposeful experimenting (Gosavi et al., 2015). Therefore, problem-specific optimization criteria (e. g., *minimum cost*, *maximum resilience*, etc.) must be defined, which subsequently guide the algorithms' exploration through the hypothesis space for a most optimal solution. In this context, simulation is utilized as a parameterizable *fitness function* to evaluate solution candidates, while one candidate represents a set

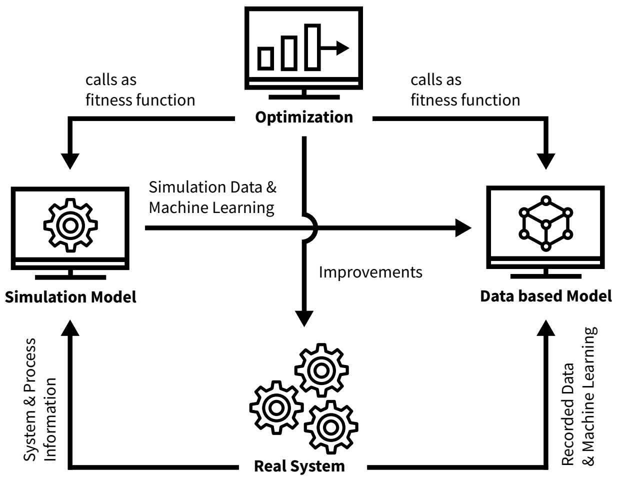

Optimization
calls as fitness function
calls as fitness function
Simulation Data & Machine Learning
Improvements
Simulation Model
Data based Model
System & Process Information
Recorded Data & Machine Learning
Real System

Figure 1. A conventional model-based optimization approach: A real-world system is digitally modeled either as simulation model by simulation experts using domain knowledge (left branch) or as data based model by machine learning algorithms using recorded data (right branch). A data-based *surrogate* model may also be created, using collected data from an existing simulation. Subsequently, an optimization system can call one of the models as *fitness function* to evaluate solution candidates to a certain optimization goal (e. g., a cost-optimal production or logistics schedule), which may be transferred back in to the physical world ultimately.

of parameters to setup the system (e. g., a production or logistics schedule). Ultimately, such a digitally generated optimization result may be transferred back to the physical world for real improvements.

This conventional simulation-based optimization approach is depicted in Figure 1 including the extension by so-called *surrogate models* (Koziel and Leifsson, 2013): Computer simulation is a practical way to transfer domain knowledge to a user-friendly, i. e., comprehensible, digital model, often including graphical interfaces. However, in the context of optimization, not usability, but execution-speed is a major performance indicator of simulation models, since the exploration for an optimal solution may take numerous iterations and thus, fitness evaluations (Pitzer and Kronberger, 2015). By using machine learning methods on data, recorded from previous simulation runs, surrogate models can be trained. If these models achieve good estimation quality, they can be used as a substitute for the original simulation in the optimization process, since they are much faster to run. According to Werth et al. (2019), in addition to a suitable and optimally configured machine learning method, the "right" choice of abstraction level is an important factor on the efficiency of surrogate models for optimization. With the "right" abstraction level, a good tradeoff between execution speed and modeling accuracy can be found and furthermore, *overfitting* can be avoided, which enables well generalizing models.

With regards to these conclusions, in this paper we want to pick up and further develop the idea of hierarchical decomposition and dynamic aggregation of complex systems by Zenisek et al. (2022). Figure 2 illustrates this concept and transfers it into the previously described optimization scenario. Its main advantage is the additional flexibility in terms of granularity of simulation and surrogate modeling. The concept enables that surrogate modeling can be used where it is meaningful and to stick to the original simulation where not. Surrogates might be used for system parts where a certain accuracy can be reached with machine learning, and/or where the original simulation runtime is relatively high. In case there are surrogates used for all system parts, the overall execution speed will at least add up to the same value as a monolithic surrogate would produce, so that no advantage remains. However, having individual surrogates for all system parts enables to distribute their execution over multiple computation nodes for parallelization and thus, creates new speedup potential. One additional benefit of the decomposed modeling approach is that different surrogate model types can be used for each component and level as long as it is possible to aggregate them porperly. Finally, regardless of using surrogates or not, by following this approach, models may be created easier in an iterative-incremental fashion. Apart from these advantages, more components also mean more effort for synchronization. A monolithic optimization system may be configured on one machine with a few clicks or some lines of code. A decomposed and distributable system, however, needs a full communication technology stack and accordingly, increased configuration effort.

In this work, we present a new domain specific language, called *Structued interaction description language (Sidl)*, which enables the definition of a directed communication graph to support the proposed modeling approach. In section 2 an outline of comparable languages is given, which influenced our design considerations. The developed language syntax for its core application as well as the tools for language development and support are presented in section 3. To demonstrate the applicability of our current developments, we show promising results from runtime tests and discuss the software design of its planned utilization in section 4. The final section 5 briefly summarizes this work and provides our ideas for further extending the developed language.

