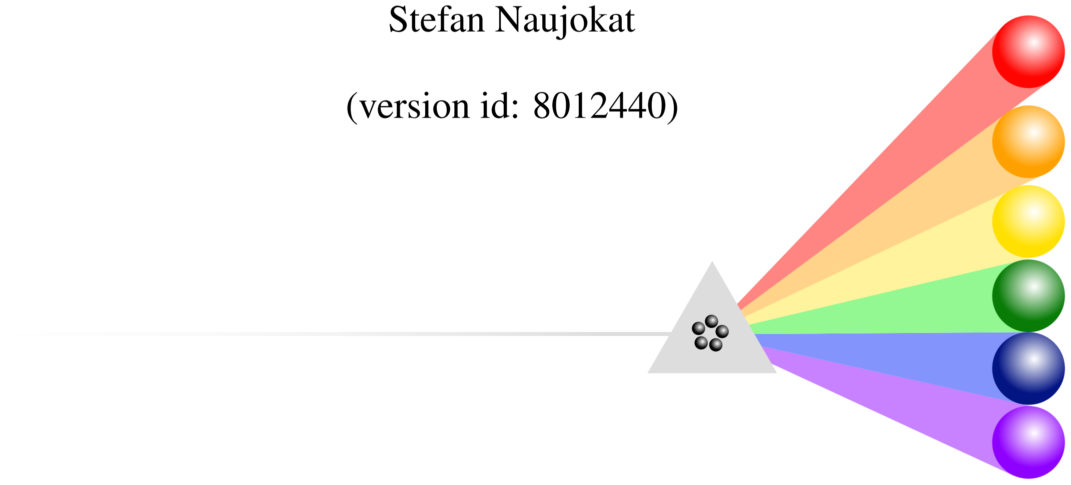
Cinco User's Manual

Stefan Naujokat

(version id: 8012440)



1 Basics

1.1 About This Document

This is the technical manual for the "Cinco SCCE Meta Tooling Suite" (in short just Cinco). It is intended to provide a basic usage introduction as well as (coming soon^{TM}) reference for all features and keywords. It only implicitely covers the conceptual ideas behind Cinco. For a more in-depth and scientific view on those, please refer to the following publications:

- B. Steffen & S. Naujokat: Archimedean Points: The Essence for Mastering Change [SN16]
- S. Naujokat et al.: Meta-Level Reuse for Mastering Domain Specialization [NNMS16]
- D. Kopetzki: Model-based generation of graphical editors on the basis of abstract meta-model specifications [Kop14]

For concrete applications, i.e, tools developed with Cinco, consider the following publications:

- S. Boßelmann et al.: DIME: A Programming-Less Modeling Environment for Web Applications [BFK+16]
- N. Wortmann et al.: A Fully Model-Based Approach to Software Development for Industrial Centrifuges [WMN16]
- L.-M. Traonouez et al.: A Model-Based Framework for the Specification and Analysis of Hierarchical Scheduling Systems [TLK+16]
- S. Naujokat et al.: Domain-Specific Code Generator Modeling: A Case Study for Multi-Faceted Concurrent Systems [NTI+14]

This document will constantly be revised and extendend. The latest version will always be available for download at `http://cinco.scce.info/resources/`.

covered version Cinco v. 0.7

git commit id 8012440

date of build October 18, 2016

1.2 Introduction

The "Cinco SCCE Meta Tooling Suite" is an integrated devlopment environment (IDE) for the quick and easy development of domain-specific graphical modeling tools. Cinco makes use of several frameworks from the Eclipse Modeling Project, but its main goal is to hide those frameworks' intricacy from the tool developer. Cinco provides three simple textual specification languages (MGL, MSL, and CPD, see below) for the high-level characterization of the developed tool's structural and visual features. Above that, Cinco is also a full Java IDE, so that the portions of the developed tool that are not described in a declarative way can seamlessly be developed in the same environment.

Crucial to Cinco is the concept of code generation. Much of the code of a modeling tool that is developed with Cinco (in the following called Cinco Product, or just CP) is generated automatically from abstract specifications, such as the already mentioned MGL, MSL and CPD files. It is of utmost importance that you *never* make any changes to automatically generated code[1], as all those changes will be gone once the code generation is run again.

1.3 Download & Installation

As Cinco is a full Eclipse-based IDE including the Eclipse Modeling Tools (EMF, Xtext, Graphiti, etc.) and the Java Development Tooling (JDT) in addition to our own plug-ins the download size is almost 300 MB. We are planning to provide the Cinco-specific plug-ins via an Eclipse update site in the future, but for now you need to download the whole package as one.

