STEFAN MITSCH

MODELING AND ANALYZING HYBRID SYSTEMS WITH SPHINX

A USER MANUAL

Computer Science Department
Carnegie Mellon University / Johannes Kepler University
http://www.cs.cmu.edu/~smitsch
December 2013

Stefan Mitsch: *Modeling and Analyzing Hybrid Systems with Sphinx,* A User Manual, © December 2013.

WEBSITE:
http://www.cs.cmu.edu/~smitsch

E-MAIL:
smitsch@cs.cmu.edu

CONTENTS

LIST OF FIGURES

1 OVERVIEW

1.1 GENERAL INFORMATION

This document is a user manual for the Sφnx verification-driven engineering environment for hybrid systems. Sφnx is an extensible verification-driven engineering toolkit based on the Eclipse platform. It provides textual and graphical modeling editors to describe the structure, the discrete dynamics, and the continuous dynamics of cyber-physical systems. Sφnx uses KeYmaera as hybrid verification tool.

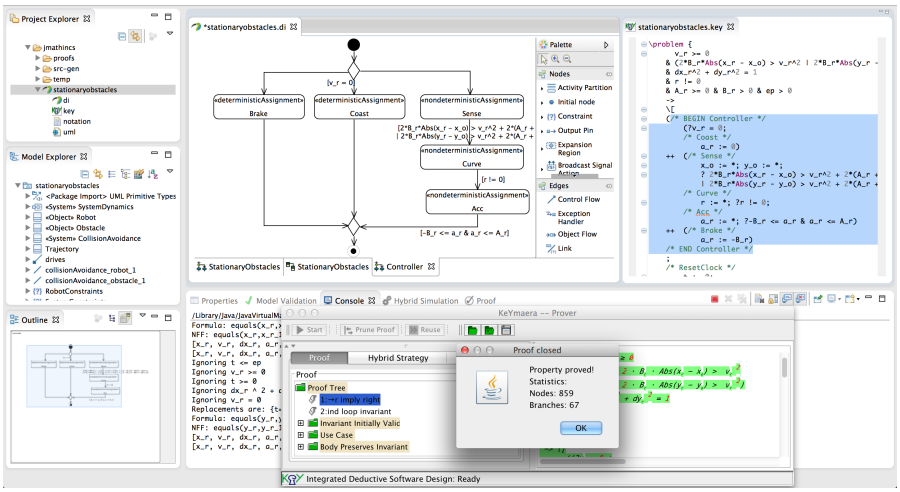


Figure 1.1: Screenshot of the Sφnx toolkit

Details on the underlying principles of Sφnx can be found in [3]. Sφnx was used in formal verification case studies on road traffic safety [2] and obstacle avoidance for autonomous robotic ground vehicles [1].

2 INSTALLATION

Abstract. This chapter introduces the installation procedure of Sφnx and the subsequent configuration steps to setup KeYmaera as hybrid verification tool.

Contents

2.1 INSTALLATION FROM ECLIPSE UPDATE SITE

Sφnx comes with an *Eclipse update site*, which automates installation and updates. It assumes, that Eclipse Kepler (Modeling Tools) is already downloaded from `http://www.eclipse.org` and installed.

To check for Sφnx updates, click Help → Check for Updates

.

1. Start Eclipse
2. Click *Help* → *Install Modeling Components* to open the modeling component selection wizard
3. Select *Xtext* and *Papyrus* and click *Finish*. Follow the on-screen instructions to install these modeling components.
4. Click *Help* → *Install new Software...* to open the Eclipse update manager
5. Click *Add* to add a new Sφnx update site and type the following into the location of the update site: `http://www.cs.cmu.edu/~smitsch/updates/releases` to use the latest tool release.
6. Click *Add* to add a new Xsemantics update site and type the following into the location of the update site: `http://master.dl.sourceforge.net/project/xsemantics/updates/releases/1.3`
7. If you want to compare proofs
   a) Click *Add* to add a new EMF Compare update site.
   b) Type the following into the location of the update site: `http://download.eclipse.org/modeling/emf/compare/updates/releases/`
8. If you want to install the simulation features of Sφnx
   a) Click *Add* to add a new Wolfram Workbench update site. Sφnx installation retrieves the Mathematica integration libraries from this update site. Note: you need a Mathematica license to use the hybrid program simulation features of Sφnx. Type the following

into the location of the update site: `http://workbench.wolfram.com/update`

b) Click *Add* to add a new Eclipse Indigo compatibility update site for Wolfram Workbench and type the following into the location of the update site: `http://download.eclipse.org/releases/indigo`

9. Select Sφnx from the drop-down menu and choose the features to install

10. Follow the screen instructions to complete the installation

2.2 CONFIGURATION

This section details the configuration of Sφnx once it is installed. Sφnx uses KeYmaera as hybrid verification tool.