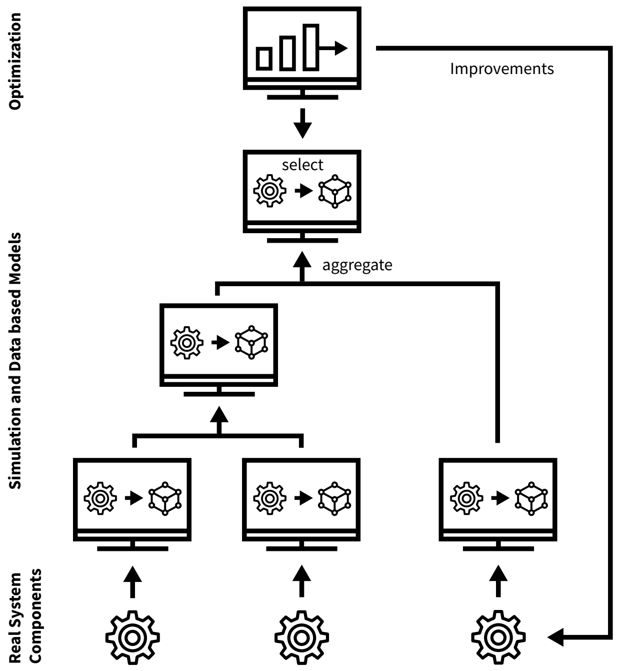


Figure 2. A modified model-based optimization approach: A real-world system is hierarchically decomposed into smaller, interdependent parts. Each component of the system components is individually, digitally modeled, either as simulation (left side in screen symbol) or data based model (right side). The output of components on lower levels represents the input of those on subsequent levels such that the top-level component represents the aggregated system. This component, may be called by an optimization system as a fitness function.

2. Related Work

Domain-specific languages (DSLs) are programming languages that are designed to address the specific needs of a particular domain or problem space. Unlike general-purpose programming languages (GPLs), which are flexible and versatile enough to address a wide range of programming tasks, DSLs are targeted to a specific application context. DSLs are typically easier to learn and use than GPLs, as they have a simplified syntax and tend to be more expressive, which allows developers to realize ideas more clearly and succinctly. DSLs enable to work at a higher level of abstraction and focus on the problem domain rather than the technical details of the implementation. Miller et al. (2010) provide a detailed description and taxonomy to DSLs and GPLs. In the following, we list several existing DSLs that were developed specifically for modeling and simulation. We evaluate their applicability, features, and shortcomings, and explain why we developed a new DSL. After that, we outline several existing languages from different domains, which influenced this development.

2.1. Modeling and Simulation Languages

Modelica[1] is a modeling language specifically designed for complex cyber-physical systems, including mechanical, electrical, thermal, and control systems. It provides a powerful framework for modeling and simulation of multi-domain systems.

Simulink[2] is a graphical programming environment developed by MathWorks. It is widely used for modeling, simulating, and analyzing dynamic systems, particularly in the fields of control engineering, signal processing, and communications.

GPSS (General Purpose Simulation System, Greenberg (1972)) is a discrete-event simulation language used for modeling and simulating systems with discrete events. It provides constructs for modeling entities, resources, queues, and transactions, allowing the simulation of a wide range of systems. Nowadays, it only plays a subordinate role. Nevertheless, there is still corresponding software for today's architectures, e. g., SLX.

SLX (Simulation Language with eXecution, Henriksen (1996)) is a DSL specifically designed for discrete event simulation. It combines a high-level modeling language with a runtime system that supports the execution of simulation models. SLX is suitable for various simulation domains, including manufacturing, logistics, and transportation.

AMPL (A Mathematical Programming Language, Fourer et al. (1990)) is a high-level modeling language for mathematical optimization problems. While it is not specific to simulation, AMPL is commonly used in simulation models that involve optimization and decision-making components.

The listed DSLs all represent quite powerful, but also complex languages (i. a., concerning syntax), with large ecosystem for the eventual application domain (e. g., science, engineering, production, logistics, etc.). Our goal however, is to facilitate the work of many modeling stakeholders by one, more simplistic language, which is exactly tailored to the pursued modeling approach, as in Figure 2. Several DSLs for modeling and simulation incorporate connections to heuristic optimizers and exact solvers, e. g., AMPL. This is also necessary for the pursued optimization approach in this work. However, at this time, there is no DSL which supports the component based, surrogate modeling idea, we envisaged. Since it

1 https://modelica.org/documents/MLS.pdf, version 3.6
2 https://www.mathworks.com/products/simulink

seemed not suitable to extend one of the listed DSLs for this purpose, we decided to create a new one.

2.2. Influencing Languages

Markup languages, which enable the definition of key-value pairs, such as Json[3], Yaml[4], Unix INI-files or many others are well suited for most software configuration purposes. They are easy to learn and hence, used for a broad range of applications, e. g., also as payload format for messaging. In order to enable domain specific validation of Json- or Yaml-based configuration files, so-called schema files can be setup and checked against. Due to their general-purpose approach, however, these languages lack of problem-specific writing support and texts tend to get verbose.

Lua[5] is a general purpose, extensible programming language, which is regularly used as an embedded scripting language, e. g., in computer games to specify additional, custom functionality (Ierusalimschy et al., 1996). Due to its brevity and clarity, the Extended Backus-Naur Form (EBNF) of Lua has been the major template regarding the style of our DSL's EBNF and many language structure definitions (e. g., statements, variables, lists, etc.) have been imitated. Moreover, for future development of the Sidl DSL, we consider the possibility to integrate Lua as embedded language for programming node functors – minimalistic data transformations. Besides Lua, also C# and Python are among the GPLs which inspired the syntax design of our DSL.

