SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 173–192 P.

The Configurator in DSL Tool Building

Arturs Sprogis
Institute of Mathematics and Computer Science, University of Latvia
Raina bulv. 29, Riga, LV-1459, Latvia
Arturs.Sprogis@lumii.lv

This paper describes the Configurator which provides ability to create graphical tools for different domain-specific languages (DSLs) quickly and convieniently. To define different DSLs by the Configurator, a TDA graphical tool building platform and its main component – a *Tool Definition Metamodel*, is used. By using this technology, a specific graphical tool is built as an instance of the *Tool Definition Metamodel,* the main task of the Configurator being creation of new *Tool Definition Metamodel* instances. The basic idea behind the Configurator is to create the instances graphically and add its properties through dialog windows. New universal graphical language and transformations converting universal language elements into *Tool Definition Metamodel* instances was developed as a materialization of this idea.

Keywords: DSL, graphical tool building, metamodel, model transformation.

1 Introduction

Many different formal models are used to describe complex information structure and each model is expressed in a particular language. DSLs [1, 2] are typically the choice because they are specially created to solve problems in one specific domain using well-known concepts for domain experts in contrary to universal languages which solve problems in many domains simultaneously. The main advantage of DSLs is that they allow thinking in a higher level of abstraction; however, their application is restricted by the lack of corresponding tools. Programming every single tool from scratch is time-consuming and takes a lot of effort. Therefore, more advanced methods implementing DSL tools are necessary.

Currently the leading DSL tool definition frameworks are MetaEdit+ [3, 4, 5], Eclipse GMF [6] and Microsoft DSL Tools [7]. DSL tools in MetaEdit+ are defined by GOPPRR [3] (Graph, Object, Property, Port, Relationship, Role) modelling language. All concepts are defined independently from each other but their relationships are specified later when all concepts have been defined. Although in MetaEdit+ new DSL tools are made easily by defining language concepts graphically, the main disadvantage is that it is impossible to change the default behaviour with additional code.

Eclipse GMF and Microsoft DSL Tools use code generation approach. DSL tools are created by compiling generated code and if any changes are necessary, the generated code has to be altered. This requires DSL developers to have advanced knowledge in generated code and in the programming language used in code generation. In addition, one of the main disadvantages of Eclipse GMF is that a user interface is hard to understand, whereas Microsoft DSL Tools is a commercial product and it may only be used together with Microsoft Visual Studio.

In this paper, a new approach of defining DSL tools is presented incorporating both – an easy-to-use graphical interface for typical use cases and a programmatic approach for more specific cases. This idea is implemented in the Configurator, allowing tool builders to define DSL tools with greater flexibility.

Chapter 2 is an overview of the Configurator. An overview of the graphical platform used to build the Configurator is presented in Chapter 3. The implementation of the Configurator and an example illustrating the use of the Configurator is described in Chapter 4.

2 An Overview of the Configurator

Each DSL consists of a number of graphical concepts. One of the basic principles used in the Configurator is to define each concept graphically by defining concept prototypes. Thus, it is necessary for the Configurator's DSL to define other DSLs in the same way OMG defines UML [8] by using Meta-Object Facility (MOF) [9].

DSLs are implemented as graph diagrams; therefore, the Configurator's DSL consists of three main concepts – box, line and property. Box describes nodes, line describes edges and property describes compartments added to node or edge in graph diagrams. Thus, prototypes are expressed in terms of these three concepts. However, each concept has its own behaviour, notation and constraints distinguishing it from other concepts and these features are specified by complex dialog windows. Thus, graphical concepts together with dialog windows make the Configurator's DSL.

The Configurator is implemented using the TDA [10, 11, 12] graphical tool-building platform. The TDA platform consists of engines and metamodels. Every engine has its own corresponding metamodel. For example, *Presentation Engine* uses *Presentation Metamodel* to depict diagrams, whereas *Dialog Engine* uses *Dialog Metamodel* to show dialog windows to end users. Most important of those are the *Universal Interpreter* and the *Tool Definition Metamodel*. The *Universal Interpreter* is a universal transformation interpreting the *Tool Definition Metamodel* to provide working DSL tools. The basic idea of the Configurator is that it defines instances of the *Tool Definition Metamodel* and the *Universal Interpreter* does the rest of the work in cooperation with the *Presentation Engine* and the *Dialog Engine*. The main task in TDA platform which is accomplished by the Configurator is the creation of the the *Tool Definition Metamodel* instances.

Although the Configurator is a tool that defines other DSL tools, the Configurator itself is implemented as a DSL tool in TDA platform using the *bootstrapping* method. The Configurator's *Tool Definition Metamodel* instances are created as a software code and interpreted by the *Universal Interpreter* afterwards. An important thing in the TDA platform is the *Extension Point* mechanism. The *Extension Point* mechanism allows a tool builder to create his own transformations or even programmes and then stores them in the *Tool Definition Metamodel* instances, in this way defining a self-contained tool. The *Extension Point* transformations are called by the *Universal Interpreter* in certain situations. Therefore, very complex tools can be made including the Configurator itself, which maps the Configurator's DSL individuals to the *Tool Definition Metamodel* instances.

3 The TDA Platform

The Configurator is implemented in the TDA platform as a DSL tool; therefore, the TDA platform will be explained in more detail. The TDA platform consists of engines and related metamodels. Every engine accomplishes its functions by interpreting corresponding metamodel. The most important components are the *Universal Interpreter* and the *Tool Definition Metamodel*. The *Tool Definition Metamodel* defines DSLs and the *Universal Interpreter* implements them. The *Universal Interpreter* consists of two kinds of transformations – *Universal Transformations* and *Specific Transformations* executed in specific situations. The main transformation is the *Universal Transformation*, which provides the end users with working DSL tools by interpreting static part of the *Tool Definition Metamodel*.

A command and event mechanism is used to provide a communication among multiple engines. Each event corresponds to the end user's action. As an example – a double click on element corresponds to the event, commands correspond to an order for the engine, for instance, an order for the *Presentation Engine* to redraw all the elements in the diagram. Thus, the communication is organized in such a way that if the end-user does something in the diagram, the *Presentation Engine* receives this action. Then the *Presentation Engine* classifies the action and creates a new event for the transformation. At this moment, the control is assigned to the main transformation that decides which transformation is called to process the event. When the event is processed, the control is passed back to the *Presentation Engine* and a command is created if any assistance by *Presentation Engine* is necessary.

3.1 The Presentation Metamodel

The *Presentation Engine* interprets the *Presentation Metamodel* that results in diagrams seen by end users. Diagrams and their elements are presented as graphs and therefore the *Presentation Metamodel* is very similar to the graph metamodel. In Fig. 1, the kernel of the *Presentation Metamodel* is presented.

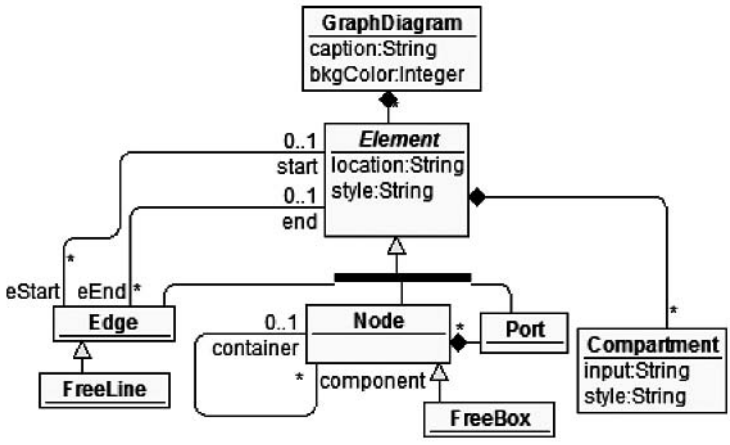

Fig. 1. The kernel of the *Presentation Metamodel*

In the *Presentation Metamodel* every diagram is a graph with name represented by the class *GraphDiagram*. Each diagram contains some elements represented by *Element*. *Element* is an abstract class and therefore real diagram elements are described by its subclasses *Node*, *Edge*, *Port*, *FreeLine* and *FreeBox*. These elements have two attributes – *location* and *style*. Attribute *style* describes how an individual element is visualized and *location* contains information about element position in the diagram and its size. Each element may have a number of attributes, which display the information entered by an end user and are represented by class *Compartment* containing attribute *input* to store entered value and *style* to represent the value.