Please note: Cinco itself is developed under the "Eclipse Public License v. 1.0", but the executable build contains several libraries from the jABC project, for which the following license applies:

```
================================================================================
This software is free to use.

You are not allowed to redistribute, modify or decompile this software.

Third-party components are included for which these restrictions may not apply.
Please refer to the documentation for details.

THIS SOFTWARE IS PROVIDED ''AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF
MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
================================================================================
```

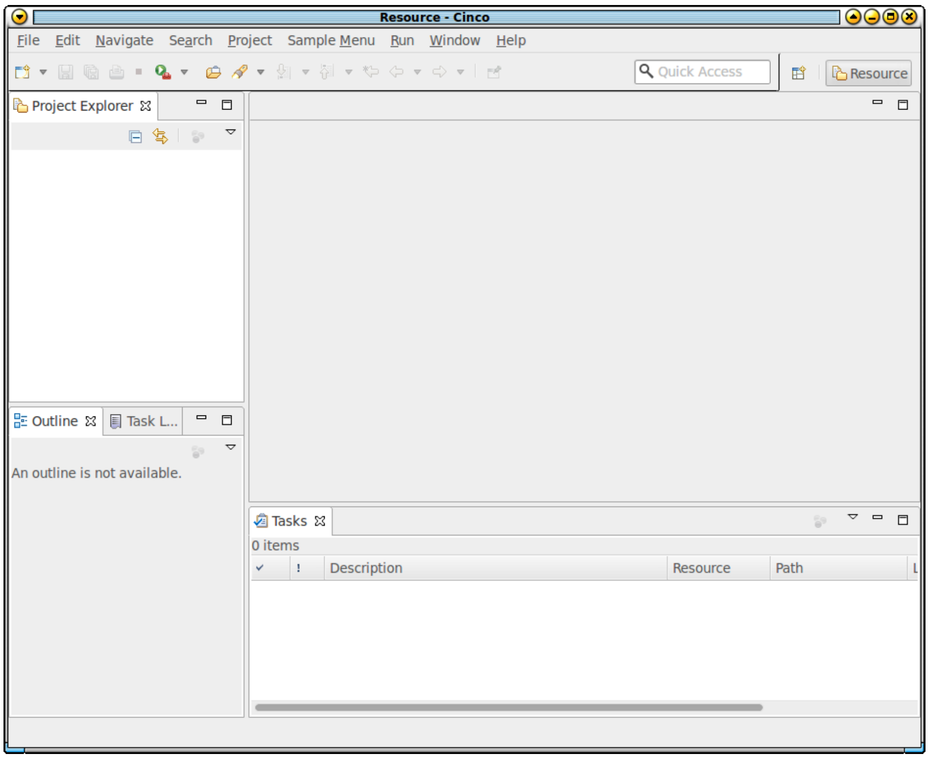
Cinco requires Java JDK 8 to be installed on your system as main Java installation (i.e. JAVA_HOME pointing there). Apart from that, you just have to choose the correct zip file for your operating system, unzip it and execute the included binary: `cinco` for Mac and Linux, `cinco.exe` for Windows.

1.4 Getting Started Tutorial

1.4.1 Launching Cinco for the First Time

When launching Cinco it will first query you for the location of a workspace directory. Choose some place on your harddisk that is not within the Cinco installation folder. This workspace will contain all your Cinco projects and the according metadata, as well as sub-projects, specification files, source code etc. It is possible to use different workspaces (e.g. for different projects). If you don't want to use multiple ones for now, you can check «Use this as default and do not ask again» before clicking «OK».

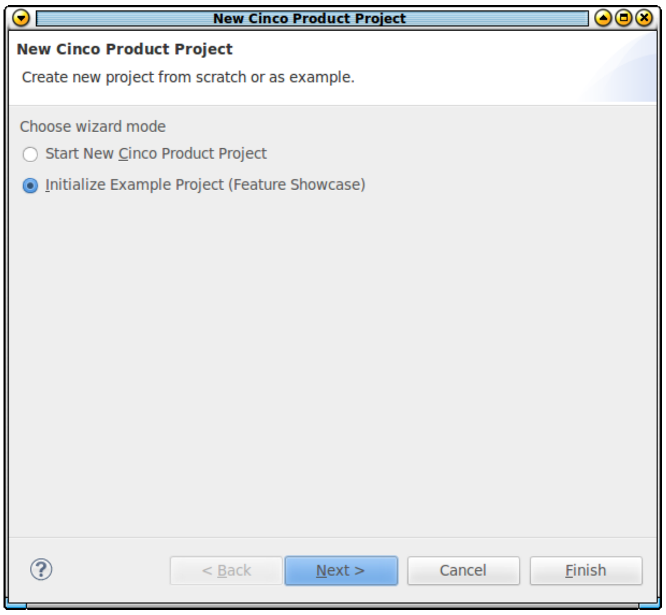
After choosing the workspace Cinco should start and look like Fig. 1.1. The GUI is partitioned into so-called *Views*. For now, the «Project Explorer» and the editor area (the big empty grey space on the right) are the most important ones.



An outline is not available.

0 items

Figure 1.1: First startup should look like this



New Cinco Product Project

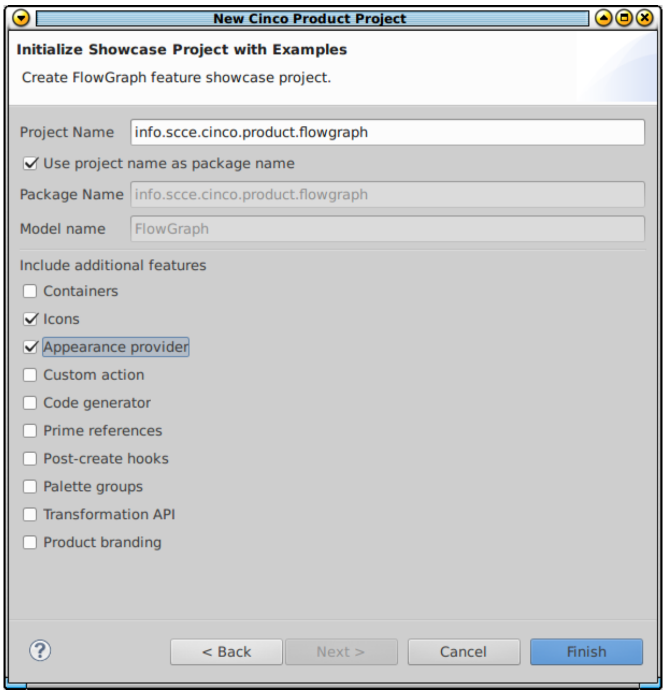
Create new project from scratch or as example.