DOT[6] is a description language for structure and visual appearance of graphs and part of the open source software package *GraphViz* (Gansner and North, 2000). Its syntax is inspired by the GPL *C*, but very lean, defined by just 12 non-terminal EBNF rules. For our DSL, we imitate the concise graph description of DOT using an arrow operator to define edges between nodes.

Cypher[7] is a declarative query language for graph data bases, originally developed for *neo4j*, but now open source. It follows the property graph data model: nodes and relationships have properties, which can be queried. This allows to define filters by pattern matching at the relationship-level (graph edges) of entities (graph nodes). We integrated this concept for message routing purposes in the new DSL.

Apache Kafka[8] is a high throughput messaging system based on a distributed transaction log. The Apache Kafka Streams DSL is an extension, which aims to ease the work with standard data transformations, especially for beginners. With the DSL one can define the logical data processing workflow of applications. Therefore, DSL users define data input types, planned transformations, and targeted output types. By this means, the Kafka Streams DSL follows a similar basic message-oriented concept as we insinuated for the Sidl DSL. Although the Streams DSL facilitates the use of the Streams Processor API, it is sill quite hard to understand for non-software developers as it follows a function chaining approach, comparable to the C# data query language LINQ. Moreover, transformations are more or less considered to be simple data wrangling operations, not the call of runtime-intensive computations. Most importantly however, the Streams DSL is tailored to and only available for Apache Kafka, while our approach is to remain technology/tool-agnostic as far as possible.

3. Method: Developing the Structured Interaction Description Language (Sidl)

Inspired by the advantages and disadvantages of several similarily situated GPLs and DSLs, as summarized in the latter section, we decided to develop a new language, tailored to the approach outlined in section 1. In the following, we detail grammar, development process and tool support of the language: *Structured interaction description language (Sidl)*. All presented components, including the most current development progress, are made available on GitHub[9].

3.1. Language Syntax

With the *Sidl* code listings in this subsection, we want to highlight the most important syntactic constructs of the developed language. We do not focus on the underlying EBNF lexer tokens and parser rules, but show sample code instead, as it is more expressive. For details regarding the EBNF, the reader is referred to the stated GitHub repository. The listings implement a real-world case study, drawn from a recent publication of Zenisek et al. (2023). Therein, a small set of private houses, equipped with photovoltaic panels and battery packs, connected to a public power grid, are modeled as a directed graph in a hard-coded simulation. This *prosumer*-network has

3 https://www.json.org/, ECMA-404 (2017)
4 https://yaml.org/, YAML-1.2 (2021)
5 https://www.lua.org/
6 https://graphviz.org/doc/info/lang.html
7 https://opencypher.org/
8 https://kafka.apache.org/
9 https://github.com/prescriptiveanalytics/Sidl

been setup to test several what-if scenarios regarding potentials and pitfalls of energy communities under varying conditions and hence, provides a suitable use case for the Sidl DSL and its future application context.

In Listing 1, we present several language basics, such as named scopes (cf. C# namespaces), the assignment of constants and the use of comments. Furthermore, the listing shows how to use atomic and complex types to arrange data using `structs`.

```
1   basics { # begin of scope
2     name = "ProducerPowerGrid (PPG)"
3     host = "localhost"
4     ... # port, connection details etc.
5   } # end of scope
6   ...
7   typedef string datetime # type alias
8
9   struct position { float lat, float lng }
10  struct weatherData {
11    float globalRadiation,
12    float airTemperature, float humidity
13  }
14  struct inverterData { float pvPowerProduced,
15    float powerConsumed, float batterySOC }
```

Listing 1. Basic language data structures: line 1: named scopes, line 2: assigning values to basic constants, line 7: type aliasing, line 9: data containers.

With Listing 2 we already focus on the Sidl specifics: First, we set up `message` containers, which may be used to communicate between nodes, i. e., the edges of the envisaged graph. Next, using the `nodetype` statement, we show how to define node classes, which represent the template of simulation or surrogate models.

```
1   import "https://spa.io/reps/ppg/basics.sid"
2   ...
3   message forecastRequest { position pos, datetime dt }
4   message weatherReport { weatherData rep,
5     topic int zip }
6   message inverterReport { inverterData rep }
7   ... # messages for simulationSetup and trafo
8
9   nodetype weatherService {
10    input forecastRequest fr
11    output weatherReport wr
12    property string provider
13  }
14
15  nodetype inverter {
16    input weatherReport wr
17    output inverterReport ir
18    property position pos
19    property int zip
20    ... # more system properties
21  }
22  ... # nodetypes for simulationSetup and trafo
```

Listing 2. Graph component type definitions: line 1: import statement to include other Sidl texts at this point, line 3: message type (i. e., edge) definitions, line 9: node type defintions.