However, every element must have its own default style and therefore the *Presentation Metamodel* is symmetrically extended with classes *GraphDiagramStyle*, *ElemStyle*, *EdgeStyle*, *NodeStyle* and *CompartStyle*. The extended *Presentation Metamodel* is presented in Fig. 2. Default styles are used when a new element is created, but they can be changed by an end user as well. Thus, when each element is created the default style value is stored in *style*, but if the individual style is changed afterwards, the value stored in *style* is overwritten.

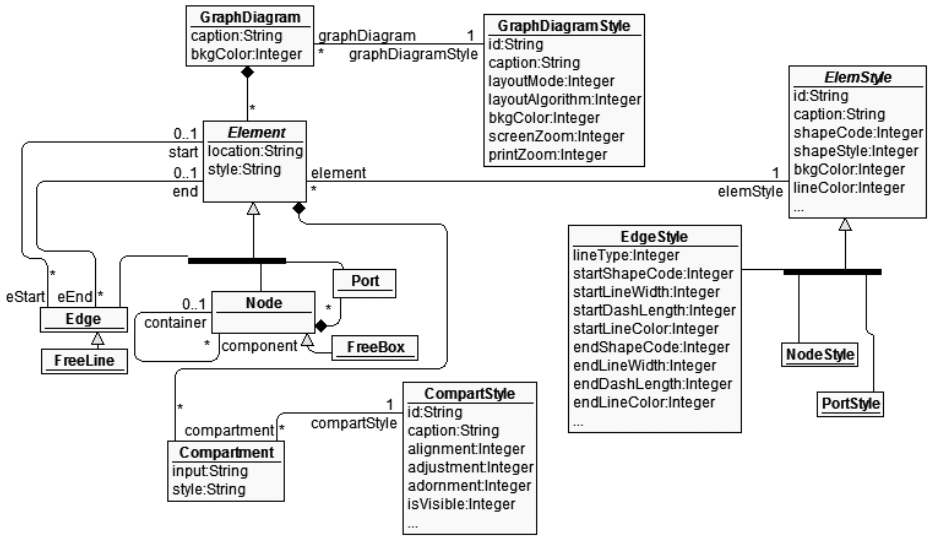

Fig. 2. The *Presentation Metamodel*

The next step is to extend the metamodel to support additional services. Classes *Palette*, *PaletteElement*, *PaletteNode*, *PaletteFreeBox*, *PaletteFreeLine*, *PalettePort* and *PaletteEdge* describe controls allowing to create new elements in diagrams. Classes *Toolbar* and *ToolbarElement* add a toolbar component and classes *PopUpDiagram* and *PopUpElement* add context menus.

To ensure the previously described event and command mechanism, *Event* and *Command* classes must be added to the metamodel as well. Each particular event and command is represented as a subclass of *Event* or *Command* (they are not presented in this paper). Class *Event* has exactly one instance at any given time, whereas

several *Command* instances can be linked by *previous-next* links simultaneously. Two additional classes *CurrentDgrPointer* and *Collection* indicate the state of the tool. *CurrentDgrPointer* indicates the active diagram; *Collection* indicates elements selected by an end user. In Fig. 3, a simplified metamodel is presented.

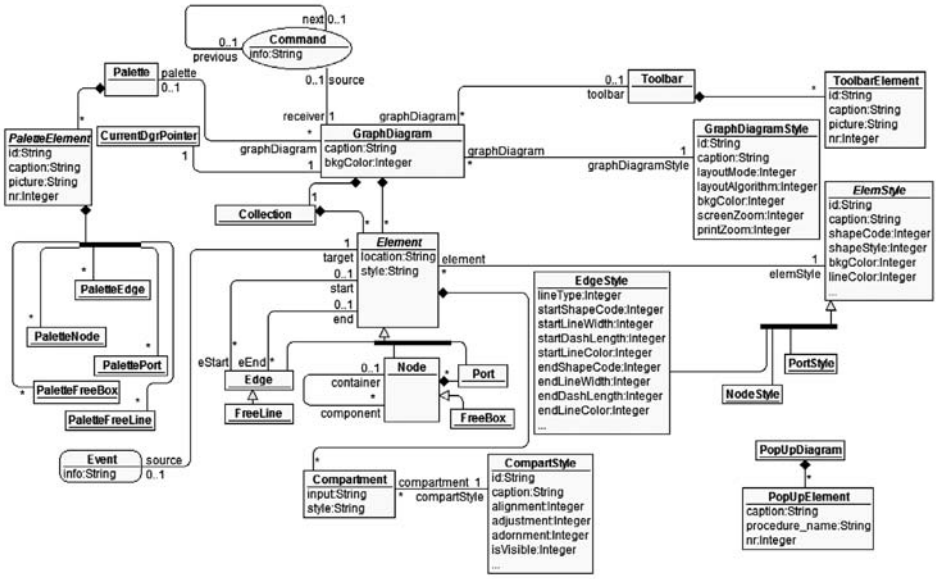

Fig. 3. A simplified *Presentation Metamodel*

3.2 The Structure of the Tool Definition Metamodel

The *Tool Definition Metamodel* is created as an extension of the *Presentation Metamodel* and its basic idea is to describe DSL's graphical elements, their behaviour, constraints, and the necessary information to automatically generate dialog windows. The main classes are *GraphDiagramType*, *ElemType*, *NodeType*, *EdgeType*, *PortType*, *CompartType* that are symmetric to the *Presentation Metamodel*. These classes store metainformation about each individual tool and are interpreted by the *Universal Interpreter* which processes all end user's actions in cooperation with other engines, for example, *Presentation Engine* and *Dialog Engine*. To create more powerful tools, types are complemented with a special kind of attributes starting with prefix "proc" in order to implement the *Extension Point* mechanism which allows adding specific transformations by tool developers to specify element behaviour in certain situations, for example, to fill dynamically drop-down menus.

However, types not only describe element behaviour, they describe the existing constraints as well. A composition relationship between *GraphDiagramType* and *ElemType* is a constraint, which determines a set of elements contained in the diagram, whereas composition between *ElemType* and *CompartType* defines attributes linked to the element. A class *Pair* determines which types of elements may be connected. Thus, associations *pair-start* and *pair-end* define which type of elements can serve as start

and end elements. Another constraint is the association *containerType-componentType* defining which type of *Boxes* may contain other *Boxes*. At the same time association *nodeType-portType* determines the type of *Box* that is enchained to a *Port*.

There are situations when some attribute values have to be entered independently from other attribute values and they have to be concatenated when shown in diagrams. This is implemented using composite attributes by adding associations *subCompartment-parentCompartment* and *subCompartType-parentCompartType*. A hierarchy of attributes is made in a way that only first level attributes are displayed to end users and attributes above hold their temporary values when processed by the *Universal Interpreter*. For instance, in a UML class diagram, *Object* name (full name) is made by concatenating three values – "individual_name", ":" and "class_name". There is one first level attribute showing the result of concatenation, for example, "John:Person", and three second level attributes holding values for each attribute – "John", ":" and "Person".