Choose wizard mode

( ) Start New Cinco Product Project

[✓ Initialize Example Project (Feature Showcase)]

Figure 1.2: Use the wizard to create a new example project


New Cinco Product Project

Initialize Showcase Project with Examples

Create FlowGraph feature showcase project.

Project Name: info.scce.cinco.product.flowgraph

☑ Use project name as package name

Package Name: info.scce.cinco.product.flowgraph

Model name: FlowGraph

Include additional features

- ☐ Containers
- ☑ Icons
- ☑ Appearance provider
- ☐ Custom action
- ☐ Code generator
- ☐ Prime references
- ☐ Post-create hooks
- ☐ Palette groups
- ☐ Transformation API
- ☐ Product branding

Figure 1.3: Select example features that shall be generated

1.4.2 Creating a Cinco Product Project

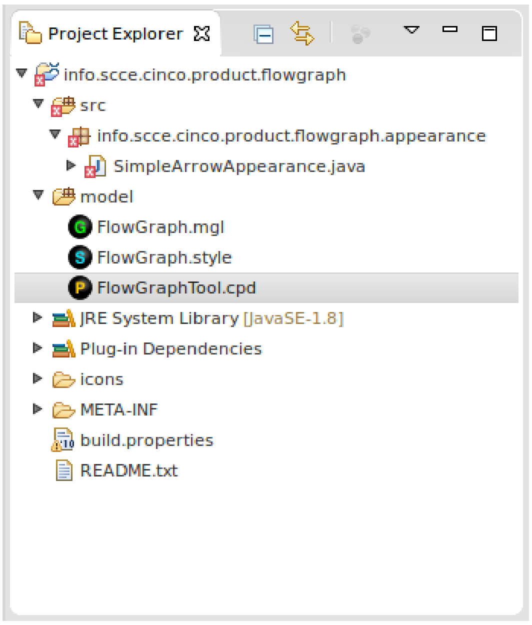
To get a new *Cinco Product* project initialized, we'll now use the built-in wizard to create some example files to start working with. Right-click somewhere within the «Project Explorer» and choose «New / Cinco Product Project». The wizard as depicted in Fig. 1.2 appears. The wizard has two modes. You can either start a completely new project from scratch or generate an example project showing off some of the Cinco features. We'll go for the latter one for now. Select «Initialize Example Project (Feature Showcase)» and hit «Next».

The next page of the wizard now lets you configure some basic things for your example project (cf. Fig. 1.3). Please note that the model name is fixed, as the generated example is a flow graph. For now, you probably should leave the project/package name as suggested; if you create additional examples with the wizard later, you can choose different names here to distinguish them. Each feature you select at the bottom will generate additional parts in the example. We suggest to only use «Icons» and «Appearance provider» to keep the files manageable small for your first attempt using Cinco. Confirm your selection with «Finish»

After running the wizard, the project explorer contains your freshly created project and a `README.txt` (which was generated based on the features you previously selected) is opened in the editor area. For now, you don't need to read it, as it contains information similar to this tutorial, but once you generate the example project with additional selected features, you should do so. Depending on the names you've chosen, the project explorer now looks somewhat similar to Fig. 1.4a. Please note that the compile error marked at `SimpleArrowAppearance.java` is not unexpected. The Java class refers to other Java entities that have not yet been generated.

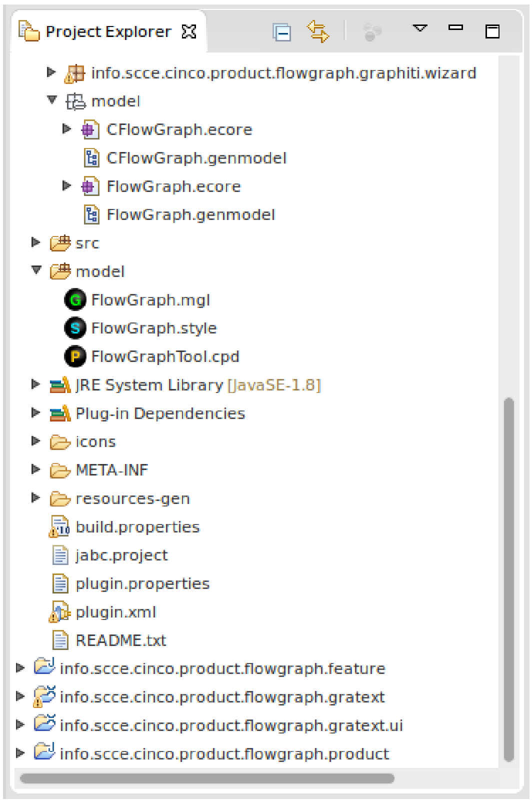
You might want to take a short look into those initialized files (especially the ones in the `model` folder), but before we delve more deeply into their contents (which we will do in Sec. 1.5), let's first generate this example CP and give it a try. Just right-click on the `FlowGraphTool.cpd` and choose «Generate Cinco Product». Shortly after, a dialog reporting success should appear and your project explorer looks like Fig. 1.4b. Several new files, directories, and projects – which will be even more once you start using