In Listing 3 the previously defined `nodetypes` are now instanciated using the `node` keyword, the nodetype name, an instance name and a named value list for all nodetype properties. With the arrow operator (`-->`) connections between compatible nodes can be established. By this means, output `messages` from the left-hand side `node` are from now on transferred to the input of the right-hand side `node`.

```
1   import "https://spa.io/reps/ppg/types.sid"
2   ...
3   node weatherService wsMain (provider = "zamg")
4   node inverter system1 (
5     pos=system1Position,
6     zip = 4470, ...)
7   node inverter system2 (...)
8   ... # instantiations for simulationSetup and trafo
9
10  sim --> wsMain
11  wsMain --> system1, system2, system3
12  system1 --> trafo
13  system2, system3 --> trafo
```

Listing 3. Graph component instantiations: line 3: node instantiations, line 10: node connections (i. e., edge instantiations).

With the latter listing the core graph model is complete. In order to enable surrogate modeling we introduce a machine learning method scope via an `import` statement on top of Listing 4. By selecting a library method via the dot operator, defining a surrogate name and using the `imitate` keyword, a surrogate for a particular node instance can be mandated. Within an optional scope one may provide parameters for the underlying machine learning algorithm.

```
1   import "https://spa.io/reps/ppg/instances.sid"
2   import "https://spa.io/reps/methods:latest" as m
3   ...
4   m.symReg surrogate1 imitate system1
5   m.symReg surrogate2 imitate system2 {
6     treeSize = 50,
7     grammar = "[+, -, *, /, log, htan]"
8   }
9   m.privacyPreservingRFReg surrogate3 imitate system3 {
10    m = 0.5,
11    r = 0.3,
12    trees = 100,
13    obfuscationRuns = 10
14  }
```

Listing 4. Surrogate modeling: line 2: importing the latest package version of machine learning methods, line 4 and 5: command to create a surrogate model for an existing node using the symbolic regression ML method, line 9: surrogate modeling command using a privacy preserving ML method.

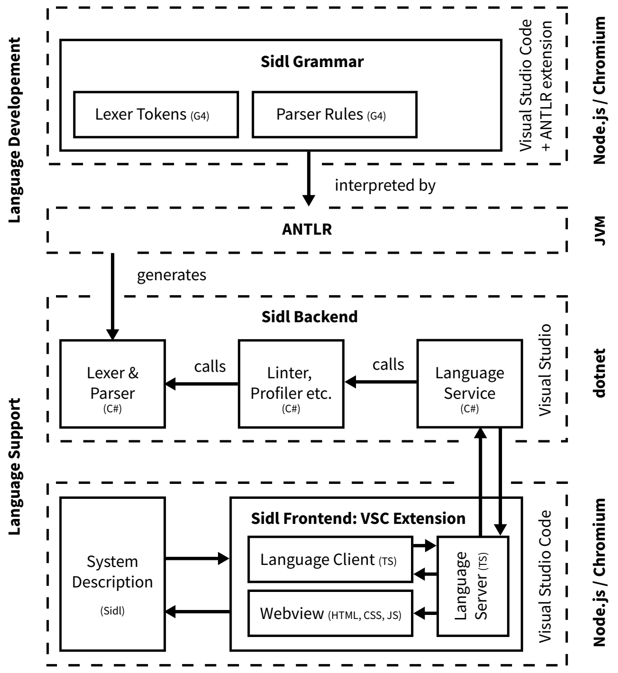


Figure 3. Software architecture of language development and support: The dashed-line boxes represent the software environments in which the tools and packages are run in. The components inside the upper two boxes have been used for *language development*, the lower two correspond to *language support*. To understand the implemented architecture from a language development perspective, one can follow from grammar development on top, via the ANTLR software tool, to the generated lexer and parser. From the viewpoint of a language user, one should focus on the lower figure part, starting with the *Sidl*-system description, which uses the generated lexer/parser via a Visual Studio Code extension and a backend service.

3.2. Language Development and Support

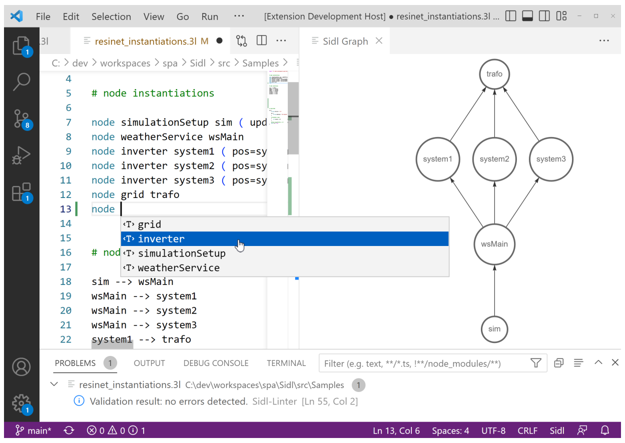
In order to develop our DSL, we utilized several open source tools and created extensions to provide language support for future users. Figure 3 depicts an overview of the installed toolchain. Therein, ANTLR[10], a powerful parser generator, represents the development centerpiece in the upper figure part. To define the language grammar, i. e., lexer tokens and parser rules, we used the ANTLR-EBNF format *G4*, which is supported by the available ANTLR extension for Visual Studio Code (VSC)[11] – a popular, versatile source code editor. Once having developed a valid ENBF, ANTLR allows to generate respective lexer-/parser-code for a variety of popular programming languages, such as *Java*, *Python* or *C#*, from which we selected the latter.