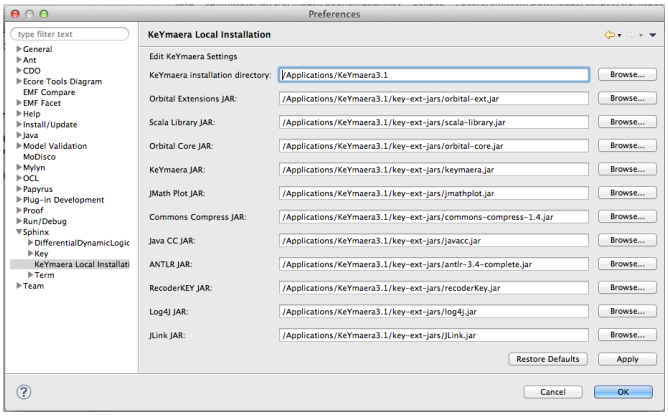
2.2.1 Install and Configure KeYmaera

1. Follow the instructions on the KeYmaera[1] web site to download and install KeYmaera locally. Note, that Sφnx does not yet work with the Webstart version!

2. Click *Eclipse* → *Preferences...* and select *KeYmaera Local Installation* from the tree view.

3. Click *Browse...* on the KeYmaera Installation Directory line and select your local KeYmaera installation directory. Sφnx will try to figure out the library dependencies automatically.

Figure 2.1: Specify the locations of KeYmaera JAR libraries to enable KeYmaera startup from Sφnx



4. If necessary, supply the remaining JAR libraries manually using the respective *Browse...* buttons.

2.2.2 Configure the Editors

Click *Eclipse* → *Preferences...* to open the Eclipse preferences dialog.

1. Select *DifferentialDynamicLogic* → *Compiler* to activate/deactivate LaTeX code generation and configure output directory and further code generation settings.

1 `http://symbolaris.com/info/KeYmaera.html#download`

2. Select *DifferentialDynamicLogic → Syntax Coloring* to change the coloring of terminal symbols, comments, and other syntactical elements of dℒ.

3. Select *DifferentialDynamicLogic → Templates* to change the existing templates or add new ones (see Sect. 2.2.5 for details).

4. Select *DifferentialDynamicLogic → Refactoring* to change the default refactoring settings.

5. Select *Papyrus → Embedded Editors* to set the Sφnx-included dℒ editors as default popup editors for the UML elements Constraint, OpaqueAction, and ControlFlow.

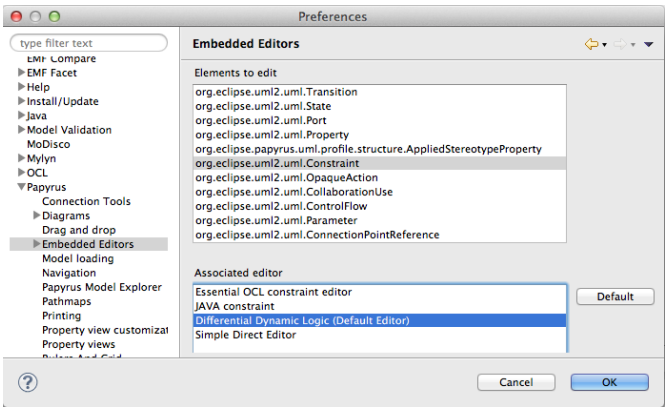


Figure 2.2: Set the dℒ editors of Sφnx as embedded popup editors for Papyrus UML models

2.2.3 Configure Mathematica and Hybrid Program Simulation

1. Select *Sφnx → Mathematica Setup* to let Sφnx access your local Mathematica installation.
   - On MacOS, a typical Mathematica link is similar to `"/Applications/Mathematica.app/Contents/MacOS/MathKernel" -mathlink` (including the quotation marks).
   - On Unix/Linux, use `math -mathlink`.
   - On Windows, use a link similar to `"c:\program files\wolfram research\mathematica\9.0\mathkernel.exe"` (including the quotation marks).

