Implementation of graphical editor using Sirius

David Phung
D13, Lund University, Sweden
dat13tph@student.lu.se

Abstract

One of the first steps in the process of model-driven development is to construct an editor that would allow for creation and modification of models. For domain specific languages, there might not be an existing product on the market that have support for the language. Moreover, the cost of building one such editor can be too expensive if done from the ground up. A number of tools and frameworks have therefore been developed to help alleviate this problem. We present our study of one such frameworks, Sirius. Sirius is an Eclipse project aims at enabling rapid development of graphical editor without high level of required knowledge. By implementing an editor ourselves using the framework, we have highlighted a number of its strengths and the factors that should be considered before a developer unfamiliar with Eclipse modeling projects can start using Sirius.

1 Introduction

Effort has been put into researching and developing model-driven engineering as a way to address the platform complexity problem and to express domain concepts more effectively [3]. To support MDE, a number of Eclipse projects have been conducted to develop frameworks that would facilitate the modeling process. One of the most notable among these frameworks is the Eclipse modeling framework (EMF) which allows developer to describe a model and generate artifacts from it [1]. Based on EMF, GMF is another framework that offers support for developing graphical editors in Eclipse [2].

The Sirius framework is built on top of GMF, serving a similar purpose, that is to support development of editor for modeling languages. The difference between the two is that Sirius hides away the complexity of GMF and eliminate the need for a high level of knowledge in object-oriented programming. This has the effect of enabling rapid development without the need for back-end knowledge [5]. Using Sirius, the developer can graphically define the features and capabilities of an editor, tailor it so that it would suit a specific modeling language. This can be a quite useful solution for a lot of Domain specific languages where creating an editor from the ground up is too expensive.

Sirius's approach is quite interesting as most of the work is done in a graphical editor, in comparison to other tools such as EuGENia, which uses annotation. Work has also been done to compare Sirius with Graffiti with Sirius being evaluated as better for domain specific modeling graphical editors [6]. In this project, we will take a close look at the process of development using Sirius to further evaluate its strengths and weaknesses. We will also highlight the level of required knowledge and the different factors that should be considered before a developer unfamiliar with the Eclipse frameworks can start working with Sirius. The work will be done in an practical fashion, by actually implementing an editor ourselves. An editor for a language heavily inspired by statechart, a state-machine language that supports containment, will be implemented. We will also investigate whether Sirius have support for user-defined type, a feature that could be beneficial for graphical editors.

2 Eclipse modeling framework and Sirius

2.1 Eclipse modeling framework

Aside from being an IDE for programming languages, such as Java, Eclipse also serves as a platform for a number of model-based plug-in projects. An important project among these is the Eclipse modeling framework (EMF) which facilitates the creation of a data model as well as provide code generation support. In particular, EMF allows the developers to describe a data model in one of the three forms, Java interfaces, UML diagram or XML scheme, and then generate the others from it [4]. Here a data model means a model used to represent different concepts, their attributes and relationship to one another in a certain domain, the logic and behavior of the data are often not included. Not only that, EMF also provides generated code for a simple Eclipse-based editor to construct and manage instances of the data model. As an example to further illustrate this, let us suppose that we are working with a small domain specific language used to describe a family tree (the example language in the Sirius tutorials). In the first step we can use EMF to create a data model describing the different concepts in a family tree such as father, mother, children, their attributes and relationship. Then in the second step we can use the generated code and editor to create and modify actual family tree instances. The model that we created in step one is called a meta-model which is often defined as a model of model. A more precise definition also exists in which a meta-model is defined as a model describing the abstract syntax of a language, capturing its concepts and relationship [2].

[1] https://www.eclipse.org/modeling/emf/
[2] https://www.eclipse.org/modeling/gmp/

Course paper, EDAN70, Lund University, Sweden
January 15, 2018.

2.2 Sirius

While the editor generated by EMF is easy obtainable, its tree-based structure and slow working flow make it inefficient for large models, or models in form of graph. There is therefore a need for a way to create editors that can suit different meta-models defined in EMF . GMF, a framework which encapsulates the Graphical editor framework and Draw2D, was built to provide developers with the capabilities of creating graphical editor for EMF models. However, one of its big disadvantages is that it is rather complex and requires a high level of domain knowledge in object oriented programming. The Sirius framework was in turn developed to solve this problem. Sirius is built on top of GMF, encapsulates it and provides the ability to rapidly develop a graphical editor without the need to know about the underlying processes.[5]

It is important to mention that the scope of Sirius is limited to only to the graphical representation specification of data, the data itself must be specified in EMF. To further clarify this, we will continue with our example about the family tree domain language above. Once the abstract grammar of the language has been specified in the meta-model, we can start building family tree instances. Suppose we want to display an instance in a diagram where each family member is represented by a person icon together with a label displaying their name. Obviously we would need to specify information such as how the icon will look like, the label's text size and color etc. These pieces of information only describe how to represent each person graphically and does not at all concern the specification of their attributes and relationships. There is therefore a clear distinction between a graphical- and a data specification. This is also the line separating the scope of Sirius and EMF. To change the definition of data, we must modify the EMF meta-model, to change the graphical representation of data, we must modify the Sirius specification.