The lower part of Figure 3 depicts our developments for language support. To work with the new DSL, domain experts should be able to use a code editor of their choice, which is why we implemented the *Language Server Protocol (LSP)*[12]. LSP enables to pack most parts of language support into a server application, which is tool-agnostic. This server can be subsequently used by arbitrary, tool-dependent clients, which can be kept quite simple, as they only translate user actions to server calls. As a first example, we developed a language client for VSC using TypeScript (TS). Also the agnostic server part is written in TypeScript, so that both can be packed into one VSC extension, although the server could also be run as a standalone, communicating with another LSP client. As a third component, the extension includes a webview to visualize the written system graph. The language client provides syntax highlighting and sends the current program text to the server after each keystroke of a user. In our particular case, we use the server to call a central REST service for the actual program validation. The server itself is dedicated to pick the "right" service endpoint and subsequently, translate its response, such that either debug information, code suggestions, webview updates or similar is displayed. Figure 4 shows a screenshot with the respective graphical output inside the VSC extension. The referenced REST service belongs to our developed Sidl backend suite. This includes a linter for static code analysis – the actual program validator during Sidl writing – and a profiler for runtime analysis. Both create a scoped symbol table – a data structure to persist the state of a program, with all current values. Therefore, both make use of the generated lexer/parser code, which closes the loop to language development. The Sidl backend is entirely *dotnet*-based and can be run on a remote machine or on the same machine as the VSC extension. One future advantage of a remote backend might be that new language features could be rolled out without the necessity to update the extension. In our current setup, however, the backend is booted locally when running the VSC extension. For the final version of the extension, we plan to pack and deploy it together with a lean, local version of the Sidl backend. The extension user can then decide to use the locally available version or configure one that is running somewhere remote.

4. Evaluation and Application

Evaluating a language, whether GPL or DSL, is a non-trivial task. In this section we provide several metrics to quantify language characteristics of Sidl to some extent. We compare the new language, to the configuration language alternatives Json and Yaml using a set of increasingly long sample texts and the developed metrics. After that,

10 https://www.antlr.org/, Parr and Quong (1995)
11 https://code.visualstudio.com/
12 https://microsoft.github.io/language-server-protocol/



```
# node instantiations

node simulationSetup sim ( upd
node weatherService wsMain
node inverter system1 ( pos=sy
node inverter system2 ( pos=sy
node inverter system3 ( pos=sy
node grid trafo
node
          grid
          inverter
# nod     simulationSetup
          weatherService
sim --> wsMain
wsMain --> system1
wsMain --> system2
wsMain --> system3
system1 --> trafo
```

```
resinet_instantiations.3l C:\dev\workspaces\spa\Sidl\src\Samples 1
Validation result: no errors detected. Sidl-Linter [Ln 55, Col 2]
```

File
Edit
Selection
View
Go
Run
[Extension Development Host] ● resinet_instantiations.3l ...
resinet_instantiations.3l M
Sidl Graph
C: > dev > workspaces > spa > Sidl > src > Samples >
trafo
system1
system2
system3
wsMain
sim
PROBLEMS 1
OUTPUT
DEBUG CONSOLE
TERMINAL
Filter (e.g. text, **/*.ts, !**/node_modules/**)
main*
0 0 1
Ln 13, Col 6
Spaces: 4
UTF-8
CRLF
Sidl

Figure 4. Screenshot of the *Sidl* Visual Studio Code extension in action: Left hand side the language editor with a sample Sidl text is depicted. The editor supports syntax highlighting and – if connected to the language server – code suggestions and completion (cf. drop-down box), as well as error highlighting via underlining. On the right-hand side the graph webview is shown, which is synchronized with the editor and constantly updated by the Sidl backend service. In the terminal window on the bottom one can observe validation feedback from the language server, such as errors, warnings or information messages.

we provide a software design chart regarding the future Sidl application context, we planned. Finally, we critically analyze current limitations of the developed language.

4.1. Experiments and Results

For our experiments we continued with the example from the latter section 3: We extended the sample text with an increasing number of inverter nodes, starting with a single node and finishing at 100 000, representing additional photovoltaic systems. We connected all of them to the described simulation loop node and the power grid node, forming a large energy community. Obviously, writing the description of a system consisting of more than e. g., 100 node instantiations by hand is not a usual real-world task. However, for future versions of Sidl we consider the possibility to automatically generate nodes, e. g., with an import routine on a connected data base. Working on such (partly) generated Sidl texts should still be efficient.