   For debugging purposes, Sφnx logs Mathematica output to the console if activated.

2. Select *Sφnx → Hybrid Program Simulation Preferences* to configure how hybrid program simulation handles nondeterministic choice, nondeterministic repetition and nondeterministic assignment when simulating your models. Note, that in the current version Sφnx makes those choices in a randomized fashion, which means that you may have to simulate multiple times to develop an intuition about possible execution behavior of your program.

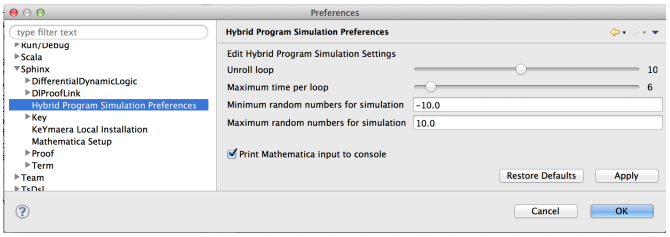


Figure 2.3: Set the number of loop unrollings, the maximum time spent per loop execution, and bounds for random number generation.

2.2.4 Show Additional Views

Click *Window* → *Show View* → *Other...*, then select the following views.

- Select *General* → *Properties*
- Select *Papyrus* → *Model Explorer*
- Select *Sφnx* → *Hybrid Simulation*

2.2.5 Add Modeling Templates

3 TEXTUAL MODELING

Abstract. This chapter introduces the textual modeling features of Sφnx. These include project creation wizard, loading dℒ models to KeYmaera, model refactoring, syntax checking, code completion, outline and quick outline, as well as code folding.

Contents

3.1 CREATE A NEW PROJECT

1. Click *File* →*New* →*Other...*
2. Select *Differential Dynamic Logic Project* and click *Next*

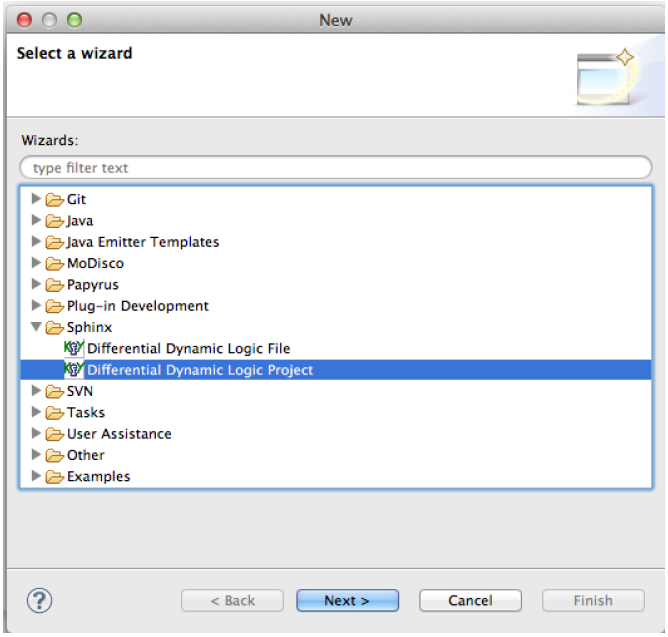


Figure 3.1: Create a new dℒ project using the new project wizard

3. Enter the name of your new project and click *Finish*

The project creation wizard creates a new project with a sample .key-file that shows the principal structure of a theorem in dℒ, including a hybrid program. Details on dℒ can be found on the KeYmaera web site[1], including tutorials and cheat sheets.

Below, we give a of a bouncing ball.

Listing 3.1: Hybrid model of the bouncing ball example in dℒ

1 http://symbolaris.com/info/KeYmaera.html#download

```
/** Hybrid model of a bouncing ball */
\functions {
    R c;        /* damping coefficient */
    R g;        /* gravity */
    R H;        /* initial height */
}
\programVariables {
    /* state variable declarations */
    R h, v, t;
}
\problem {
     /* initial state characterization */
     g>0 & h>=0 & t>=0 & 0<=c & c<1 & v^2 <= 2*g*(H-h) & H>=0
     ->
     \[                              /* system dynamics */
       (
         {h'=v, v'=-g, t'=1, h>=0}; /* falling/jumping */
         if (t>0 & h=0) then        /* if on ground */
            v := -c*v;              /* bounce back */
                t := 0
         fi
       )*                           /* repeat transitions */
     \] (0<=h & h<=H)               /* safety/postcondition*/
}
```

3.2 REFACTOR YOUR MODEL

Sφnx has preliminary model refactoring support in the form of variable renaming. Proofs are not yet adapted automatically.