[1]Of course, this does not apply to automatically generated 'source' files (such as examples and templates). Those are generated for convenience to have a basic version to start working on.



```
Project Explorer
▼ info.scce.cinco.product.flowgraph
  ▼ src
    ▼ info.scce.cinco.product.flowgraph.appearance
      ▶ SimpleArrowAppearance.java
  ▼ model
      FlowGraph.mgl
      FlowGraph.style
      FlowGraphTool.cpd
  ▶ JRE System Library [JavaSE-1.8]
  ▶ Plug-in Dependencies
  ▶ icons
  ▶ META-INF
    build.properties
    README.txt
```

(a) Project after running the wizard



```
Project Explorer
  ▶ info.scce.cinco.product.flowgraph.graphiti.wizard
  ▼ model
    ▶ CFlowGraph.ecore
      CFlowGraph.genmodel
    ▶ FlowGraph.ecore
      FlowGraph.genmodel
▶ src
▼ model
    FlowGraph.mgl
    FlowGraph.style
    FlowGraphTool.cpd
▶ JRE System Library [JavaSE-1.8]
▶ Plug-in Dependencies
▶ icons
▶ META-INF
▶ resources-gen
  build.properties
  jabc.project
  plugin.properties
  plugin.xml
  README.txt
▶ info.scce.cinco.product.flowgraph.feature
▶ info.scce.cinco.product.flowgraph.gratext
▶ info.scce.cinco.product.flowgraph.gratext.ui
▶ info.scce.cinco.product.flowgraph.product
```

(b) Project after code generation

Figure 1.4: Project file system view before and after CINCO product generation

advanced features of CINCO, such as meta plug-ins – have been generated. Also, the compile error we've noticed before is gone.

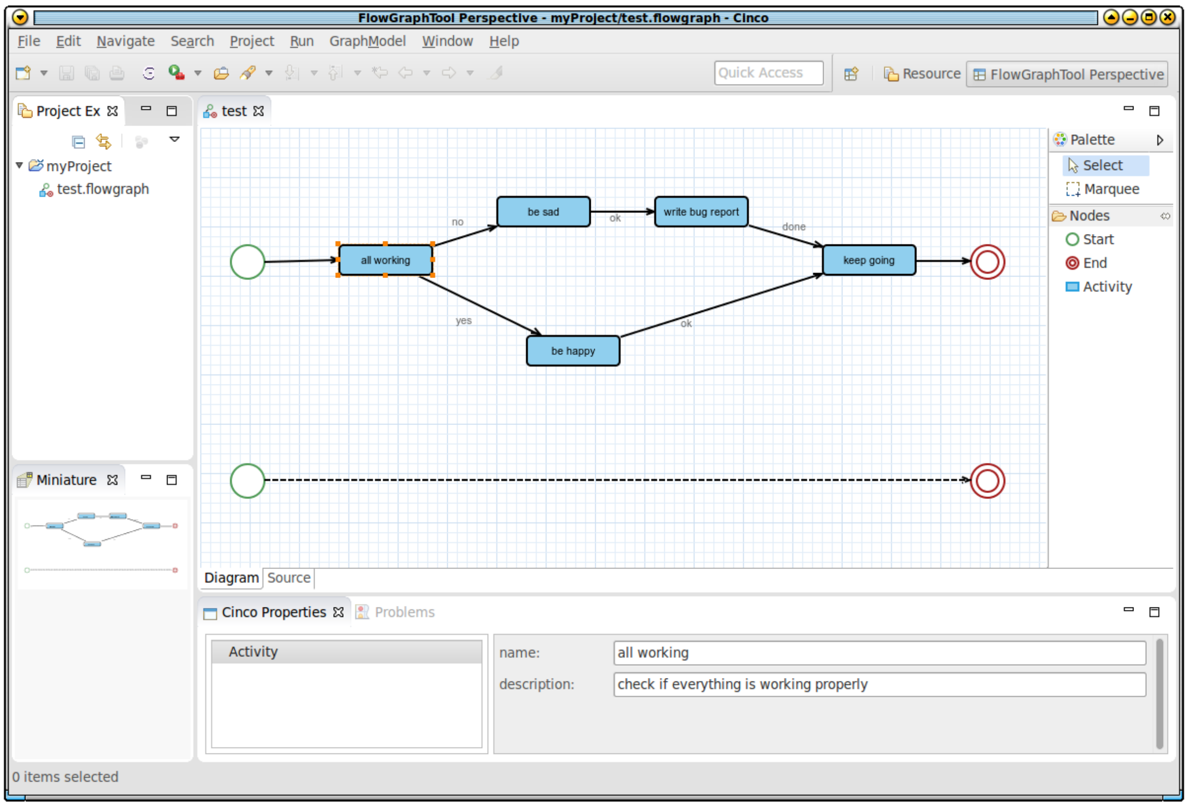
Whenever you change the CP's specification files, you need to re-generate the code. Previously generated code (i.e. everything that is in a *-gen folder and all generated projects) will be deleted on re-generation. But as you never do any manual changes to the generated code, this is no problem.