After generating the texts, we configured the following first test setup: Each text is parsed 1000 times for which the runtime is measured individually. The first 20% of all runs are declared as "burn-in-phase", i. e., time to let the computer processor reach an optimal plateau of performance (cf. activate a *performance core* instead of an *efficiency core*), and thus, discarded. According to custom tests and credited by scientific studies, e. g., Bovet et al. (2018), we assume an average typing speed of 50 Words per minute, which is 4.5 characters per second or 175 milliseconds per character. Hence, while writing the program text, the language support backend has to parse the text every 175 millisecond on average. Therefore, in between the runs, we command the current thread to sleep between 150 and 200 milliseconds (uniformly distributed) to simulate a realistic user-language support interaction scenario. The goal of this initial experiment is to show up to which count of nodes, the parsers for the tested languages still execute in time (e. g., $\leq$ 200ms). For Json, we used the native dotnet parser, for Yaml we installed the very popular *YamlDotNet* parser from nuget[13]. Obviously, for Sidl we used our custom-made language backend for parsing, in particular the *Linter* which is capable of creating and validating a scoped symbol table. In this test, we focus on the parsing time only, while time to transmit the program texts and call the parsing methods is not considered. The reason for this is that the test is about the language performance, and thus only their parsers vary for the experiments. The transmission setup remains the same. In our current setup, with the backend running on the same machine, transmission and call times do not have a big impact on the overall runtime, as we assessed in a series of 1000 test runs: We measured $\leq$ 2ms on average, no matter how long the text is, which would take only a little share of the assumed 200ms threshold. However, future tests could include measuring transmission runtime as well as memory profiling on different backend setups (remote, local, different transport protocols, etc.). The measured median runtimes in milliseconds for Sidl, JSON and YAML texts are shown in Figure 5.

The runtime of all three language parsers grow almost linearly with increasing node count, after a brief initialization phase: The text basics – i. e., definition of structs, nodetypes, messages etc.– loose importance with increasing node count. The dotnet-native JSON parser is the fastest by far. After a minor performance advantage at low node count for the Sidl parser, both Yaml and Sidl perform almost equally. The horizontal, dashed, black line indicates our custom, virtual feasibility threshold. For Yaml and Sidl 3000 nodes can be parsed "in time", for JSON the limit is at about 40 000 nodes.

The second language evaluation test is a simple character count for each of the generated sample texts. The results are depicted in Figure 6. The plot shows that in this test category, Sidl outperforms the other two, which have very similar results. The narrowed focus of the Sidl DSL enables authors to express more terse than with general-purpose configuration languages. This brevity makes Sidl also interesting as potential payload format for transport

13 https://github.com/aaubry/YamlDotNet/wiki

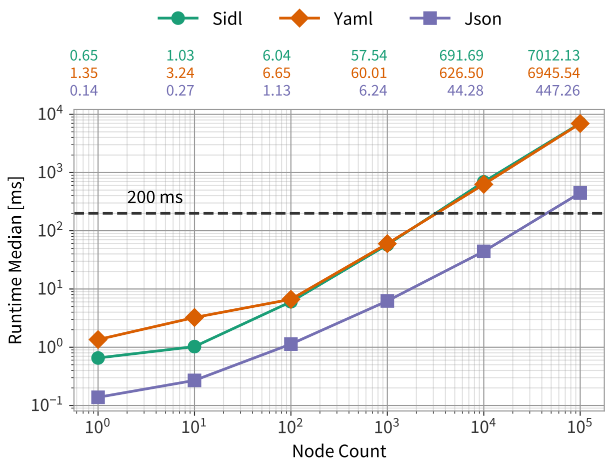


Figure 5. Runtime comparison: The tested node counts are given on the x-axis, respectively measured median runtimes on the y-axis. Both axes use logarithmic scales. Additionally, the exact runtimes per node count are given on top of the chart for each of the used configuration languages with the respective color. The horizontal, dashed line marks an assumed mean timespan between requests to the language backend.

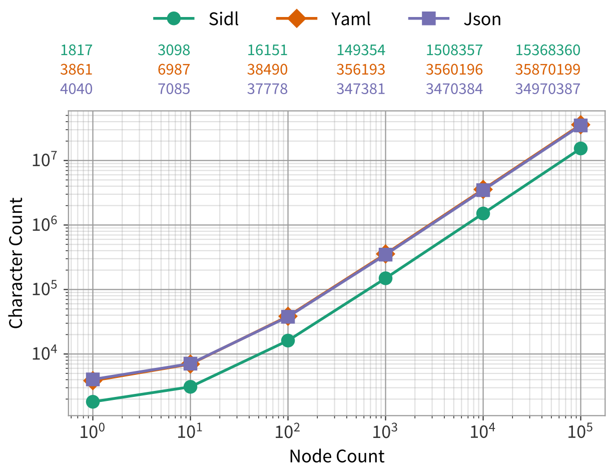


Figure 6. Character count comparison: Apart from Figure 5, here the overall count of characters of the text samples is plotted on the y-axis. Due to high correlation of Json and Yaml results, the latter series is almost hidden.

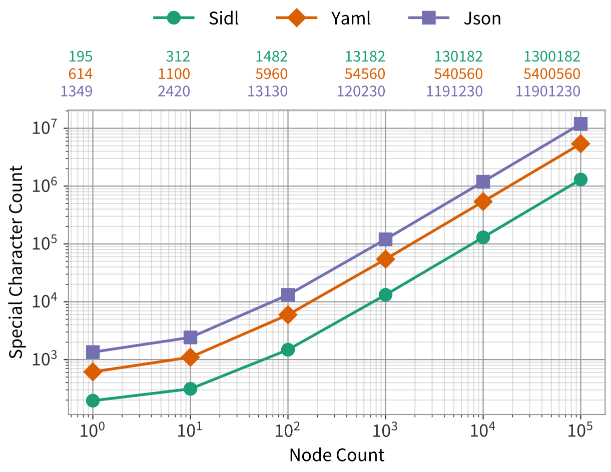


Figure 7. Special character count comparison: Apart from Figure 5, here the overall count of used special characters (non-alphanumeric characters) is plotted on the y-axis.

protocols, if messages should be serialized in a human readable form.