1. Right-click a variable, select *Refactor →Rename...*

3.3 FURTHER EDITOR FEATURES

- Double-click any tab to make it full-screen.
- Open a searchable quick-outline of your textual model from the context menu of a dℒ textual model

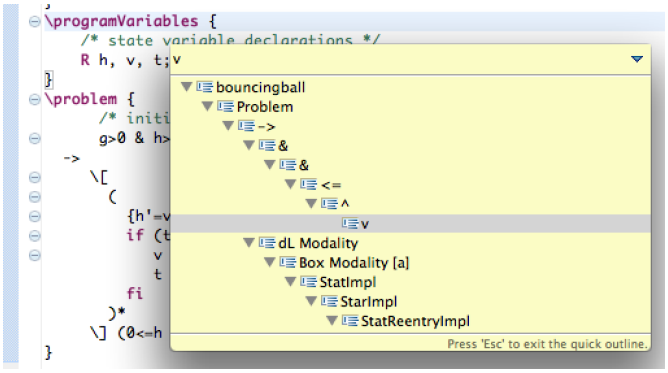


Figure 3.2: Open the quick outline using Ctrl-O *(Windows, Unix) or* Cmd-O *(Mac)*

- Sφnx checks the syntax of hybrid programs for correctness and indicates syntax errors with an light bulb/exclamation mark icon in the left editor bound and a red wiggly line.

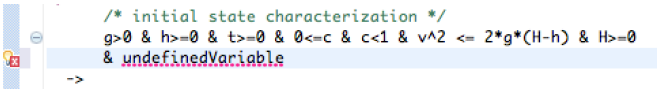


Figure 3.3: Syntax and cross references are checked on-the-fly

- Code completion.

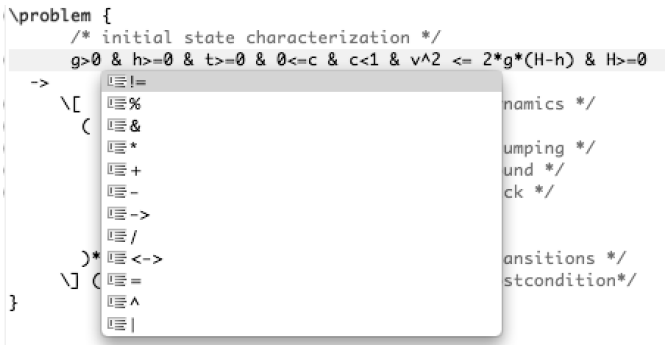


Figure 3.4: Get cross references to variables and other code completion help by pressing Ctrl-Space

- Keep overview with Code Folding using +/- in vertical editor bar.

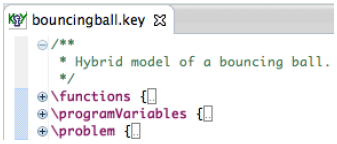


Figure 3.5: Use code folding

- Quick peek folded content as tooltip + in vertical editor bar.

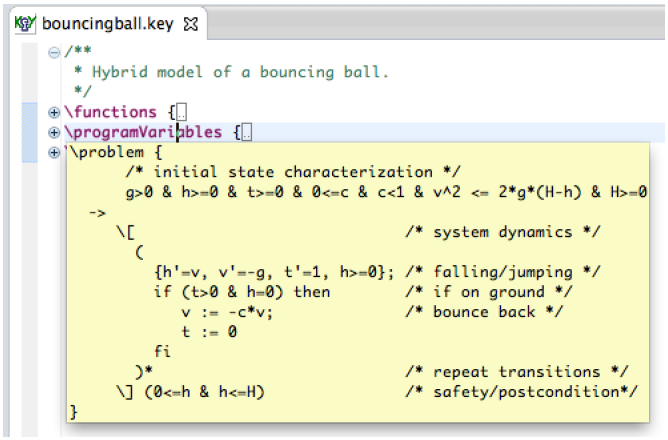


Figure 3.6: Hover the mouse over a folded code snippet to take a quick peek

- Whenever you save a dℒ textual model, Sφnx generates a LaTeX representation of the model.

