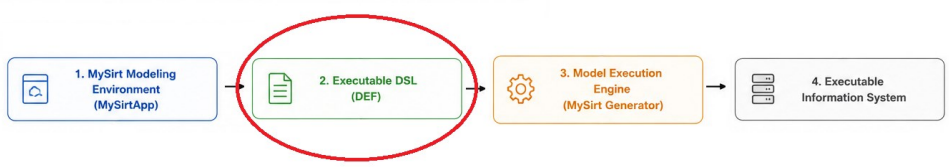
MySirt Execution Engine

MySirt DSL Design Principles



Technical Documentation
Version 1.1 – July 2026

Prepared by
MySirt Team

Author
Leda Santosuosso

Document Information

Document	MySirt DSL Design Principles
Product	MySirt
Version	1.1
Date	July 2026
Document Maintainer	Leda Santosuosso
Status	Technical Reference

Preface 1
1. Why We Model Reality 1
2. From Models to Software Systems 2
3. Modeling Languages 2
4. The Metamodel 3
5. The MySirt DSL 4
6. Domain-First Modeling 5
7. Single Source of Truth 6
8. Describe Only What Cannot Be Derived 6
9. Implications of the Design Principles 7
9.1 Eliminating Redundancy 7
9.2 A Unified View of the System 7
9.3 Model Evolution 7
10. Measuring the Value of the DSL 8
11. Final Remarks 8

Preface

The design philosophy behind the MySirt DSL is the result of several years of experimentation, including a comparative analysis of existing low-code and model-driven platforms conducted during the early stages of the research.

Over time, the focus evolved from usability and code generation toward a more fundamental objective: minimizing the explicit knowledge required from the designer while maximizing the knowledge automatically derived by the Model Execution Engine.

1. Why We Model Reality

The ability to build models is one of the fundamental characteristics of human thinking. Whenever we represent reality through a map, an architectural blueprint, or a dressmaking pattern, we are creating a model.

A map represents a geographical area without reproducing every detail. An architectural blueprint represents a building before it is constructed. A dressmaking pattern defines the shapes and dimensions required to produce a garment, but not the fabric from which it will be made. In all these cases, the model is not the reality itself; it represents only those aspects that are relevant to the purpose for which it has been created.

A model is therefore an abstract representation of reality. Its value does not lie in its completeness, but in its ability to highlight the information needed to address a specific problem while omitting what is irrelevant in that particular context. The same real-world object can be described through different models, each designed to serve a different purpose.

Information systems can also be regarded as models of reality. They represent people, documents, products, processes, organizational rules, and relationships that characterize a specific application domain. The purpose of an information system is not to reproduce reality in its entirety, but to provide a representation that is sufficiently faithful to allow users to operate effectively within that domain.

Designing an information system therefore begins with identifying which elements of the domain should be represented and which can be omitted. Only afterwards are these models translated into a software solution.

If an information system originates as a model of a domain, then the fundamental challenge is not only how to implement it, but also how to describe that model in a way that is precise, consistent, and machine-processable. It is from this need that modeling languages—and, more broadly, the Model-Driven Engineering approach—have emerged.

2. From Models to Software Systems

If an information system represents a model of an application domain, a natural question arises: is it possible to build the software directly from that model?

For many years, the answer was only partial. Models were used primarily during the analysis and design phases, while the actual system was developed through manual coding. As development progressed, the model and the software often diverged, making it increasingly difficult to keep them consistent.

Model-Driven Engineering (MDE) was introduced precisely to overcome this separation. In this approach, the model is no longer merely a documentation artifact; instead, it becomes the central element of the development process. The software is produced by automatically processing a formal model of the domain through a series of model transformations.

This does not eliminate the need to design and develop software, but it shifts the focus toward defining the model and the transformation rules that enable it to become an executable system. As a result, the model becomes the primary carrier of domain knowledge, while a significant portion of the implementation activities can be automated.

For this to be possible, the model must be expressed in a rigorous, unambiguous form that can also be understood and processed by software. This is the role of modeling languages, introduced in the following chapter.