Classes *PropertyDiagram*, *PropertyTab*, *PropertyRow* represent components used by the *Universal Interpreter* to generate complete dialog windows automatically. Class *PropertyRow* has an attribute *rowType* determining the type of control used to enter attribute values. For example, the value "InputRow" specifies the text box control. To allow calling multiple dialog windows in several levels an association *calledPropertyRow-calledPropertyDiagram* is introduced. This feature is used if the *rowType* value, for example, "InputRow+Button" is chosen. As a result, a text box and a button are added, and by pressing the button, another dialog window is opened.

Sometimes it is necessary to dynamically change element and attribute styles. Element and attribute type has at least one corresponding style that is joined by the associations *elemType-elemStyle* or *compartType-compartStyle*. If there is more than one style, they must be switched in certain situations. It is implemented by adding a class *ChoiceItem* and associations *choiceItem-ElemStyleByChoiceItem* and *choiceItem-compartStyleByChoiceItem*. Each *ChoiceItem* instance holds a certain value and if this value is entered, the linked style is added. For instance, in UML class diagram, *Class* name's text has to be shown in normal or in italic based on whether the class is abstract or not. Thus, there must be a check box, which is checked if the class is abstract and not checked otherwise. According to the metamodel, there are two *ChoiceItem* instances holding values "True" and "False". A normal style is added to "False" value and an italic style is added to "True" value. Hence, when an end user checks or unchecks the check box, the *Class* name's style is changed accordingly. In Fig. 4, the complete *Tool Definition Metamodel* is present.

4 The Configurator

The implementation of the Configurator is predominantly based on the *Universal Interpreter* and the *Tool Definition Metamodel*. All the tools store two different kinds of instances in the *Tool Definition Metamodel*. One kind of instances defines the tool and is static as far as instances never change. The *Universal Interpreter* uses them to process end user actions. The second kind of instances is created dynamically and those correspond to the elements end users work with. End users may add, update, and delete them. The main problem is how to define static instances because they are individual for every single tool, whereas dynamic instances are processed equally in all tools.

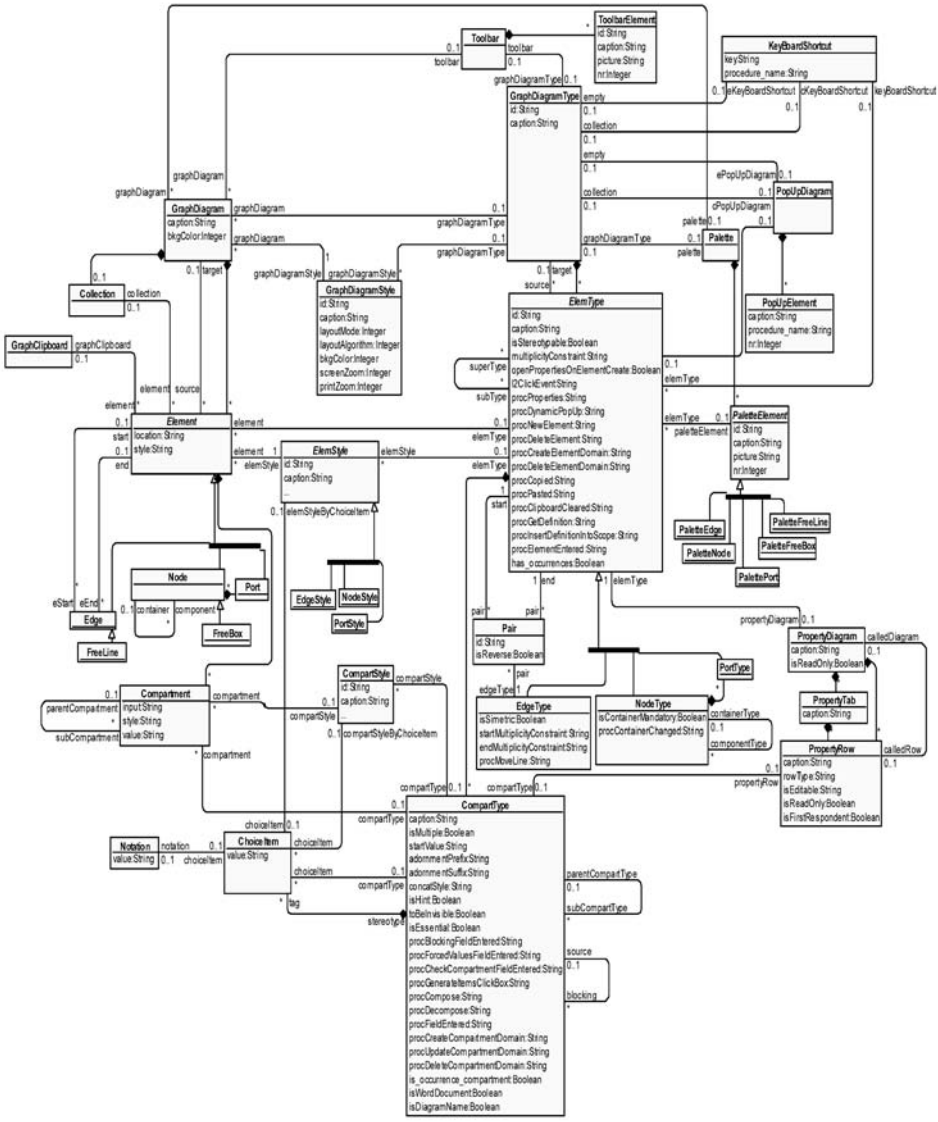

Fig. 4. The *Tool Definition Metamodel*

If static instances are “somehow” created, the working tool is obtained immediately. The naïve approach would be to create them manually but it causes several problems. Firstly, a number of instances soon grow very large. Secondly, there are many links and attribute values to be set and those can easily cause an error; therefore, this approach is significantly error-prone. Thirdly, a tool developer must know the *Tool Definition Metamodel* and attribute values expected by the *Presentation Engine*.

The Configurator is built to automate the creation of the *Tool Definition Metamodel* instances using the *bootstrapping* method. The basic idea is to implement the Configurator

as a DSL tool using the *Tool Definition Metamodel* and the *Universal Interpreter*. There are static instances defining the Configurator like any other tool in TDA but the new approach is to define static instances of DSL tools by dynamic instances using mapping from dynamic instances to static. The mapping is created using the *Extension Point* mechanism. The *Universal Interpreter* calls specific transformations in certain situations and this process consists of two steps. In the first step, the *Universal Interpreter* creates dynamic instances, which are seen by tool developers. Then, in the second step, a specific transformation which creates, deletes or updates static instances is called. The structure of DSL tool definition in the Configurator is presented in Fig. 5.

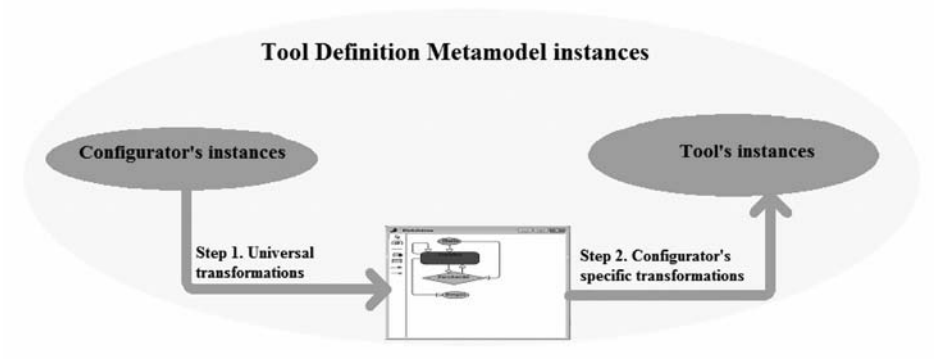

Fig. 5. The structure of DSL tool definition in the Configurator

The scaffolding must be added to the *Tool Definition Metamodel* to implement the Configurator according to the schema. The main purpose of scaffolding is to map dynamic instances to the static instances. There are three associations *presentation-target_type* added to the metamodel to identify an element type being defined by *GraphDiagram*, *Element* or *Compartment*. In Fig. 6, an extended *Tool Definition Metamodel* fragment is presented.