4 GRAPHICAL MODELING

Abstract. This chapter introduces the graphical modeling features of Sφnx. Sφnx uses UML class diagrams to model the structure of a hybrid system, and UML activity diagrams to model their behavior. Graphical models can be transformed into textual dℒ models and then loaded to KeYmaera.

Contents

4.1 CREATE A NEW GRAPHICAL MODEL

1. Click *File* → *New* → *Other...*
2. Select *Papyrus Model* and click *Next*
3. Enter the name of your new model and click *Next*
4. Select UML and click *Next*
5. Select UML *Activity Diagram* (system dynamics) and UML *Class Diagram* (system structure), then check *A UML model with basic primitive types*.

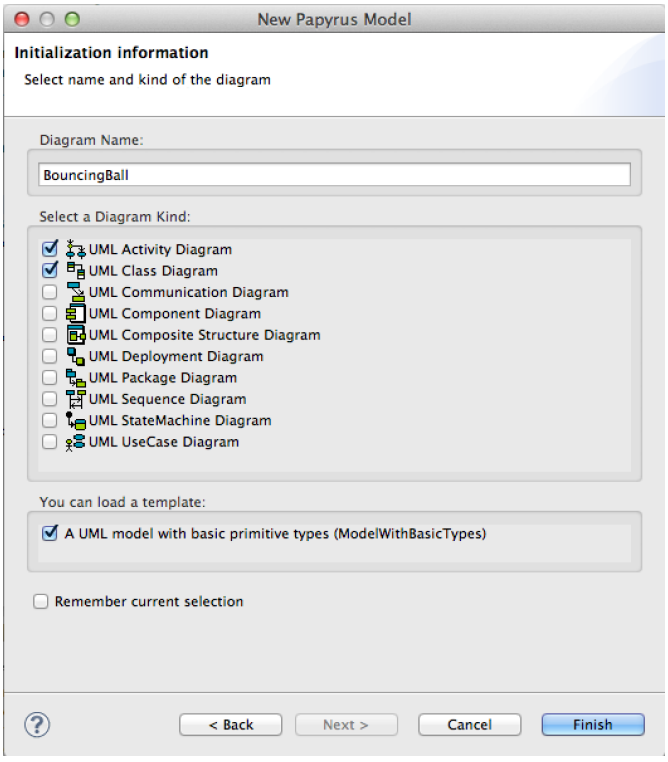


Figure 4.1: Create a new graphical model of a hybrid system describing structure and behavior

6. Click *Finish*

The editor pane now shows the graphical editor. Switch to the class diagram (structure) and follow the steps below to apply the UML profile for dℒ.

1. In the Properties View, select the tab *Profile* and click the button *Apply Registered Profile*.
2. Select *Differential Dynamic Logic Structure* and *Differential Dynamic Logic Behavior* and click *OK*.
3. Check *dldynamic* and *dlstatic* and click *OK*.

4.2 MODEL SYSTEM STRUCTURE

In this section, we discuss how to model the system structure with classes and properties. We will use the stereotypes of the dℒ UML profile to mark important parts of the model for code generation and subsequent verification.

1. From the palette, select *Class* and click on the editing area. Alternatively, wait for the popup palette to appear in an empty part of the editing area. We create two classes: one represents the bouncing ball, the other one the world.

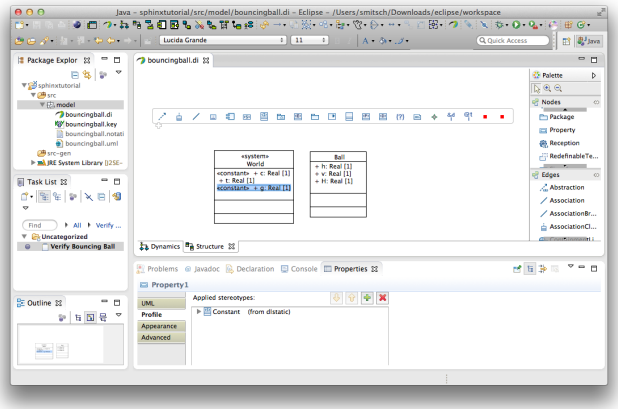


Figure 4.2: Create new classes by dragging Class *elements from the palette to the editing area.*