The last calculated metric is an extension to the latter character count: We modified this metric by counting special characters only – results are provided in Figure 7. In this test, Sidl clearly outperforms the other languages again, with Yaml ranking second and Json third. We did not count Sidl keywords as special characters, since then every Yaml and Json identifier must have been counted as well. The test should only demonstrate, how "natural" the program may be written, by considering special characters (non-alphanumeric) as artificial symbols. Those are much more cumbersome to write and read, especially for non-software developers.

All tests have been performed on a state-of-the-art consumer notebook with a *12th Gen Intel Core i7-1265U* processor and single-threaded settings as this represents a realistic working environment for domain experts. The software application which we developed for these experiments, including text generation and the follow-up tests, is available in the stated GitHub repository of the Sidl language.

4.2. Language Application Context

With the results shown in the last subsection, we were able to demonstrate the competitiveness of our language in terms of parsing runtime, brevity and writing simplicity. In this subsection, we aim to detail how these characteristics may be beneficially used and present the final application context of our developments for this purpose. Figure 8 sketches the planned software architecture, which covers Sidl language support components, as well as the future Sidl runtime and the Sidl-based integration of simulation and surrogate models. The figure does not show language development, which is considered to be done at this point.

Domain experts (e. g., from production industry, cf. bottom of Figure 8) as well as modeling experts (i. e., for simulation and machine learning, cf. top of Figure 8) are the target users of the Sidl language, while the runtime environment is currently developed by the authors of this work (cf. middle of Figure 8). Most of the previously described language support components represent the domain experts' view on Sidl. So far, we detailed a palette of *description* features for the planned simulation-, surrogate and optimization system, presented in section 1, which we

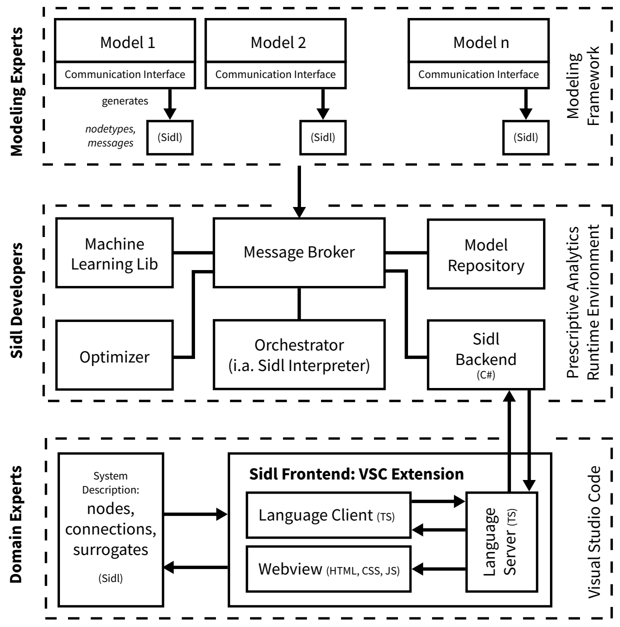


Figure 8. Software architecture of the planned application context of Sidl: In the bottom dashed-line box, one can find the described language frontend from a domain expert's point of view. In the middle one, the developments of the Sidl creators for language support (cf. *done*) and interpretation (cf. *work in progress*) are depicted. The box on the top includes a planned communication interface for simulation modeling experts, which will enable to generate Sidl type definitions instead of writing the statements by hand.

labeled here as *prescriptive analytics runtime environment*.

In order to use Sidl for actually configuring such a system, language *interpretation* is necessary. The software components, responsible for this, are currently under development and shown in the middle figure part. Therein, one can observe a central message broker, which is responsible for the communication between all components. The actual Sidl interpretation is done by the *Orchestrator*. This component is responsible for booting computation nodes for each of the described models in the Sidl texts and managing them in a repository (cf. *Model Repository*). Furthermore, it also handles the communication between the models by performing the message subscriptions. Surrogate modelling is triggered by the *Orchestrator* via access to an arbitrary *Machine Learning Library*. Furthermore, the *Orchestrator* connects the instantiated model graph to an *Optimizer*, which enables to perform the envisaged search for system improvements.

In addition to the runtime environment, we are currently developing a software library, to support modeling experts, the second Sidl user group. The library is a lean messaging client, which shall be used for each model (i. e., later node) in a Sidl-described graph. By using the library, the nodetype and message information can be derived via type reflection and the Sidl texts can be generated automatically for each model. These model descriptions can be subsequently used for instantiating and connecting nodes by the domain experts via the Sidl backend in the runtime environment. In summary, with the presented and current developments, we aim to bridge the gap between domain and modeling experts, i. e., pursuing the principle of Industry 5.0: to keep the human in the loop (Emmanouilidis et al., 2019).

4.3. Limitations

Obviously, the pending implementation of the actual language interpretation and the respective application context represents the major current limitation of the developed DSL. At this point, the language, however, may be used as a planning tool for designing the communication flow in distributed software applications. For this purpose, Sidl and the accompanying language support may replace UML activity diagrams, as it is more maintainable, than a graphical approach.