3. Modeling Languages

For a model to be interpreted and processed automatically, it is not sufficient for it to be understandable to humans; it must be expressed through a formal language with precise rules and no ambiguity.

A modeling language therefore provides the means to describe a model. Like any language, it offers a set of expressive elements and syntactic rules that enable the consistent representation of the aspects that are relevant to a specific domain.

Modeling languages can be graphical, where the model is represented through diagrams and symbols, or textual, where the description is expressed using a textual notation. Although they employ different forms of representation, both pursue the same objective: to describe a model in a rigorous, consistent, and machine-processable way.

When a modeling language is designed to describe a specific domain, it is referred to as a **Domain-Specific Language (DSL)**. Unlike programming languages, its purpose is not to develop algorithms or general-purpose applications, but to provide a natural and effective way of representing the concepts of a particular application domain.

4. The Metamodel

In order to describe a model, it is necessary to define which concepts can be used and which rules must be respected. This set of concepts and rules is known as the **metamodel**.

A metamodel does not describe a specific system; rather, it defines the language through which that system can be represented. In other words, it provides the conceptual structure that all models within a given domain must conform to.

A useful analogy is that of music.

Musical notation defines the available symbols (notes, rests, clefs, accidentals) and the rules governing how they can be combined. A musical score, on the other hand, is a concrete model written using that notation. The score does not describe the rules of music; it uses them to represent a specific composition. Finally, the performance of the score transforms that description into music.

The same principle applies to software systems.

The MySirt DSL provides the language through which a model conforming to the platform's metamodel is expressed. The result is a **DEF** file, representing the model of a specific application domain. The **Model Execution Engine** interprets this model and transforms it into an executable information system.

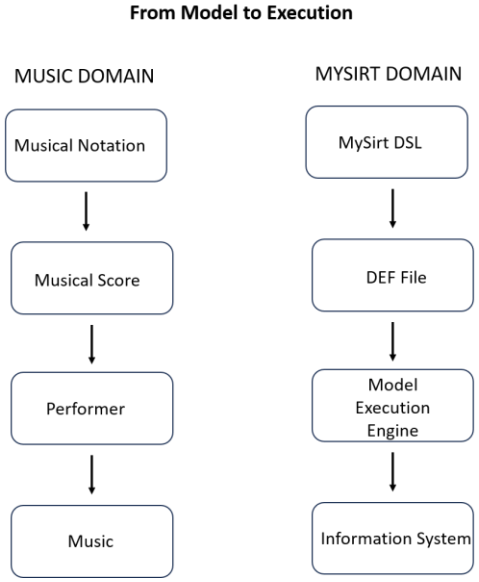


Fig. 1 Analogy between musical notation and the MySirt DS.

Musical notation is not a random collection of symbols, but a language defined by a set of concepts and rules. Similarly, the MySirt DSL is defined by a metamodel that specifies which constructs can be used and how they can be combined to describe an application domain. The musical score and the DEF file are, in turn, models that conform to their respective languages.

5. The MySirt DSL

The MySirt DSL is a **Domain-Specific Language (DSL)** designed to describe the model of an application domain.

The language enables the representation of the domain's relevant concepts, their relationships, behaviors, organizational rules, and constraints that characterize the system to be developed. Its purpose is not to describe the application's source code or technical architecture directly, but to express, in a structured form, the knowledge required to represent the application domain.

The DSL is founded on a metamodel that defines:

- which concepts can be represented;
- which properties they possess;
- how they can be related to one another;
- which rules and constraints they must satisfy;
- the meaning associated with each element of the model.

These concepts can be expressed through the MySirt modeling tools and translated into a textual representation that can be processed by the execution engine. The **DEF** file constitutes this representation: it describes a specific application model that conforms to the rules of the language.

It is therefore important to distinguish between:

- **Metamodel**, which defines the conceptual structure and the rules of the language;
- **DSL**, which comprises the concepts, semantics, and representation forms through which a model can be expressed;
- **DEF**, which represents a specific application model written in a form that can be interpreted by the **Model Execution Engine**.