2. On the properties view, select the profile element. Use *System* to flag the main system class, and *Object* to flag other agents in the system.
3. From the palette, select *Property* and click on a class to add a new property. We add three properties to the class World: a clock t, a damping coefficient c, and gravity g.
4. On the properties view, you can apply stereotypes *Constant* or *Variable* to these properties to flag them as either being constant or variable. By default, all properties are variable. Alternatively, you can use the UML tab to set a property read only, or use the popup editor to add *{readOnly}*

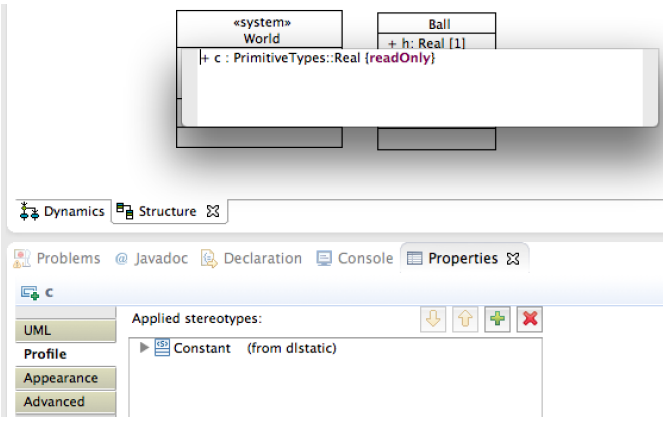


Figure 4.3: Flag properties as constant or variable using stereotypes, setting readonly *to true/false on the* UML *tab of the properties view, or apply* {readOnly} *with the popup editor*

5. Model associations between your classes, as appropriate. Currently, these are for documentation purposes only and do not influence code generation.

6. Select *Constraint* from the palette to define conditions that must always be true. In the bouncing ball example, we add constraints on gravity (must be strictly positive), the damping coefficient (must be positive and less than 1), time (must be positive), and the initial height of the ball (must be positive). The popup editor for constraints in dℒ supports syntax highlighting and code completion. You can confirm the constraint by pressing *Ctrl-Enter*, or leave the popup editor with *Esc*.

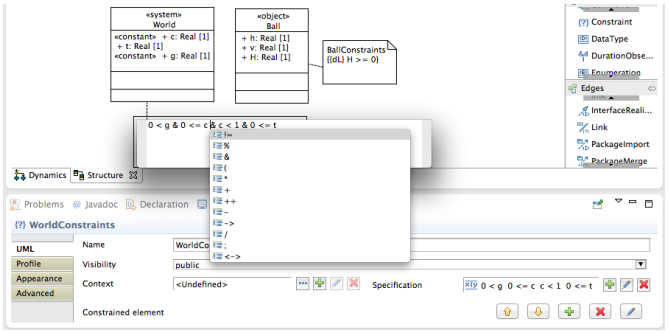


Figure 4.4: Define constraints using the dℒ *popup editor*

4.3 MODEL SYSTEM DYNAMICS

Now that we defined the structure of our system, we can define its discrete and continuous dynamics. Since system dynamics can become rather complicated, we will model hierarchically. We start with the overall system dynamics and will supply details in sub-diagrams. As cautious modelers, we first define a safety condition before we model any behavior.

1. Define the safety condition: select the tab UML of the properties view and scroll to the precondition and postcondition section. Click the button + on the postcondition to open the postcondition window and add a new *Constraint*.

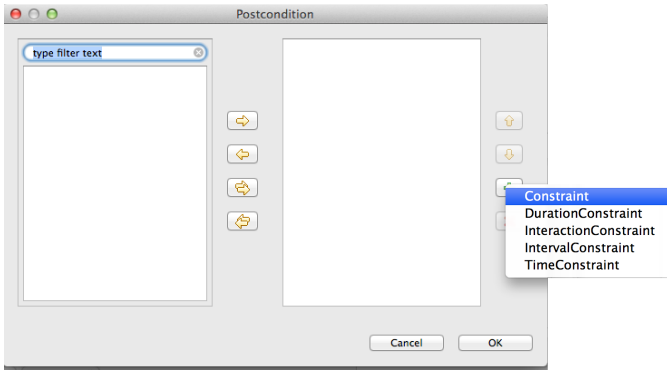


Figure 4.5: Add a new constraint as postcondition for a UML *activity*

2. On the constraint definition window, name the constraint, and optionally select the constrained element. In our example of the bouncing ball, the safety condition will demand that the ball's height is positive but no larger than its starting height, thus we constrain h.