1.4.3 Testing the CINCO Product

Now, let's test the tool we just generated. Within the project explorer, right-click on the root folder of your main project and choose «Run As / Eclipse Application». You should now see the CINCO splash screen again, starting a different Eclipse product: your modeling tool. [2]

Again, you should see an empty tool like Fig. 1.1. To create a new model we first need a new project where it can be placed. This time, however, we don't need a CINCO product project (as we don't want to develop another CINCO product within the just started CINCO product). Right-click somewhere in the project explorer and select «New / New FlowGraphTool Project», give the project some name and click «Finish». Then, right-click on your newly created project and select «New / New FlowGraph» to create your first CP-specific model.

[2]Please note: When you use the here presented 'simple way' of starting your product (via the «Run As» menu) you will actually start your tool including *all* the CINCO plug-ins and libraries. In fact, you start a second CINCO enriched with the new features we just generated. In the long run, this not what is desired, because a tool developer needs to work with different tools than the tool user. You might have noticed the new .product project. This is a generated Eclipse product definition that provides an easy way of exporting your tool with an own branding (splash screen, icons, etc.) into a standalone application. For testing purposes, however, the easy way should be favored, as it does not impose any obstacles other than ample unnecessary GUI elements.



name: all working
description: check if everything is working properly

Figure 1.5: The first model that was modeled using the example Cinco product

The editor area now shows your empty model with a drawing grid. On the right side you see a «Palette», containing the elements you can now use for modeling. The elements you see there are in the «Palette / Nodes» category: `Start`, `End`, and `Activity`. Those are the three node types that are were automatically included in the `FlowGraph.mgl` by the wizard. Begin modeling by just drag&dropping a few different nodes into the modeling area. To connect nodes, you can just hover over one and drag the small appearing arrow icon onto another one. For the `Activity` node type and the edges originating from them, you can use the «Cinco Properties» view (bottom of your window) to set attributes; `name` and `description` for the node as well as `label` for the edge. An activity's name is displayed within the node shape (the blue box) and the edge label is shown as a floating (and movable) textbox next to the edge. Fig. 1.5 shows an example how your tool might look now.

The modeling elements abide by the following syntactic rules:

- `Start` nodes (green circle) have at most one outgoing edge of type `Transition`, but no incoming edges.
- `End` nodes (double red circle) have arbitrarily many incoming edges, but no outgoing.
- `Activity` (blue box) nodes have multiple outgoing edges of type `LabeledTransition`, and may have multiple incoming edges of both types. The attribute `name` is displayed centered within the box.
- `LabeledTransition` edges connect `Activity` nodes to arbitrary other nodes and have their attribute `label` displayed as movable caption next to it.
- `Transition` edges only originate from `Start` nodes. If such an edge directly leads to an `End` node, it is displayed with a dashed line.

Those rules are all defined within the example Cinco specification files that were created by the wizard. The next section will therefore provide a detailed discussion on the contents of those files.

1.5 Discussion on the Example Files

We consider our specification formats MGL, MSL, and CPD fairly self-explanatory, meaning that one usually can understand most of their contents without detailed knowledge on available keywords or syntactic structures. This section will nonetheless detail on our two example files, but you might want to have a look at your `FlowGraph.mgl`, `FlowGraph.style`, and `FlowGraphTool.cpd` beforehand, to decide for yourself.

However, writing (instead of just understanding) a programming or specification language usually is a totally different matter. But here, CINCO heavily benefits from being implemented with libraries from the Eclipse Modeling Project, in this case especially Xtext: The so-called *Content Assist* that is generated for every Xtext-based editor provides a very useful autocompletion when Ctrl + Space is pressed (if you ever programmed Java using Eclipse, you most likely know this shortcut already). With this feature it is often not even required to read the documentation, as one can autocomplete through the whole specification file, manually inserting a name/identifier every now and then.

Modeling tool development with CINCO centers around two textual specification files. On the one hand we have a file in the "Meta Graph Language" (MGL) that contains the structural information on the tool's model. On the other hand, a "Meta Style Language" (MSL) file[3] is required to specify the visual characteristics (e.g. shapes and colors) of this model. The third specification format is the "CINCO Product Definition" (CPD), which is the entry point for the generation. In it, (potentially) multiple MGL files can be given that are generated in one sweep. The following subsections will each provide an introduction to one of this specification languages alongside the example files created by the new project wizard.

1.5.1 MGL File

Listing 1.1 shows the `FlowGraph.mgl` as it was initialized in Sec. 1.4.2. Aside from some general configuration stuff in lines 1-7 and an attribute declaration in line 9, it contains three two different kinds of modeling component declaration:

- Node type definitions (3x)
- Edge type definitions (2x)

Each of those **`node`** `{...}` or **`edge`** `{...}` blocks can be preceeded by a varying number of annotations (starting with the @-symbol). Also following the syntax of Java, everything from `//` to the end of the line is ignored (single line comment), as well as everything from `/*` to `*/` (multi-line comment).

Node Types

Each node type defines one kind of component that can be inserted into the graphModel by drag&dropping it from the «Palette» into the modeling area. Doing so will create a dedicated instance of this node type, so that attributes (like name or description in the Activity node, cf. lines 36 and 37) can be set separately for each occurance in the model.

Node types can declare an arbitrary amount of such attributes. An attribute's name (written after the keyword **`as`**) is displayed as a label next to the editing field in the «Cinco Properties» view. Possible data types for attributes (written directly after the keyword **`attr`**) are currently `EString`, `EInt`, `EDouble` and enumerations defined within the MGL (not in this example).