The **Model Execution Engine** processes this description together with the general knowledge embedded in the engine itself, deriving the artifacts and behaviors required to realize the executable information system.

6. Domain-First Modeling

One of the fundamental principles of the MySirt DSL is the separation between the application domain and the software implementation.

During the modeling process, the designer does not describe how the application should be built, but rather the concepts that characterize the domain to be represented. The focus is placed on the business concepts, their relationships, the rules that govern them, and the processes that define their behavior.

Consequently, the DSL does not require the specification of technology-specific aspects, such as the database structure, Java classes, application services, user interface implementation, or other implementation details. These elements do not belong to the problem domain, but to the software solution derived from it.

In MySirt, software is therefore regarded as a consequence of the model, not as its content. The designer describes the application domain, while the **Model Execution Engine** uses that description to automatically derive the artifacts required to realize the executable information system.

This approach keeps the model independent of implementation technologies and allows the design effort to focus on the aspects that truly represent domain knowledge.

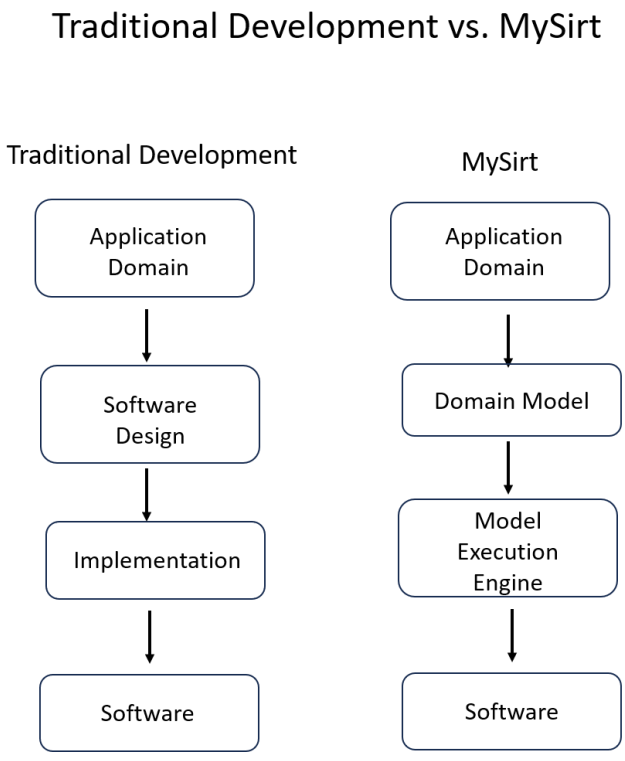


Fig.2 Comparison between the traditional software development process and the MySirt domain-first approach.

7. Single Source of Truth

Since the designer describes only the application domain, that description naturally becomes the central source from which the entire system can be derived.

One of the fundamental principles of the MySirt DSL is that the entire system is derived from a single description of the application domain.

The model defined through the DSL represents the single authoritative source of the knowledge required to describe the system. Each piece of information is declared only once and becomes available for all subsequent processing stages.

This approach eliminates the need to maintain duplicate descriptions of the same information across different application components. The consistency of the system does not depend on synchronizing multiple artifacts, but on the fact that all of them are derived from the same domain representation.

As a result, the evolution of the system coincides with the evolution of the model. Every change is made exclusively to the DSL model; the **Model Execution Engine** then uses the updated model to automatically regenerate the artifacts that constitute the executable information system.

This principle not only simplifies the development process, but also improves the system's understandability and maintainability, since domain knowledge remains concentrated in a single, coherent artifact.

8. Describe Only What Cannot Be Derived

One of the primary design objectives of the MySirt DSL is to minimize the amount of knowledge that the designer must describe explicitly.

The goal of the MySirt DSL is not to minimize the amount of work, but to minimize the amount of explicit domain knowledge required to describe a complete application domain.