Another limitation concerns the sequence of communication: Sidl enables to describe a system as directed graph, i. e., system components and their interactions via messages. But it does not cover the aspect of messaging sequence or simulation time. This limitation has been made by choice, mainly because Sidl does not aim to describe the inner workings of system components (i. e., nodes) and hence, cannot be used to define the exact sequence, of when messages will be sent or received. However, future language applications may lead the development to cover this aspect to some extent. For instance, to enable surrogate modeling of nodes, one usually has to define input features and an output target, which already implies a certain sequence of messages. Currently, machine learning algorithms would consider all input to model a node's output. To enable adjustments, the language needs to be further developed.

5. Conclusion and Outlook

In this work, we presented a new domain specific language, called *Structured interaction description language (Sidl)*, to model complex systems as directed graph of communicating system components. The languague also enables the definition of machine learning based surrogates for individual components. This should accelerate the evaluation speed of the modeled system graph, when it is used for optimization. The paper includes details regarding the language development toolchain, the resulting syntax and the developed support software. Furthermore, we showed the competitiveness of our language concerning parsing runtime, and its superiority

concerning language brevity and simplicity compared to other configuration languages. One major benefit of the developed language is that it allows a separation of concerns between domain experts and modeling experts, while having a brief, easily extendable syntax, all in one language. The presented and further results are available as open source software in our GitHub repository:
https://github.com/prescriptiveanalytics/Sidl.

So far, the language enables the *description* of how a system should be modeled. This process is already well supported by the developed software tools. The actual *interpretation* of the resulting texts, however, is in the focus of ongoing work. In the latter section 4, we provide a sketch of the software architecture to realize this plan. Apart from language interpretation, we look forward to further enhance the language itself: One element missing is the possibility to configure optimization algorithms with Sidl. With this feature, domain experts shall be enabled to define optimization goals, constraints or a maximum runtime budget, without having to know the optimization algorithm or tool used.

6. Acknowledgements

The presented work was done within the project "Secure Prescriptive Analytics", which is funded by the state of Upper Austria as part of the program "#upperVISION2030".

References

Bovet, S., Kehoe, A., Crowley, K., Curran, N., Gutierrez, M., Meisser, M., Sullivan, D. O., and Rouvinez, T. (2018). Using traditional keyboards in vr: Steamvr developer kit and pilot game user study. In *2018 IEEE Games, Entertainment, Media Conference (GEM)*, pages 1–9. IEEE.

ECMA-404 (2017). *ECMA-404 Standard: The JSON data interchange syntax*. Ecma International, Geneva, Switzerland.

Emmanouilidis, C., Pistofidis, P., Bertoncelj, L., Katsouros, V., Fournaris, A., Koulamas, C., and Ruiz-Carcel, C. (2019). Enabling the human in the loop: Linked data and knowledge in industrial cyber-physical systems. *Annual reviews in control*, 47:249–265.

Fourer, R., Gay, D. M., and Kernighan, B. W. (1990). A modeling language for mathematical programming. *Management Science*, 36(5):519–554.

Gansner, E. R. and North, S. C. (2000). An open graph visualization system and its applications to software engineering. *Software: practice and experience*, 30(11):1203–1233.

Gosavi, A. et al. (2015). *Simulation-based optimization*. Springer.

Greenberg, S. (1972). *GPSS Primer*. Publisher Wiley & Sons Canada, Limited, John.

Henriksen, J. O. (1996). An introduction to slx. In *Proceedings of the 28th conference on Winter simulation*, pages 468–475.

Ierusalimschy, R., De Figueiredo, L. H., and Filho, W. C. (1996). Lua – an extensible extension language. *Software: Practice and Experience*, 26(6):635–652.

Koziel, S. and Leifsson, L. (2013). *Surrogate-based modeling and optimization*. Springer.

Miller, J. A., Han, J., and Hybinette, M. (2010). Using domain specific language for modeling and simulation: Scalation as a case study. In *Proceedings of the 2010 Winter Simulation Conference*, pages 741–752. IEEE.

Parr, T. J. and Quong, R. W. (1995). Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810.

Pitzer, E. and Kronberger, G. (2015). Concise supply-chain simulation optimization for large scale logistic networks. *Computational Intelligence and Efficiency in Engineering Systems*, pages 429–442.

Werth, B., Pitzer, E., Backfrieder, C., Ostermayer, G., and Affenzeller, M. (2019). Surrogate-assisted microscopic traffic simulation-based optimisation of routing parameters. *International Journal of Simulation and Process Modelling*, 14(3):223–233.

YAML-1.2 (2021). *YAML Ain't Markup Language (YAML) version 1.2, Revision 1.2.2*. YAML Language Development Team.

Zenisek, J., Bachinger, F., Pitzer, E., Wagner, S., and Affenzeller, M. (2022). Utilizing interpretable machine learning to detect dynamics in energy communities. *34th European Modeling and Simulation Symposium, EMSS 2022*.

Zenisek, J., Dorl, S., Falkner, D., Gaisberger, L., Winkler, S., and Affenzeller, M. (2023). Shapley value based variable interaction networks for data stream analysis. In *Computer Aided Systems Theory – EUROCAST 2022*, volume 13789, pages 130–138. Springer Nature Switzerland.