[3]Please note that the file extension for MSL is currently `.style`. We are planning to change this to `.msl` in a future release.

```

@style("model/FlowGraph.style")
graphModel FlowGraph {
  package info.scce.cinco.product.flowgraph
  nsURI "http://cinco.scce.info/product/flowgraph"
  iconPath "icons/FlowGraph.png"
  diagramExtension "flowgraph"

  attr EString as modelName

  @style(greenCircle)
  @icon("icons/Start.png")
  node Start {
    // allow exactly one outgoing Transition
    outgoingEdges (Transition[1,1])
  }

  @style(redCircle)
  @icon("icons/End.png")
  node End{
    /*

    allow an arbitrary number (>0) of incoming edges

    the following would have been valid as well, meaning the same:
      incomingEdges (*[1,*])

    */
    incomingEdges ({Transition,LabeledTransition}[1,*])
  }

  // use the "blueTextRectangle" as style and pass the attribute "text" as parameter
  @style(blueTextRectangle, "${name}")
  @icon("icons/Activity.png")
  node Activity {
    attr EString as name
    attr EString as description
    incomingEdges (*[1,*])
    outgoingEdges (LabeledTransition[1,*])
  }



  @style(simpleArrow)
  edge Transition {
  }

  @style(labeledArrow, "${label}")
  edge LabeledTransition {
    attr EString as label
  }
}
```

Listing 1.1: FlowGraph MGL file

The definition how the different node types can be connected using edges is done with the **`incomingEdges`** and **`outgoingEdges`** keywords. Enclosed in parantheses you can express a comma-seperated list of constraints, each suffixed by an upper/lower bound range in brackets. Line 39 shows a declaration of outgoing edges using one such contraint. It states that at least one LabeledTransition has to originate from every Activity node. The * that is used as upper bound means "infinity" (i.e. arbitrarily many). If a set of multiple edge types shall be defined per constraint – meaning that any element from the set can be used – they can be explicitely enumerated in curly brackets. Line 29 shows one such set constraint. It expresses that each End node needs at least one incoming edge of the type Transition or of the type LabeledTransition. As there are only those two edge types in our example, this is equivalent to the wildcard expression `(*[1 ,*])`

Node types can have an `@icon` annotation to specify an image that will be displayed in the tool palette next to the type name (cf. Fig. 1.5). The parameter can either be a path relative to the MGL project, as used within our example (lines 12, 19 and 34), or a so-called platform URI which for instance allows to reference files in other bundles. The relative path from line 12 would look like this in absolute notation:

```
@icon("platform:/resource/info.scce.cinco.product.flowgraph/icons/Start.png")
```

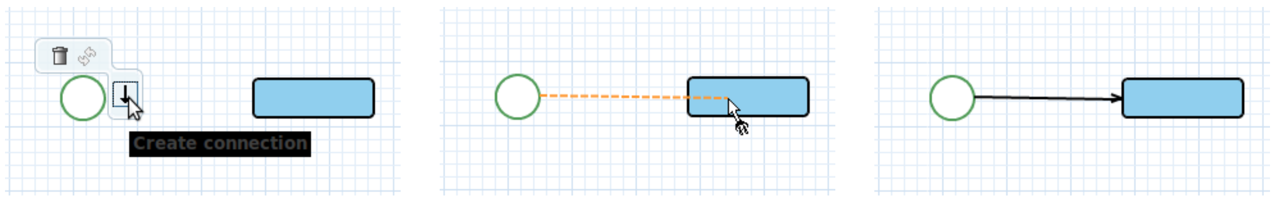
Finally, the `@style` annotation is used to assign a node style from the MSL definition to this node. The first parameter must equal the style identifier (following after the **`nodeStyle`** keyword, see Sect. 1.5.2). After that, a list of further parameters may follow, which can be used to pass the runtime value of attributes to the style. For example, for the Activity node (cf. line 33) the current value of the attribute "name" is passed and displayed as text in the center of the style "blueTextRectangle". If the description attribute needed to be passed as well, the line would look like this:

```
@style (blueTextRectangle, "${name}", "${description}")
```

It is not mandatory to establish a one-to-one relation between nodes and styles like it is done in the generated example. Multiple nodes can use the same style and, for instance, pass different attributes. However, depending on the domain the Cinco product is developed for, it might be a good idea to be able to visually distinguish the different node types.

Edge Types

Edge type declarations are similar to node type declarations. Or more precisely, they allow a subset of the nodes' features: multiple attributes can be defined and a style is given with the `@style` annotation (though this time of course an edge style, not a node style, is referenced).



(a) Hover over node (b) Drag connect icon to target (c) Done

Figure 1.6: Creating a new edge using the connect tool that appears when hovering a node

The major difference is that in the running Cinco product the edge types are not inserted by selecting the corresponding tool from the palette, but by drag&dropping a connect icon that appears when hovering over a node onto a target node. In case multiple edge types are possible (not in the generated example), you will be queried to choose one after releasing the mouse button on the target node.

GraphModel Type

Prior to the definition of node and edge types some global things can be specified for your Cinco product's model type using a **graphModel** {...} block[4]. This time, the @style annotation points to your FlowGraph.style, but attributes are defined identically as for nodes and edges.