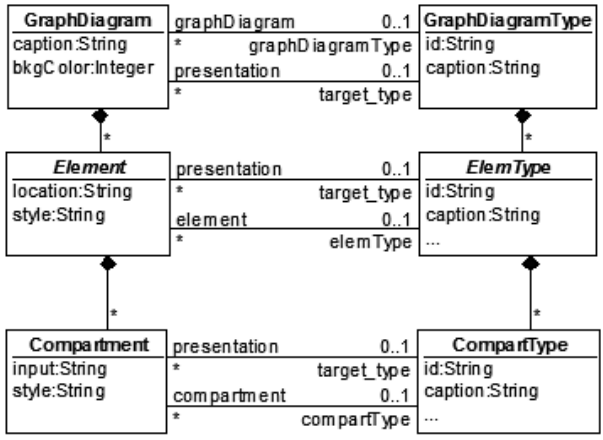

Fig. 6. An extended *Tool Definition Metamodel*

4.1 Implementing the Configurator

According to the TDA platform, static and dynamic instances are stored for each DSL tool. As far as the Configurator is implemented as a DSL tool, there are static instances defining the Configurator and those are presented in this sub-section.

The Configurator's DSL consists of two diagram types. One type of diagrams defines prototypes for diagram seeds, and lines illustrating dependencies between them. The second type of diagrams defines element prototypes. The first type of diagram is named *Specification Diagram* with three types of elements possible – *Seeds*, *Lines* and *Specializations*. *Seed* is an element which defines seed prototype; *Line* is an element which defines dependency prototype, and *Specialization* is a line used to indicate that a sub-element inherits incoming and outgoing lines and constraints from a super-element. *Tool Definition Metamodel* instances presented in Fig. 7 specify element types for *Seeds*, *Lines*, and *Specializations*. There are additional instances specifying context menus and corresponding palette buttons.

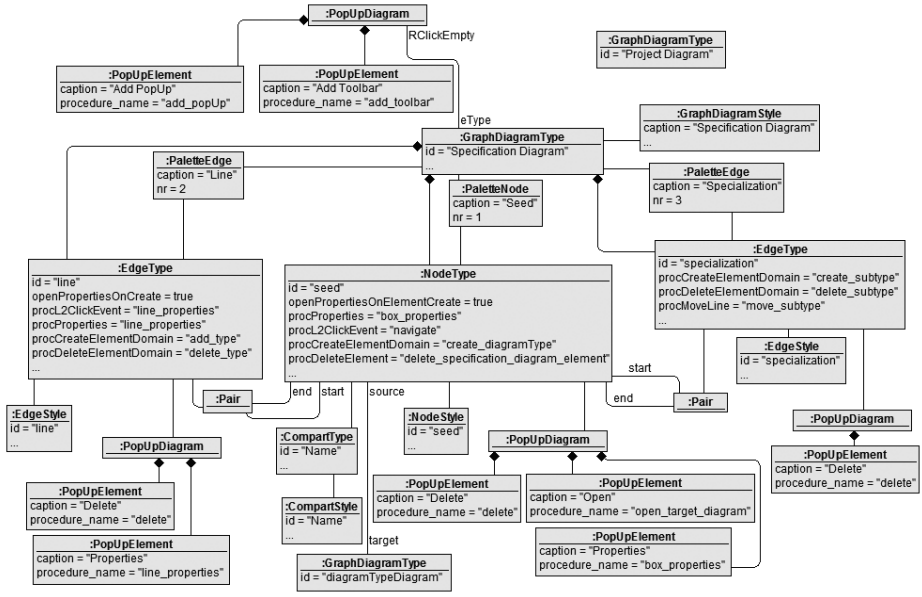

Fig. 7. The specification diagram defined in the *Tool Definition Metamodel*

The second diagram type defines element prototypes. There are six different element types available – *Box*, *Line*, *FreeBox*, *FreeLine*, *Port* and *Specialization*. *Box* and *Line* elements allow defining prototypes for boxes and lines; *FreeBox* allows defining boxes always remaining in background; *FreeLine* allows defining lines having no start and no end elements; *Port* allows defining small boxes which are always attached to some *Box*; *Specialization* is used for the same purpose as in *Specification Diagram*. In Fig. 8, a definition of prototype diagram and its elements with relevant context menus in the *Tool Definition Metamodel* is presented. *Line* and *Specialization* elements are allowed to connect all the elements except *Specialization.* For example, *Box* and *Box* elements are

allowed being connected by *Line* or *Specialization* element, but *Box* and *Specialization* elements are not allowed being connected by either *Line* or *Specialization*. Thus, all the possible pairs must be present in the metamodel to indicate which elements may be connected and which may not. there is a special *NodeType* instance with *id* value "superType" added as a super-type for all the elements, except *Specialization,* and two *Pair* instances, which connect "superType" instance and *Specialization*'s type instance, "superType" instance and *Line*'s type instance. Introduction of "superType" is needed to save the effort of making all the necessary pairs because the incoming and outgoing lines are inherited from the super-type.

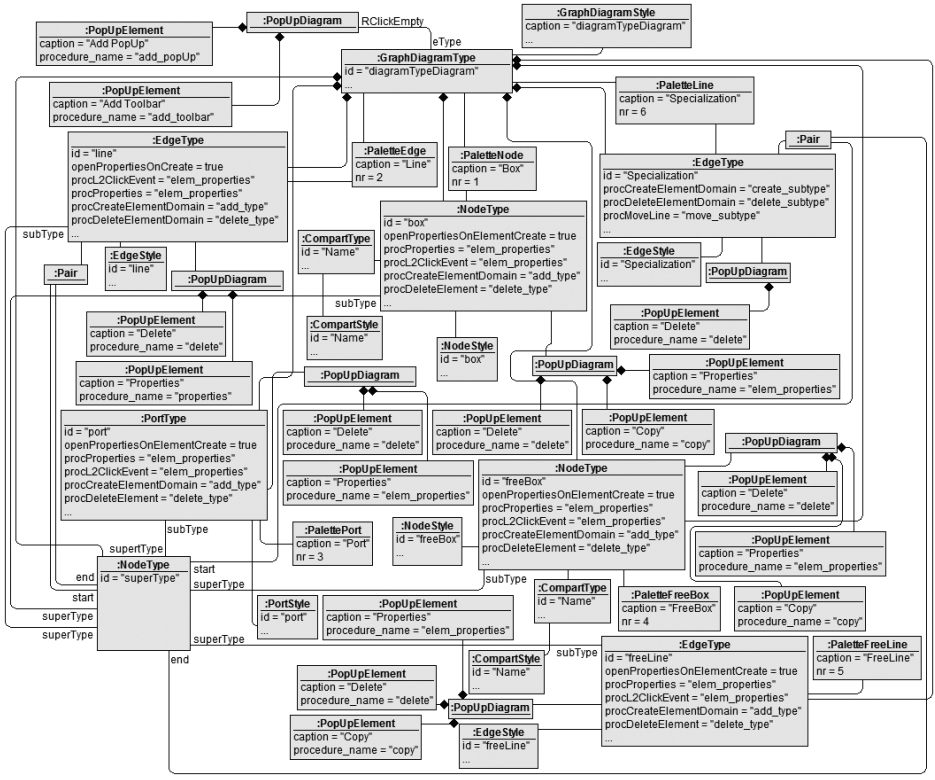

Fig. 8. Prototype diagram's definition in the *Tool Definition Metamodel*

4.2 The Configurator in Use

In Fig. 7 and 8, the *Tool Definition Metamodel* instances defining the Configurator are presented. If the tool is specified by static instances, the *Universal Interpreter* creates and processes dynamic instances. To illustrate how static instances are used in tool building, a simplified *Flowchart* editor is built consisting of the following symbols – *Start*, *End*, *Action*, *Branching*, *Simple Flow* and *Branching Flow*. In addition, *Action* symbol has a property *Expression*; *Branching* symbol has a property *Condition* and *Branching Flow* has a property *Choice*.