In general, the process of creating an editor in Sirius be divided into three main steps: defining the language, specifying a graphical representation and finally create a set of tools to edit the model. The first step is done solely in EMF after which models of the defined language can already be created and modified with help of the generated EMF-editor. In Sirius terms, these model, which describe the domain problem, is called semantic models and its objects semantic objects. In the second step, we specify how to represent the semantic models on the final editor by using another tree-based editor provided by Sirius. In this tree, each node denotes a mapping between a semantic object and its graphical representation. For each mapping, we can specify input information such the shape, color, icon of the graphical representation (in a style element) and additional rules on when the mapping should be applicable. Based on these inputs, Sirius will automatically scan for suitable candidates in the semantic model and uses the styling information to generate graphical objects for them. In the third step, we specify the tools necessary to create and edit the model objects. Similar as in the second step, each tool also has a mapping describing which semantic-graphic mapping it will have effect on and different rules describing when the tool should be applicable. The effects of the tools are defined using model operations which will be performed sequentially in the order they were specified. Sirius supports most commonly used model operations but also allowing developers to define new ones by invoking their own Java methods [3].

2.3 Query languages

When working with Sirius, it is common that the developer will have to provide an interpreted expression as an input to different parameters, such as the rules mentioned above. Sirius supports three default languages to write these expressions AQL, Acceleo and raw OCL and the possibility to define your own custom language. In this project, we made use of only the AQL language. It is a small, simple and fast language that is used to navigate and query an EMF-model [4]. Often it is used in our implementation to produce text, boolean values or a subset of the semantic models during runtime as input to certain parameters. If the expressions become too long and complex, they can be written instead in Java services, which are Java methods conforming to a certain format [5]3.

3 Features to be implemented

3.1 Statechart features

Since statechart is a state-machine language, the basic definition of a program is that it consists of a number of states, events and state transitions. Each state transition happens when a certain event is triggered and takes the system from a state to another. To facilitate the modeling of a large system with large amount of states and state transitions, statechart introduces a special class of state called super state. These are container that affect state transitions in certain ways. To better explain this, we will take an example of an XOR state (figure 1). When two state transitions starting from different sources but end up in the same target when a certain event is triggered, we can put the source states inside an XOR state and combine the two state transitions into one instead. When a state transition ends in an XOR state, the system will be redirected into one of its child states. An AND state is another sub-type of super state that allows the system to be in multiple states at the same time. Aside from these two super states, we will also be implementing support for history states. A history state can only exist inside an XOR state and redirect an entering state transition towards the last state in the XOR state that the system was in.

[3] https://www.eclipse.org/sirius/doc/

[4] https://www.eclipse.org/acceleo/documentation/aql.html

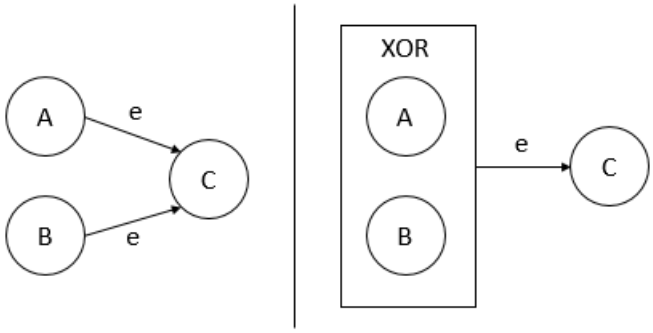

Figure 1. The graph on the left side is equivalent to the right side.

A notable feature of statechart in comparison with normal state machine diagrams is the concept of super states that can contain other states. This creates a number of interesting questions that we will use to investigate the capability of Sirius:

- Is it possible to create graphical objects that can contain others?
- A super state can contain other super state which in turn can contain other states. How can this be done in Sirius? Does it support recursion or is there a limit to the level of nested containment.
- There are a number of semantic restrictions in the features we introduced, such as: a simple state cannot contain another state, there can be only one history state in an XOR state. If these restrictions are not specified in the semantic model (to simplify the model for example), can we impose them on the graphical objects instead?