The **package** declaration defines the Java package name that should be used as base package for all generated classes and packages. If you browse the generated source files in your main project and the other generated ones, you'll notice that every package is prefixed with the one defined here. While this is intended to provide "unique" fully qualified names in the Java world, the **nsURI** keyword defines the unique identifier for the generated Ecore metamodel.

The **diagramExtension** keyword determines the file extension that will be used for your models and, last but not least, an icon file for those model can be defined using the **iconPath** keyword. It will be used on several occasions in the final product, but most importantly to highlight your created models in the «Project Explorer» (cf. Fig. 1.5)

1.5.2 MSL File

Listing 1.2 shows the FlowGraph.style as it was initialized in Sec. 1.4.2. It consists of appearances, node styles, and edge styles, which will individually be discussed in the following paragraphs.

Appearances

Appearances are used to group some basic visual attribues (such as colors, fonts, or line width and style) under one name, so that it can be referenced by the other graphical elements. They are primarily defined using an **appearance** {...} block. For instance, lines 1-4 define an appearance named "default" that sets the background color to some light blue and the line width to 2 pixels. Colors are RGB values, but for convenience Ctrl + Space can be used to open a special color chooser window.

The keyword **extends** allows to inherit the values from another appearance and add/overwrite selected ones (cf. line 6). Also, *inline appearances* are supported: Wherever an appearance can be set (cf. node/edge styles below), an unnamed **appearance** block can be inserted to directly specify those values. Such an inline appearance can also inherit from a named one using the keyword **extends** (cf. line 25).

Furthermore, it is sometimes necessary to change the appearance of a graphical element subject to some information that is only available at runtime. This could, for instance, be a second cirlce to identify accepting states in a state diagram. Also, the different appearances of the Transition edge, depending on whether it is connected to an End node or an Activity node (as discussed in Sec. 1.4.3), requires the appearance not to be static.

For such cases, Cinco supports *dynamic appearance providers*, which are special Java classes implementing the StyleAppearanceProvider interface and whose getAppearance method is called at runtime to determine the current appearance of the element. They are referenced in MSL using the **appearanceProvider** keyword (cf. line 49).

[4]Actually, the **node** and **edge** blocks are located *in* this **graphModel** block. See Listing 1.1.

```
appearance default {
  lineWidth 2
  background (144,207,238)
}

appearance redBorder extends default {
      foreground (164,29,29)
      background (255,255,255)
}

nodeStyle redCircle {
  ellipse {
    appearance redBorder
    size(36,36)
    ellipse {
      appearance redBorder
      position ( CENTER, MIDDLE )
      size (24,24)
    }
  }
}

nodeStyle greenCircle {
  ellipse {
    appearance extends default {
      foreground (81,156,88)
      background (255,255,255)
    }
    size(36,36)
  }
}

nodeStyle blueTextRectangle(1) {
  roundedRectangle {
    appearance default
    position (0,0)
    size (96,32)
    corner (8,8)
    text {
      position ( CENTER, MIDDLE )
      value "%s"
    }
  }
}



edgeStyle simpleArrow {
  appearanceProvider ( "info.scce.cinco.product.flowgraph.appearance.SimpleArrowAppearance" )

  decorator {
    location (1.0) // at the end of the edge
    ARROW
    appearance default
  }
}

edgeStyle labeledArrow(1) {
  appearance default
  decorator {
    location (1.0)
    ARROW
    appearance default
  }
  decorator {
    location (0.3)
    movable
    text {
      value "%s"
    }
  }
}
```

Listing 1.2: FlowGraph MSL file

Node Styles

Each node style is defined using a `nodeStyle {...}` block and consists of at least one primary *shape* declaration. For example, the "greenCircle" style (lines 23-31) consists of a single `ellipse` shape with width and height of 36 pixels (making it a circle) and an inline appearance that extends the appearance "default". If the primary shape is a *container shape*, additional child shapes can be included, which is done for the style "redCircle" (cf. lines 15-19). Positioning of child shapes is done with the keyword `position` and can either be absolute or relative to the parent shape. If no position is given, the upper left corner of the bounding box is placed at (0,0) of the parent shape. The position of the primary shape determines the placement relative to the mouse position where the create tool was dropped.

The following container shapes are currently supported:

`rectangle` defines a rectangle of given width and height using the `size` keyword

`roundedRectangle` is similar to the normal rectangle, but additionally provides the `corner` keyword that allows to set the width and height of the rounded corners. Setting this to (0,0) results in a normal rectangle.

`ellipse` defines an ellipse whose bounding box is determined by the `size` keyword

`polygon` can be used, if the other predefined keywords are not enough to express the desired shape of the node. With the `points` keyword a list of coordinates can be given that are connected. The last point is automatically connected to the first one, closing the polygon.

Additionally, the following simple shapes exist:

`polyline` works similar to the polygon container shape, except that it is not closed from last to first point and therefore no background color is set.

`text` creates a text label that shows the text given with the keyword `value`.

`multiText` is the same as text, except that it automatically breaks lines in case they do not fit in the box' width.

As introduced in Sect. 1.5.1, the runtime values of a nodes' attributes can be passed on as parameters to the style. In order to do so, the number of parameters needs to be declared in parantheses after the style name (cf. line 33 in Listing 1.2). Then they can be used within the `value` declarations of `text` and `multiText` with the Java String format syntax[5]. The simplest way typically is to use `%s` that causes the `toString` method to be called. In case multiple passed parameters need to be referenced individually in different `text` elements, they can be indexed with a number: `%1$s` and `%2$s` etc.