A new diagram type is defined in *Specification Diagram* by creating a new seed. A new *Node* instance is created and linked to *GraphDiagram*, *NodeType* and *NodeStyle* instances by the *Universal Interpreter*. The result is presented in Fig. 9.

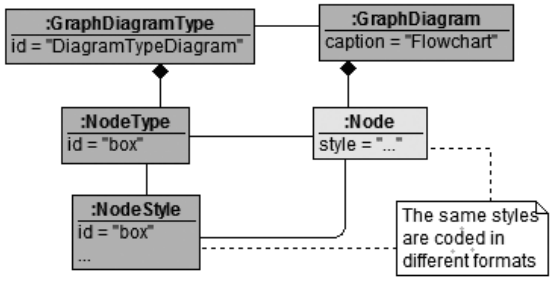

Fig. 9. Node instance created by the *Universal Interpreter*

When the *Universal Interpreter's* work is done, a specific *Extension Point* transformation from attribute *procCreateDomain* is called to create corresponding static instances for *Flowchart* editor and dynamic instances for the Configurator. It is achieved in two steps. In the first step, a new *GraphDiagram* instance is created for *Flowchart* element prototype definition. In the second step, new *NodeType*, *NodeStyle*, *GraphDiagramType* and *GraphDiagramStyle* instances are created. *NodeType* instance defines a seed type in *Project Diagram*, *NodeStyle* instance defines a seed style. *GraphDiagramType* instance defines a diagram type, which corresponds to *Flowchart* diagram, and *GraphDiagramStyle* instance defines its style. *Node* and *NodeType*, *GraphDiagram* and *GraphDiagramType* instances are linked by *presentation-target_type* to indicate which diagram type is defined; *Node* and *GraphDiagram* instances are linked by *source-target* to connect seed prototype and diagram, in which prototypes are defined. However, there is a necessity for additional instances of type *Palette* and *PaletteNode* to create a palette and a palette button in *Project Diagram*.

When a seed element is created, a property to store diagram's name must be added. This is achieved by adding *Compartment*, *CompartStyle*, *CompartType* to store and represent the end user values, but *PropertyDiagram* and *PropertyRow* instances are added to enter the value from dialog window. In Fig. 11, the instance diagram defining *Flowchart* seed is presented.

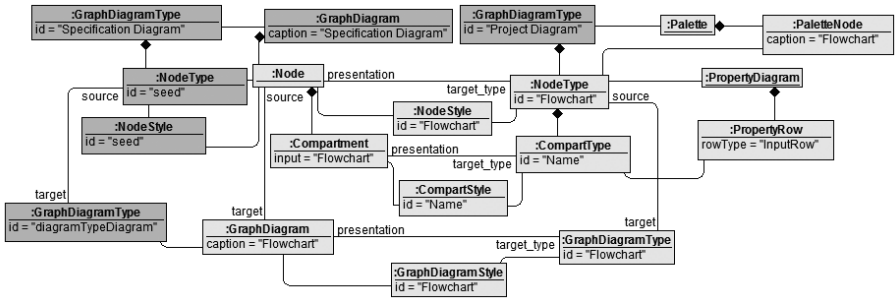

Fig. 10. Instance diagram after creating *Flowchart's* seed element

When a *Flowchart*'s seed element is defined, element prototypes must be defined. Element prototypes of *Box* type are added in the same way as seed prototype; therefore, only line definition is explained in more detail. Assuming that *Start* and *Action* elements are defined in the same manner as *Seed*, *Simple Flow* prototype is added in two steps. In the first step, *Edge* instance is created that links two *Node* instances which represent *Start* and *Action*. In the second step, *Flowchart*'s editor static instances *EdgeType*, *EdgeStyle*, *Pair* and *PaletteLine* are created. *EdgeType* and *Pair* instances define a *Simple Flow* element, which allows to connect *Start* and *Action* elements; *EdgeStyle* defines *Simple Flow*'s style and *PaletteLine* defines a palette button to create *Simple Flow* element. In Fig. 11, the instance diagram defining *Flow* is presented.

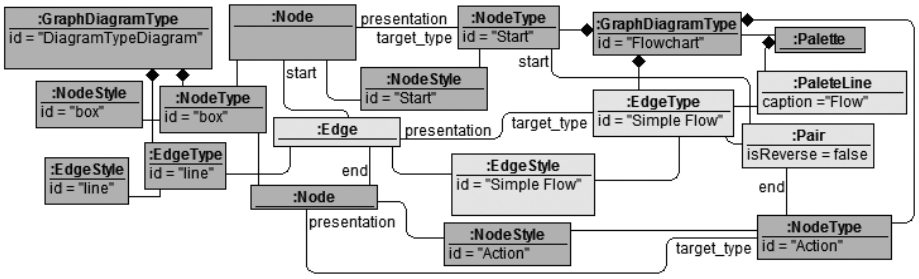

Fig. 11. Flow's definition

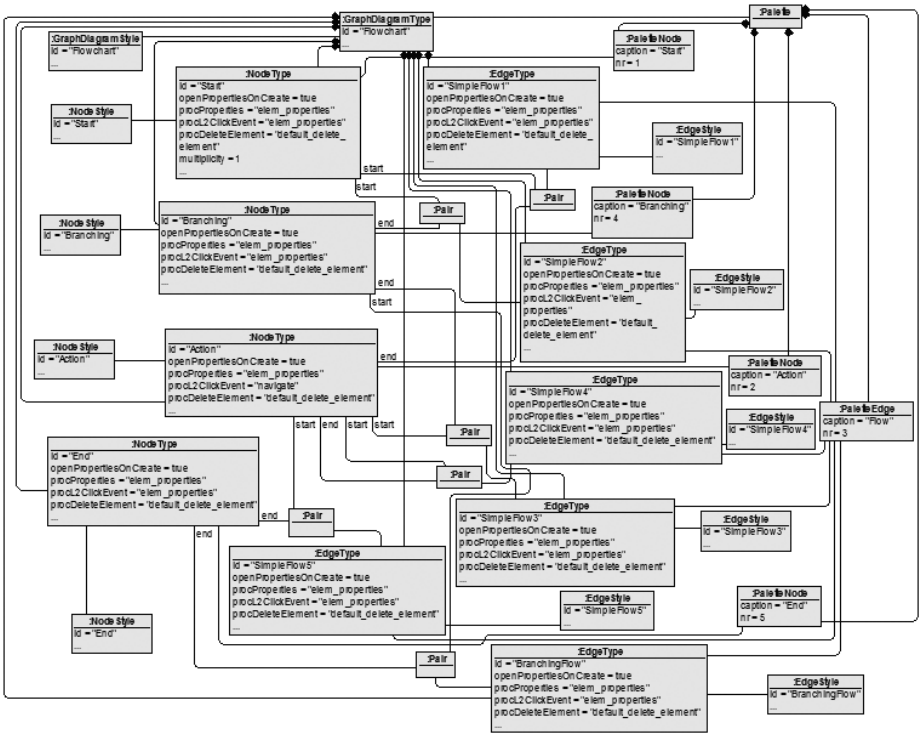

Fig. 12. The *Tool Definition Metamodel* instances

The entire *Tool Definition Metamodel* instance of the *Flowchart* editor presented in Fig. 12 can be obtained using the method described above.