3.2 Additional feature

Aside from the above mentioned, we will also investigate whether it is possible to have user-defined types in Sirius. User-defined types here means that after the user has defined type, they can quickly create instances of it in other places. In particular, we will attempt to achieve the following:

- Being able to define a type in a separate view.
- Being able to quickly create an instance of a type in the program view.
- If the content of a type is modified in the separated view, the changes will be reflected in the program view, but not the other way around. If the type instance is changed in the program view, the type definition stays the same.

Since we are working with a state-machine language, a type will consists of a number of states and state transitions. Events are declared as global and are available in all scopes.

4 Implementation

4.1 Statechart features

We have defined our meta-model to capture the four main concepts of the statechart features, details can be seen in figure 2.

- Program: Each program contains of a number of states and events. All events are direct children of the program but not all states are.
- State: An abstract class from which other concrete state types can inherit. Each state can contain other states and state jumps. It is inherited by the four state types: SimpleState, XOR_State, AND_State and HistoryState.
- StateJump: Representing a state transition. It has two references pointing to the source and the target states as well as a reference to the event that will trigger the transition.
- Event: An abstract class for the different event subclasses. In this project, we have only implemented two event types: SimpleEvent which represents I/O input and PeriodicEvent which triggers itself after a certain period of time.

Inside the Sirius editor, we have specified our mappings in accordance to table I. Super states are naturally mapped to container elements but even simple states are also mapped to containers instead of regular nodes. This is due to the fact that we wanted to use a gradient style to display simple states but regular nodes do not support this. Using the AQL language, we have defined a number of rules that would limit the set of semantic objects a certain mapping would be applied, as can be seen under the column semantic candidates. Another important detail is the import parameter. It allows developers to reuse mapping specifications as children of a certain container. For example, instead of creating another child element under the XOR container in the tree, we imported the existing one defined on the top level to avoid double maintenance. Sirius allows importing to happen in a circular fashion (e.g when an element importing itself) which can be seen as a form of recursion support.

The implemented tools and their description can be seen in table II. Similar to semantic-graphic mappings, we also used interpreted expression to prevent the tools from being applied in incorrect situations (preconditions). It is also worth to note that Sirius provides a default context dependent property view from which an object in the diagram can be modified. This view is similar to the one in the EMF editor and we have not been able to find out how to modify it.

4.2 User-defined type

In order to support user-defined type, we have extended the meta-model of the language by adding a class called Type. Each program contains zero or many types and each type can contain a number of states. The specification has also been