Edge Styles

Edge styles are defined using `edgeStyle {...}` blocks. Each edge style consists of two things: an appearance (either inline, referenced or as appearance provider as introduced above) as well as an (optional) list of `decorator {...}` blocks. While the appearance simply sets the edge's color, width and style, decorators are used to add arrow tips, labels, or other markers to the edges. Such a decorator mainly consists of a single shape as introduced in the node style section (simple or container, but not hierarchical). With the keyword `location` an initial position of the decorator can be given by means of a floating point value between 0.0 (start of the edge) to 1.0 (end of the edge). The real position will be calculated according to actual the length of the edge, e.g., the text decorator of the labeledArrow (cf. line 66) is located at 30% of the edge length. The keyword `movable` allows the decorator to be freely moved away from its initial position (this makes more sense for caption labels than it does for arrow tips ;)).

[5] See `http://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html` for the complete documentation of possibilities.

```
cincoProduct FlowGraphTool {

  mgl "model/FlowGraph.mgl"

}
```

Listing 1.3: FlowGraph CPD file

```
cincoProduct FlowGraphTool {

  mgl "model/FlowGraph.mgl"

  splashScreen "branding/splash.bmp" {
    progressBar (37,268,190,10)
    progressMessage (37,280,190,18)
  }

  image16 "branding/Icon16.png"
  image32 "branding/Icon32.png"
  image48 "branding/Icon48.png"
  image64 "branding/Icon64.png"
  image128 "branding/Icon128.png"

  about {
    text "This is the example project for the Cinco SCCE Meta Tooling Suite ( http://cinco.scce.info )
        that serves as a feature showcase. It is generated using the 'New CincoProduct' wizard"
  }


}
```

Listing 1.4: FlowGraph CPD file with product branding

As the manual specification of arrow shapes using a **polyline** can become quite annoying, CINCO provides some *decorator templates* that should cover most use cases: **ARROW**, **CIRCLE**, **DIAMOND**, and **TRIANGLE**. In combination with an appropriate appearance to simulate filled and unfilled versions, e.g. UML's most prominent arrow types can simply be defined.

1.5.3 CPD File

Up to CINCO version 0.5.1 a single MGL file was the central artifact of a CP. Thus, it was not easily possible to combine multiple MGL files into one application[6]. As of version 0.6 CINCO therefore provides a new specification language to serve as entry point for the generation of multiple MGLs. Listing 1.3 shows the `FlowGraph.cpd` as it was initialized in Sec. 1.4.2. For this first example the file is quite small, only providing two things:

1. the name of the CP, here "FlowGraphTool", is given after the **cincoProduct** keyword
2. the only MGL file of our project is given after the **mgl** keyword

Beyond the possibility to have multiple interconnected CINCO-based model types, CINCO also generates the appropriate Eclipse bundles from the CPD that allow for an easy export of the CP into a standalone application. It is even possible to define a branding (i.e. dedicated startup splash screen, about text, icons, etc.) within the CPD file. For reference, Listing 1.4 shows the product definition, when «Product branding» is selected in the CINCO product creation wizard (cf. Sect. 1.4.2).

[6]It could be done by specifying multiple MGLs in different Eclipse bundles and then manually managing the dependencies between them and the additionally generated bundles. However, this requires more detailed knowledge on internal Eclipse plug-in development structures and is not suitable for the target audience of CINCO.

2 Language Reference

... will eventually be included in this manual.

Bibliography

[BFK+16] Steve Boßelmann, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff, and Bernhard Steffen. DIME: A Programming-Less Modeling Environment for Web Applications. In *Proc. of the 7th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*, 2016.

[Kop14] Dawid Kopetzki. Model-based generation of graphical editors on the basis of abstract meta-model specifications. Master thesis, TU Dortmund, June 2014.

[NNMS16] Stefan Naujokat, Johannes Neubauer, Tiziana Margaria, and Bernhard Steffen. Meta-Level Reuse for Mastering Domain Specialization. In *Proc. of the 7th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*, 2016.

[NTI+14] Stefan Naujokat, Louis-Marie Traonouez, Malte Isberner, Bernhard Steffen, and Axel Legay. Domain-Specific Code Generator Modeling: A Case Study for Multi-faceted Concurrent Systems. In *Proc. of the 6th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I (ISoLA 2014)*, volume 8802 of *LNCS*, pages 463–480. Springer, 2014.

[SN16] Bernhard Steffen and Stefan Naujokat. Archimedean Points: The Essence for Mastering Change. *LNCS Transactions on Foundations for Mastering Change (FoMaC)*, 1(1), 2016.

[TLK+16] Louis-Marie Traonouez, Axel Legay, Jin Hyun Kim, Mounir Chadli, Bernhard Steffen, Stefan Naujokat, and Kim Guldstrand Larsen. A Model-Based Framework for the Specification and Analysis of Hierarchical Scheduling Systems. In *Proc. of Int. Workshop on Formal Methods for Industrial Critical Systems and Automated Verification of Critical Systems (FMICS-AVoCS 2016)*, 2016.

[WMN16] Nils Wortmann, Malte Michel, and Stefan Naujokat. A Fully Model-Based Approach to Software Development for Industrial Centrifuges. In *Proc. of the 7th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*, 2016.