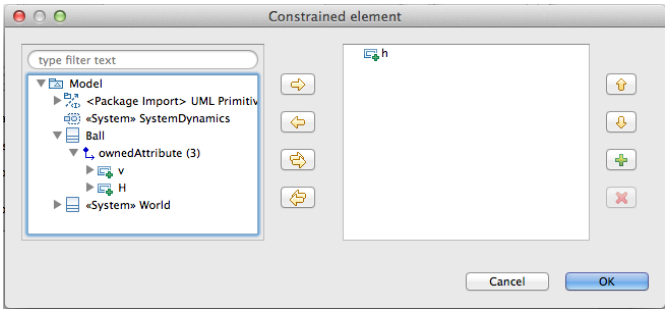


Figure 4.6: Select the constrained element

3. Exit the constrained element selection, the constraint definition window, and the postcondition window. A new postcondition (without specification) has been added to the activity.

4. Click on the postcondition to open the d$\mathcal{L}$ popup editor. Enter the postcondition and confirm with *Ctrl-Enter*.

As an alternative to the popup editor, constraint specifications can be added from the constraint editor (continue from step 2 above).

1. Add a constraint specification using the button + next to the specification text field. Specifications are always of kind OpaqueExpression.

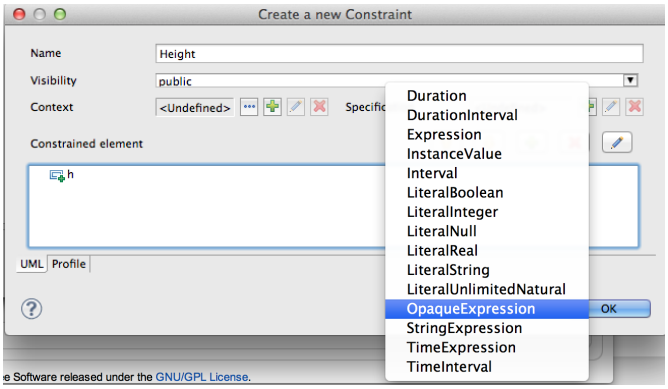


Figure 4.7: Specify the constraint using a new OpaqueExpression

2. On the constraint specification window, name the new specification. Click the button + next to language to add d$\mathcal{L}$. Type *dL* into the text

field on the left side and add it by clicking the right-pointing arrow.

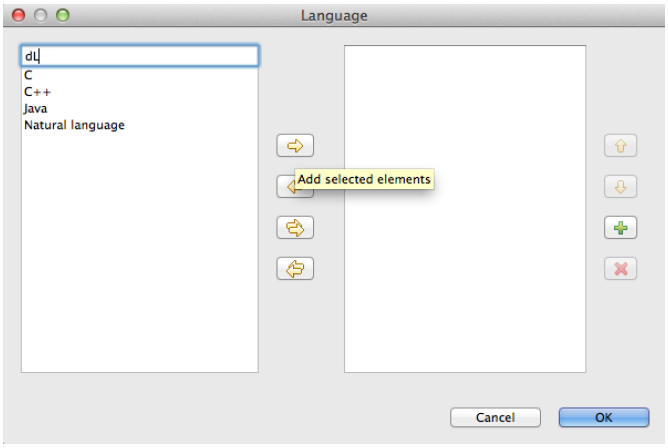


Figure 4.8: Enter dℒ as constraint specification language

3. Click into the body field and provide the constraint specification in dℒ. In our example, we want the ball's height to be between 0 and its initial height H. Click somewhere outside the body field (e. g., reselect the name field) to adopt the new body specification.

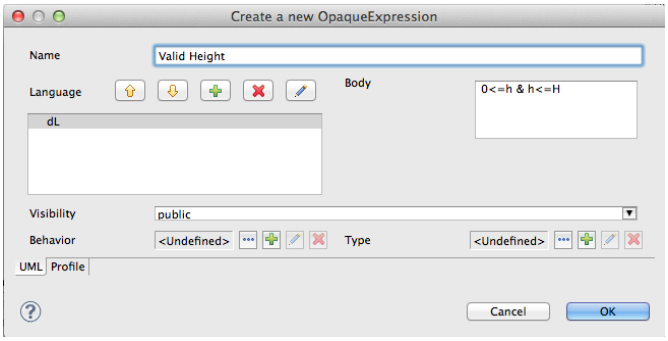


Figure 4.9: Specify the constraint body in dℒ and confirm your definition by clicking outside the text field

4. Exit the constraint specification window, the constraint window, and the postcondition window by clicking *OK* on each.