The model is not intended to represent every detail of the software application, but only the information that characterizes the application domain and cannot be derived automatically.

The general knowledge required to build the system is not repeated within the model; instead, it is embedded in the **Model Execution Engine**. Based on the domain description, the engine automatically derives the artifacts and behaviors required to realize the executable information system.

For example, the definition of a **Business_Document** may enable the automatic derivation of the data structure, management operations, application services, security rules, auditing mechanisms, and many other elements of the application. These aspects are not described explicitly in the DSL because they represent knowledge that the engine is already capable of deriving.

From this perspective, the model does not represent everything that constitutes the system, but only the knowledge that the **Model Execution Engine** cannot derive autonomously. Everything else is the result of the derivation process.

9. Implications of the Design Principles

9.1 Eliminating Redundancy

One of the direct consequences of the MySirt approach is the elimination of redundant system descriptions.

Since the DSL represents the single authoritative description of the application domain, each piece of knowledge is defined only once. The same information does not need to be replicated across multiple models or implementation artifacts.

As a result, changes are made exclusively to the domain model. The **Model Execution Engine** automatically regenerates the affected software artifacts while preserving the overall consistency of the system.

9.2 A Unified View of the System

Concentrating the application's knowledge within a single domain model also improves system understanding.

Instead of reconstructing the behavior of the application by navigating source code, configuration files, database schemas, and documentation, the designer can understand the overall structure of the system by examining a single model.

In this sense, the DSL is not only a language for system generation, but also a valuable artifact for documentation and communication.

9.3 Model Evolution

In MySirt, the evolution of the software system coincides with the evolution of its domain model.

New requirements are addressed by modifying the DSL model rather than manually updating multiple software components. Once the model has been updated, the **Model Execution Engine** derives a new, consistent version of the system.

This approach reduces the risk of inconsistencies and ensures that the implementation remains aligned with the application domain throughout the system's lifecycle.

10. Measuring the Value of the DSL

Domain-Specific Languages and Model-Driven platforms are traditionally evaluated using metrics such as the number of lines of generated code, the percentage of automatically generated code, or the reduction in development time. While these metrics describe the results achieved, they do not explain how effectively a language represents the application domain.

The design philosophy behind MySirt suggests a different perspective. The value of a DSL depends not only on the amount of code it can generate, but also on the amount of domain knowledge it is able to represent explicitly and on the amount of knowledge that the **Model Execution Engine** can derive automatically.

From this perspective, a DSL can be considered more effective when it minimizes the amount of information that the designer must explicitly provide to describe the application domain completely, while allowing the execution engine to derive automatically everything that constitutes the platform's general knowledge.

This interpretation does not replace traditional productivity metrics. Rather, it complements them with a criterion that is more directly aligned with the fundamental purpose of a modeling language: representing an application domain as completely as possible while requiring the minimum amount of explicit domain knowledge.

This perspective does not propose a new quantitative metric. Instead, it offers a different criterion for evaluating the expressive power of a Domain-Specific Language: the amount of domain knowledge that must be made explicit versus the amount of knowledge that can be derived automatically by the **Model Execution Engine**.

11. Final Remarks

The design of the MySirt DSL is founded on a simple principle: the application domain should be described, while the software should be derived.

This principle influences every aspect of the language. The DSL captures the concepts that characterize the application domain, while the **Model Execution Engine** provides the knowledge required to transform that description into a complete executable information system.

From this perspective, the objective of the DSL is not to minimize the amount of code written by developers, nor simply to maximize the percentage of generated code. Its purpose is to minimize the amount of explicit domain knowledge that must be provided by the designer while maximizing the knowledge that can be derived automatically by the **Model Execution Engine**.

This philosophy proposes a different way of thinking about model-driven development. Rather than describing how software should be implemented, the designer focuses on describing the application domain. The responsibility for implementation is delegated to the **Model Execution Engine**.

For this reason, the MySirt DSL should not be regarded merely as a language for software generation, but as a language for representing application domains in a form that can be made executable.