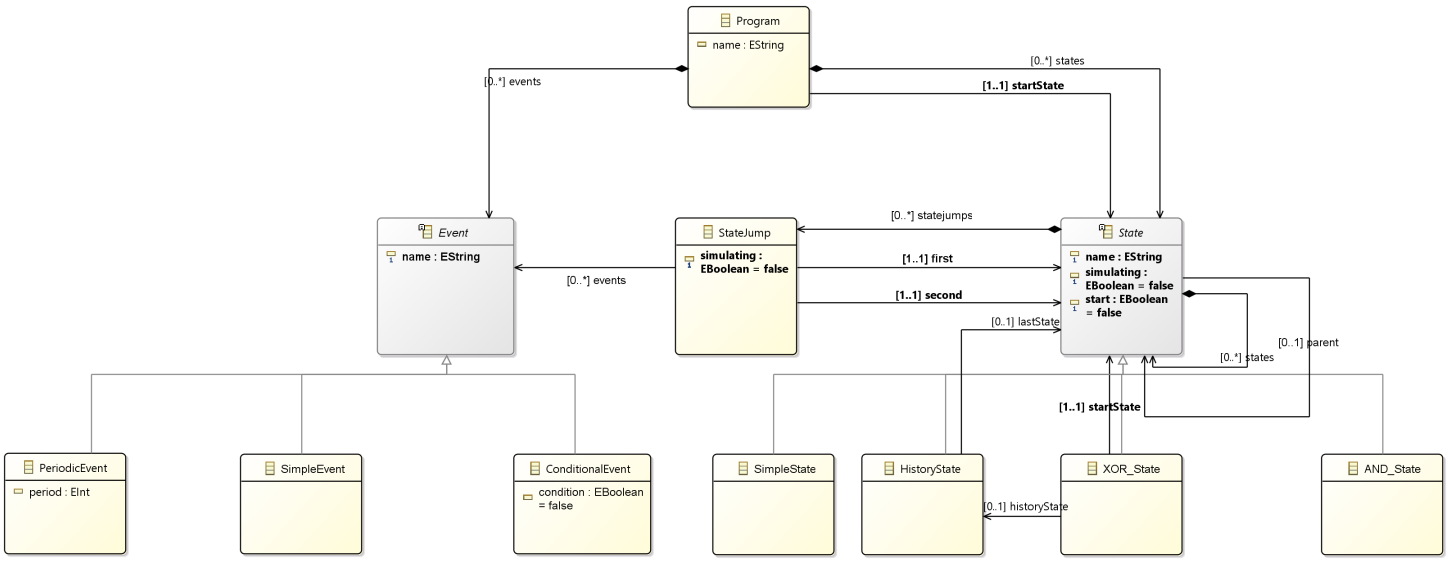

Figure 2. The meta-model of the language (the part about user-defined type has been stripped out to make it clearer).

TABLE I: Implemented mappings and their parameters.

Domain class	Graphical element	Style	Semantic candidates	Import
SimpleState	SimpleContainer	gradient rectangle	container's child states	_
XORState	XORContainer	rectangle	container's child states	SimpleContainer, XORContainer, ANDContainer
ANDState	ANDContainer	rectangle	container's child states	SimpleContainer, XORContainer,
HistoryState	HistoryNode	image	container's reference	_
StateJump	ElementEdge	solid edge, event as label	all state jumps in program	_
Event	EventSubNode	rectangle	all events in program	_

modified to allow users to create a type and edit it in a separate view using the navigate model operation supported by Sirius. The type view is essentially the same as the program view with minor differences. Two more classes were also added as super states, TypeInstance and TypeInstanceShare, to provide two different ways to instantiate a type.

When an TypeInstanceShare object is created, its content is empty, it only contains a reference to the instantiated type. By using semantic candidate expression, we instructed Sirius to search for states inside this referenced type to display. This way it appears as if we have created an instance of the type while in fact, we have only created a different graphical representation for each state in the type. Any modifications done in the editor will affect the same semantic object and be reflected on all graphical representations of that object. To prevent the changes from the program view to affect the semantic object, we manually disabled all operations involving a TypeInstanceShare object. This includes creation, deletion and renaming of states, edges. The default property view mentioned above, however, could not be disabled.

The second way to instantiate a type is by using a TypeInstance object. When created, it will go through all states in the type and create a copy of each of those states. This traversing and copy operation is not supported by Sirius so we wrote our own code. This approach solves the sharing problem above since the created instance contains actual copies of the states in the type. However, it introduces another problem as when modifications are made on the original objects in the type view, they will not be automatically propagated to the other copies. Sirius does not provide a simple way to do this so we implemented our own solution. Currently, our implementation can only propagate changes on the attributes of the objects but not the structure of the tree. Also, the user must actively trigger the propagation as

TABLE II: Implemented tools and their parameters.

Tool	Mapping	Precondition	Operations
CreateSimple	SimpleContainer	container must not be simple state	create instance, set parent and initial name
CreateXOR	XORContainer	container must not be simple state	create instance, set parent and initial name
CreateAND	ANDContainer	container must not be simple state	create instance, set parent and initial name
CreateHistory	HistoryNode	container must be xor state and have no history state	create instance, set paren's reference to this
CreateStateJump	EventSubNode	_	create instance
CreateEvent	EventSubNode	_	create instance
ReconnectEdge	StateJumpEdge	_	change source or target state
EditLabel1	all states and events	_	change name
EditLabel2	StateJumpEdge	event must exist	change event

we did not succeed in automating this process. We were able to detect when a change occurs but unable to propagate the change in the same thread due to a context issue. The concept of context here is native to Sirius and requires domain knowledge to understand, something which we do not have.

5 Evaluation

Table III presents a summary of the target features and their implementation results. The results have been successful for the implementation of statechart features with Sirius being able to solve the issues we mentioned in section 3.1. Our implementation shows that the process to create a statechart editor can be done by simply following the predefined work steps and make use of already supported Sirius features. The complexity of the implementation is low and there is no need for any object oriented programming. In fact, almost all the work was done graphically in the Sirius editor, only one small Java service was written in code. This shows a clear strength of Sirius in facilitating the development of graphical editors.

Another advantage of Sirius is its ability to prevent errors. Syntax errors are also automatically detected. During the implementation, we tested by changing the meta-model so that the target reference of a state jump must be a simple state. The result became that if when creating an edge graphically, any attempts to set a non-simple state as the target will be ignored, the cursor will also be marked with an invalid icon to indicate this. The meta-model was then changed back to the correct way. Certain semantic errors can also be prevented by the use of mapping rules. Operations violating these rules will also be ignored and the users will see an invalid icon on the cursor to indicate that. This can be useful for users who want to quickly design a number of models for comparison. Instead of making mistakes and then having to fix them, Sirius can prevent them from happening in the first place and thus increase productivity.