4.3 Defining *Flowchart* Editor Using the Configurator

After discussing the Configurator's implementation and the way it creates the instances of the *Tool Definition Metamodel* above, we shall demonstrate the use of the Configurator from the tool builder's point of view by implementing the *Flowchart* editor.

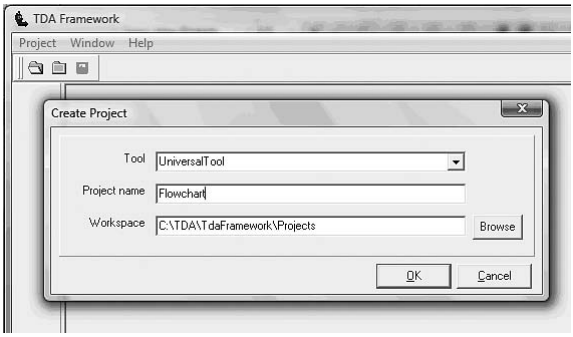

Fig. 13. A new tool definition window

When a new DSL tool is defined, a window to specify project details (Project→New project) is opened. It is presented in Fig. 13. A tool builder has to select value *UniversalTool* in the field *Tool*. Then he has to specify the name of the new tool in the field *Project name* and the project location in the field *Workspace*. When DSL developer presses *OK* button, *Project diagram* is opened. In general, *Project diagram* contains all the available diagram type seeds (elements that allow making diagrams), but that is not the case in the Configurator. When using the Configurator, *Project diagram* contains no diagram type at all, because the Configurator will define it later. New diagram types are defined in *Specification diagram*. Tool builder can navigate to *Specification diagram* by right-clicking and choosing *Specification diagram* from the context menu. A sample project diagram and the context menu are presented in Fig. 14.

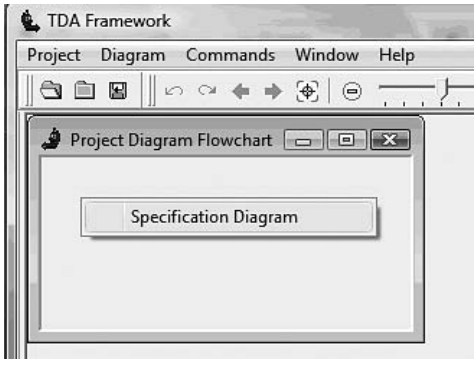

Fig. 14. A sample project diagram

In *Specification diagram*, new diagram types are defined using *Seed* element. A *Flowchart* diagram *Seed* has to be created by pressing *Seed* button in the palette. When *Flowchart* diagram type is defined, a *Flowchart* diagram is opened by double-clicking on the diagram *Seed*. In Fig. 15, *Flowchart* diagram *Seed* and diagram for element definitions is presented.

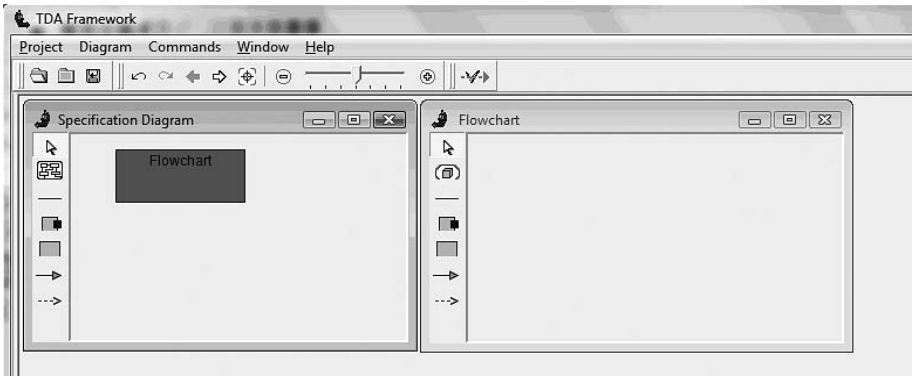

Fig. 15. Creating Flowchart Seed

When *Flowchart* definition diagram is opened, *Flowchart* elements can be defined by creating their prototypes. For instance, *Action* is defined by choosing *Box* button in the palette. A *Box* dialog window is displayed afterwards and tool builder is prompted to enter element values. In the field *Name* a value "Action" has to be entered which automatically renames a palette element name in the field *Palette Element Name*. In the field *Palette Element Nr*, a number for palette element in the palette has to be entered and in this case, the number is "2". In the field, *Icon Path* an icon's name for a palette element has to be specified. Context menu elements for *Action* have to be defined as well. Those are specified in the table *PopUpDiagram* and in this case, context menu items are default with corresponding default transformations added from the transformation library – *Delete, Cut, Copy* and *Properties*. It is possible to specify navigation target diagram in the field *Navigate To Diagram* by double-clicking on the element. If nothing is specified, no navigation is possible. However, in this particular case, a value "Flowchart" is specified meaning that double-clicking navigates the end user to one of the *Flowchart* diagrams. In Fig. 16, a window to enter *Action* values is presented.

When all the element values are entered, element properties have to be specified. It is done by pressing the button *AddChild*. In Fig. 17, a property dialog window is presented. Property value has to be entered in the field *Name*, and in this particular case, the value is "Expression". The visual control used to enter the property value has to be specified in the field *Row Type*, and in this case, the value is "InputRow", meaning the control to enter property values is a textbox.

When all the values are entered, element style has to be specified by pressing the button *Style*. In Fig. 18, the dialog window to enter element style is presented. In this dialog window, a tool builder has to specify values as box type, which may take one of the following values – rectangle, ellipse, round rectangle, etc; a default size, a colour, a border's colour, a border's width and some other visual features.

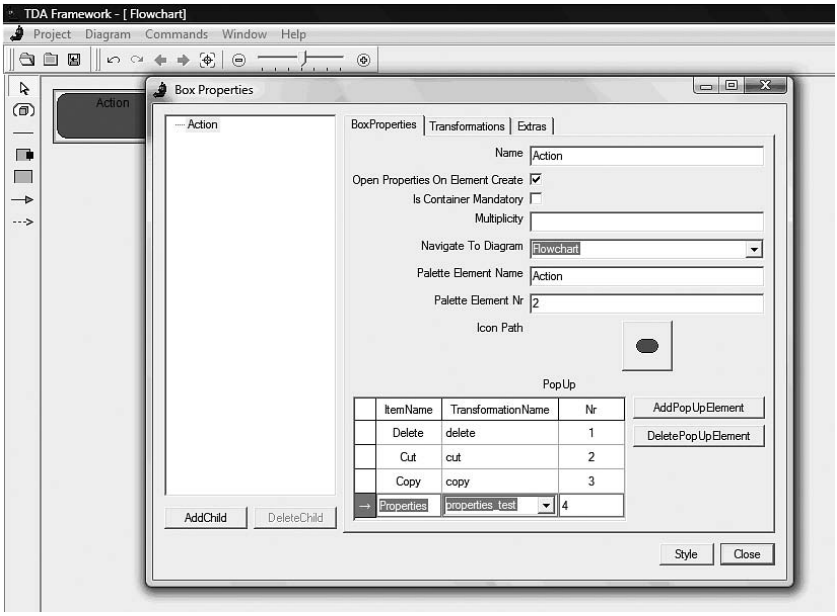

Fig. 16. Definition of an *Action* element

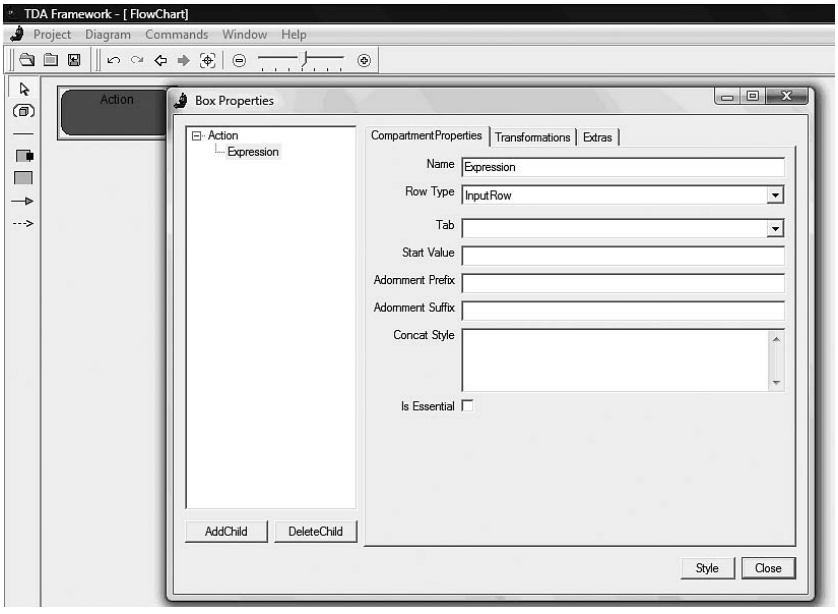

Fig. 17. Definition of the property "Expression"

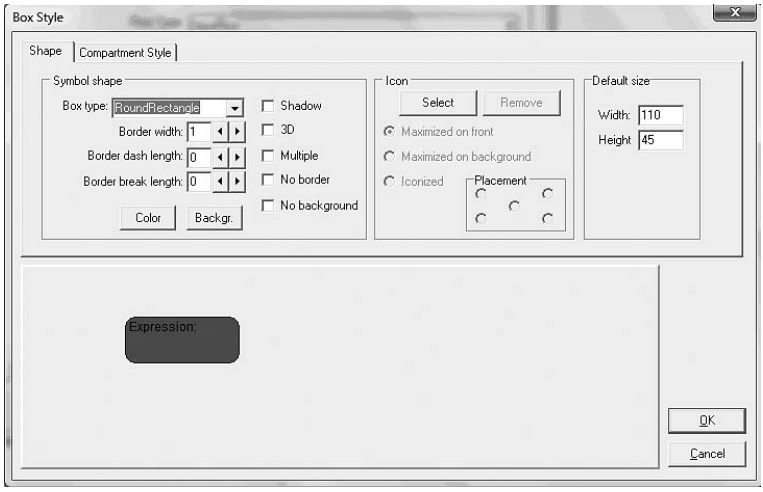

Fig. 18. A style definition window for *Box* prototype

In Fig. 19, the dialog window to enter properties style values is presented. The tool builder has to specify property values like text alignment, adjustment, font style, etc in this window.

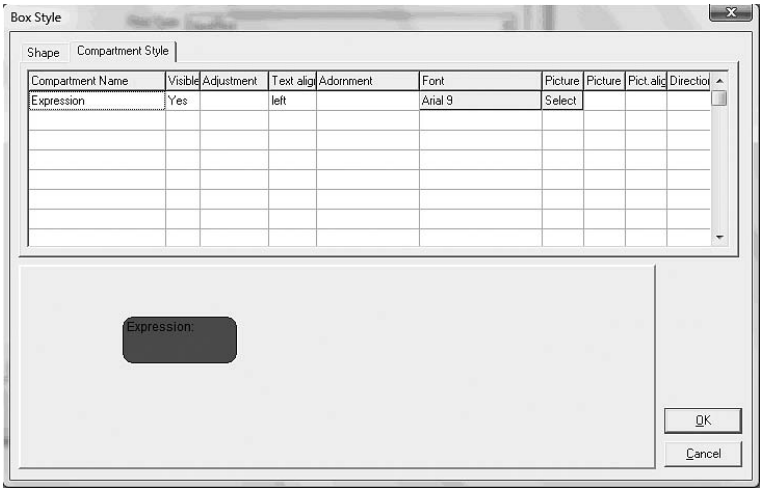

Fig. 19. A style definition window for properties

This is how concepts of *Box* type are defined in the Configurator. Other *Flowchart* concepts of *Box* type like *Start*, *End* and *Branching* symbols are defined using similar approach. In Fig. 20, all *Flowchart* prototypes of *Box* type are presented.

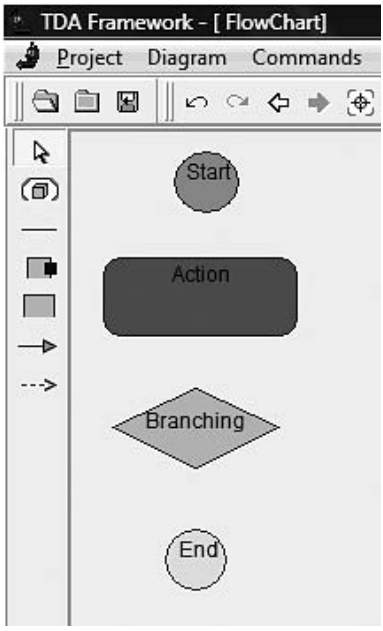

Fig. 20. Box prototypes for the *Flowchart* editor

The next step is to define prototypes of *Line* type. They are *Simple Flow* and *Branching Flow*. In the context of this example, the assumption is made that a *Simple Flow* is an element which may join *Start* and *Action*, *Start* and *Branching*, *Action* and *Action*, *Action* and *Branching*, *Action* and *End* symbols, whereas *Branching Flow* may join only *Branching* and *Action* symbols.

When a *Line* prototype for a *Simple Flow* is defined, all the mentioned cases have to be considered. One *Line* can join only two elements and wherefore there is a necessity for many new prototypes to consider all the *Simple Flow* cases. However, the tool user does not have to know all the technical constraints; therefore, an illusion must be created that there is only one *Simple Flow* element in the diagram. This is achieved by having a common palette button for all the different prototypes in diagram's palette and all the prototypes are made equal by their style and behaviour. In Fig. 21, an example is demonstrated of how prototype is defined for one of *Simple Flow* elements. The definition of elements with a *Line* type is very similar to the *Box* type definition; hence, in the field *Palette Element Name* a drop down menu is used to offer all the palette button names. If the name entered matches any item from the drop down menu, a new palette button is not created and prototype being defined is linked to an existing palette button. Otherwise, a new palette button is created. In the *Flowchart* case, all the *Simple Flow* prototypes are linked to the palette button *Flow*.