It is also worth to note that before one can start using Sirius and utilize all of its features, certain prerequisite knowledge is required. In particular, the developer would need to familiarize himself with EMF, the basic concepts of Sirius and one of the query languages. These can be argued as a small investment to pay as they only need to be done once the first time the developer works with Sirius.

The strength of Sirius lies in its automated work done by the underlying process. The developer never needs to explicitly implements operations such as saving, loading data, handling user input or drawing objects on an Eclipse view. This can reduce a lot of development time but also make the developer more reliant on the supported features of Sirius. The user-define type feature, for example, was not supported and our effort to implement it ourselves did not meet with success. Sirius does allow developers to implement their own features or overwrite certain Sirius' functions but a high level of domain knowledge is required [6]. During our project, we encountered the context issue which proved difficult to resolve. As far as we know, Sirius does not provide interfaces to facilitate coding. It does not help either that Sirius is built on top of other frameworks and certain knowledge about the underlying structure of these frameworks is required to ensure successful coding. From the perspective of a developer new to the Eclipse frameworks, it might be difficult when trying to implement a complete new feature in Sirius without first going through a steep learning process.

[6] https://www.eclipse.org/sirius/doc/developer/Sirius%20Developer%20Manual.html

TABLE III: Summary of implementation results.

Feature	Success	Implementation	Sirius support	Required understanding
Creation, modification of states, edges and events	yes	graphically	yes	basic EMF + Sirius concepts
Recursive containement	yes	graphically	yes	_
Semantic constraints	yes	graphically + some code	yes	one of the query languages
User-defined type	no	graphically + code	no	underlying frameworks

6 Related work

The focus of our work has mainly been about finding the strengths and weaknesses of Sirius using the implementation as an proof-of-concept example. A similar work has been done to promote another tool called EuGENia by implementing an editor for a simple filesystem meta-model [1]. EuGENia, however, work at a different level from Sirius. Instead of encapsulating GMF, it facilitates the process of using GMF by producing intermediate models that can then be used as inputs to GMF to generate a graphical editor. This is done by adding annotations to the meta-model of the language and then transform them to the required GMF models.

Graphiti is another framework to produce graphical editors that is somewhat similar to Sirius. Instead of GMF, it encapsulates GEF (Eclipse Graphical editing framework [7]) with the purpose to hide away its complexity. Work has been done to compare Graphiti and Sirius with each other in order to define the pros and cons of each of them [6]. The results showed that Sirius is a better choice for a number of reasons such as more supported features, less error prone, more customizable. The work also pointed out that for difficult tasks, deep knowledge of GEF and GMF is required which concurs with one of the results of our evaluation.

7 Conclusion

In this paper, we have presented our evaluation of Sirius' strengths and weaknesses together with a number of factors that should be considered before working with Sirius. This was done by practically implement an editor for a language heavily inspired by statechart with the addition of user-defined type feature. The results have shown that Sirius has support for all introduced statechart features and an graphical editor can be produced in a short time. The user-defined type feature, however, was not supported and implementing it would require high level of knowledge about Sirius and the underlying frameworks. For a developer unfamiliar with Eclipse frameworks, a small learning process must also be done to familiarize with EMF, basic Sirius concepts and one of the query languages. Possible future works could be to find an approach to make user-defined type feature possible, or to introduce interfaces to make the implementation of new features easier.

[7] https://www.eclipse.org/gef/

Acknowledgments

I would like to thank my supervisor, Alfred Akersson for providing me help and guidelines throughout the project.

References

[1] Dimitrios S Kolovos, Louis M Rose, Richard F Paige, and Fiona AC Polack. 2009. Raising the level of abstraction in the development of GMF-based graphical model editors. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*. IEEE Computer Society, 13–19.

[2] Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. 2014. A tutorial on metamodelling for grammar researchers. *Science of Computer Programming* 96 (2014), 396–416.

[3] Douglas C Schmidt. 2006. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY*- 39, 2 (2006), 25.

[4] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

[5] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. 2014. Sirius: A rapid development of DSM graphical editor. In *Intelligent Engineering Systems (INES), 2014 18th International Conference on*. IEEE, 233–238.

[6] Vladimir Vujović, Mirjana Maksimović, and Branko Perišić. 2014. Comparative analysis of DSM graphical editor frameworks: Graphiti vs. Sirius. In *23nd International Electrotechnical and Computer Science Conference ERK, Portorož, B.* 7–10.