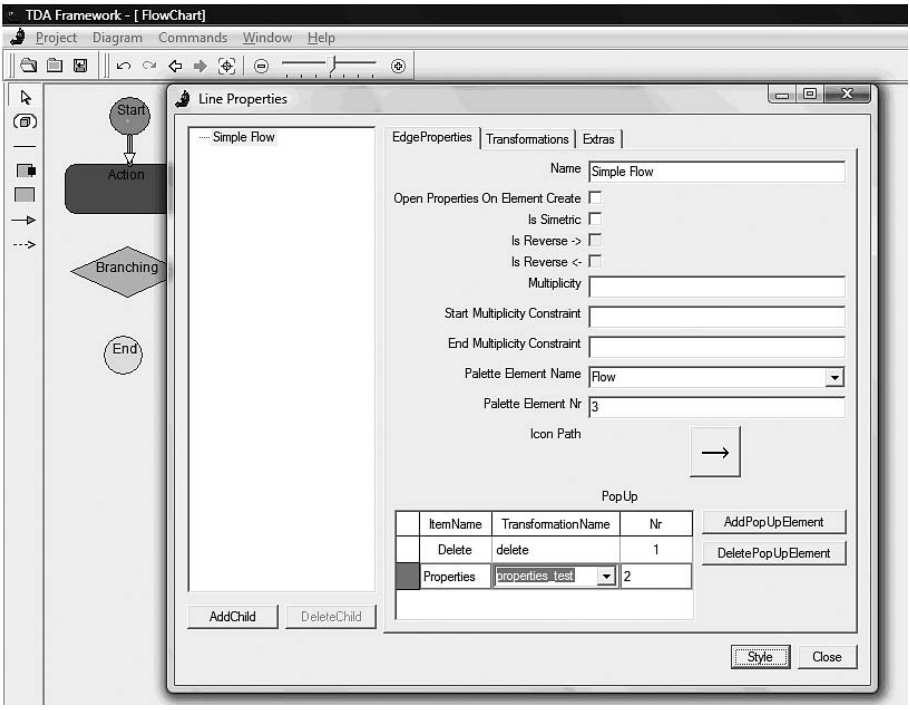

Fig. 21. Definition of a *Simple Flow* element

Branching Flow is defined almost in the same way as *Simple Flow*, except *Branching Flow* has a property *Choice* to enter values like – *Yes*, *No*, *True*, *False*, etc. In addition, *Branching Flow* is linked to the palette button *Flow*. Thus, all the *Lines* are created by only one palette button *Flow* and the decision which *Line* to choose in particular situation is made by the *Universal Interpreter*. In Fig. 22, a final definition of a *Flowchart* editor is presented.

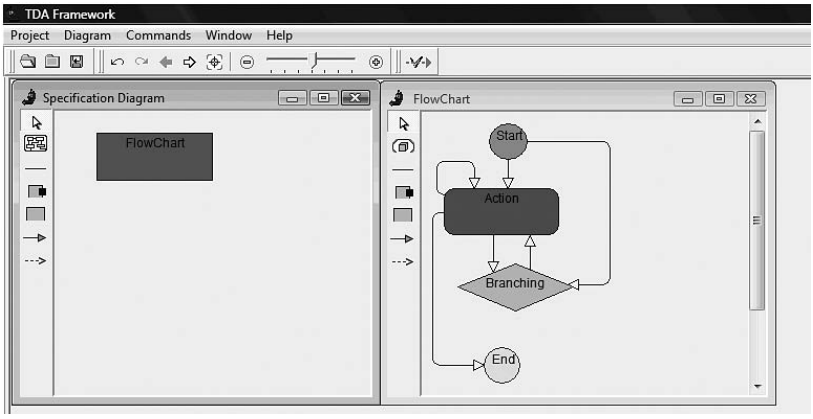

Fig. 22. Definition of a *Flowchart* editor

Yet, in Fig. 23, a working *Flowchart* editor is presented.

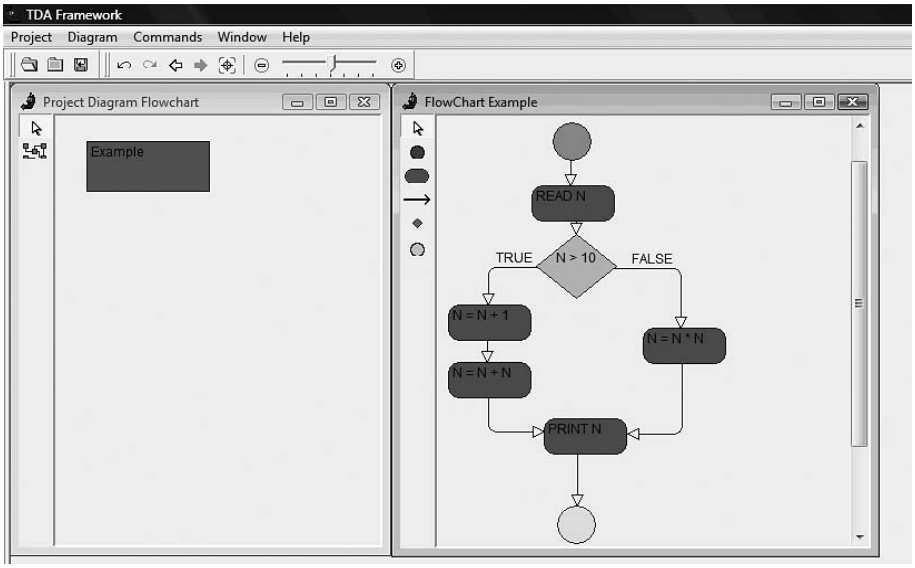

Fig. 23. A *Flowchart* editor in use

Conclusion and the Future Work

Currently the Configurator has enough functionality to implement many different DSL tools. For example, as far as the Configurator is a DSL tool, it is powerful enough to implement even such a complex tool as the Configurator itself. Real business tools are also implemented for Investment and Development Agency of Latvia and the State Social Insurance Agency. Although these tools where successfully implemented, several problems require further research – there is no multi-user mode to support multiple DSL tool developers, the graphical language is insufficiently self-descriptive and user-friendly, and incorporation of other software like MS Word, Database editors, etc in implemented tools is not completely satisfactory.

References

1. UML vs. Domain-Specific Languages. Available: http://www.methodsandtools.com/archive/archive.php?id=71.
2. Domain-Specific Language. Available: http://www.program-transformation.org/Transform/DomainSpecificLanguages.
3. MetaEdit+ Workbench User's Guide, Version 4.5. Available: http://www.metacase.com/support/45/manuals/mwb/Mw.html.
4. S. Kelly, J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008, p. 448.
5. Domain-Specific Modeling with MetaEdit+. Available: http://www.metacase.com/.
6. Graphical Modeling Framework (GMF, Eclipse Modeling subproject). Available: http://www.eclipse.org/gmf/.

7. S. Cook, G. Jones, S. Kent, A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools.* Addison-Wesley, 2007.
8. OMG modeling specification, UML 2.0 Superstructure and Infrastracture. Available: http://www.omg.org/docs/formal/07-02-05.pdf.
9. Meta-Object Facility (MOF). Available: http://www.omg.org/mof/.
10. J. Bārzdiņš, E. Rencis, S. Kozlovičs. *The Transformation-Driven Architecture.* The 8th OOPSLA Workshop on Domain-Specific Modeling, October 19–20, 2008, Nashville, TN.
11. J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins. A Graph Diagram Engine for the Transformation-Driven Architecture. *Proc. of the Workshop on Model-Driven Development of Advanced User Interfaces 2009*. Florida, USA: IUI, 2009.
12. J. Bārzdiņš, A. Zariņš, K. Čerāns, A. Kalniņš, E. Rencis, L. Lāce, R. Liepiņš, A. Sproģis. *GrTP: Transformation-Based Graphical Tool Building Platform.* The 10th International Conference on Model-Driven Engineering Languages and Systems, Models 2007, September 30–October 5, 2007, Nashville, TN.