Next, we define the overall system dynamics. These consist of the continuous dynamics (the ball falls and jumps) and discrete dynamics (when it hits the ground, the ball bounces back). We want the ball fall and jump arbitrariliy often. Thus, discrete and continuous dynamics are repeated nondeterministically many times in a loop.

1. Add an *Initial Node* from the palette. This node represents the start of our system.

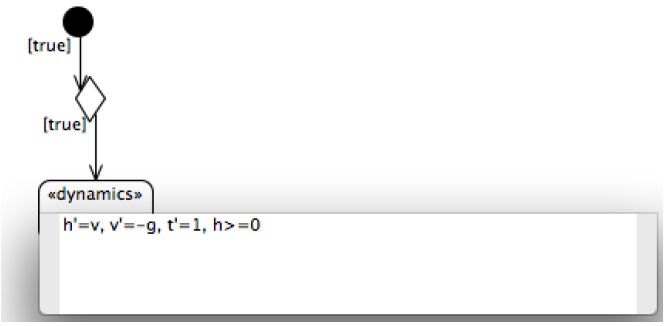
2. Add a *Merge Node* from the palette. This node represents the loop start.

3. Connect the initial node and the merge node with a *Control Flow*. The default condition for such a flow is `true`, which means that it is executed unconditionally. The condition can be hidden using *Filter* → *Manage Connector Labels* from the context menu of the control flow.

4. Add an *OpaqueAction* from the palette and connect the merge node with a control flow to the opaque action. This action represents the continuous dynamics of our system. On the properties view, choose a descriptive name for the action (e. g., "Fall and Jump") and add the stereotype *Dynamics* on the properties view. The opaque action

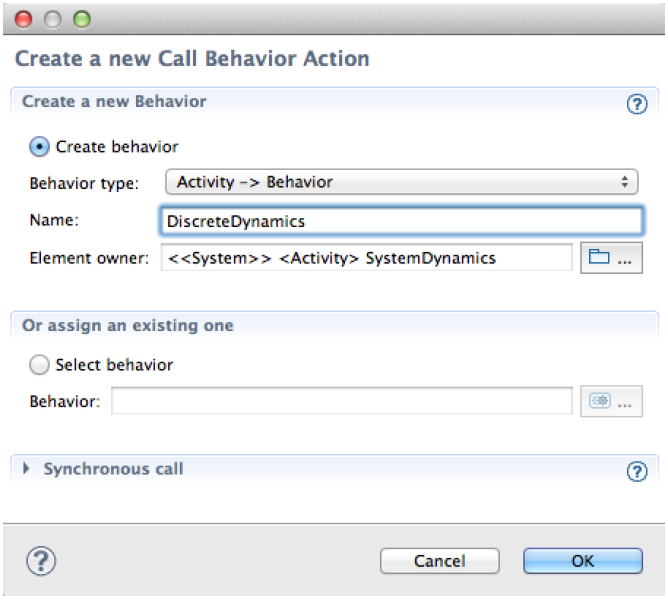
represents the continuous dynamics in our system. Click the "Fall and Jump" label to open the dℒ popup editor and define the dynamics using a differential-algebraic equation.

Figure 4.10: Define the continuous dynamics of a hybrid system as differential-algebraic equation in the dℒ popup editor of an OpaqueAction *with stereotype* Dynamics



5. Add a *Behavior Call Action* from the palette. Although the discrete dynamics of the bouncing ball example is rather simple, we want to have a separate activity diagram to demonstrate decomposition. We create a new behavior named "DiscreteDynamics".

Figure 4.11: Use Call Behavior Actions *to decompose the dynamics and create new* Behavior *nodes*



6. Add a *Decision Node* from the palette. This node represents the end of the loop body.

7. Connect the decision node with the merge node using a *Control Flow* as back edge. Select the stereotype *Nondeterministic repetition* for this control flow. Then specify a loop invariant for the stereotype.

8. Add an *Activity Final Node* and connect the decision node with a control flow. This finalizes the overall system dynamics as follows.

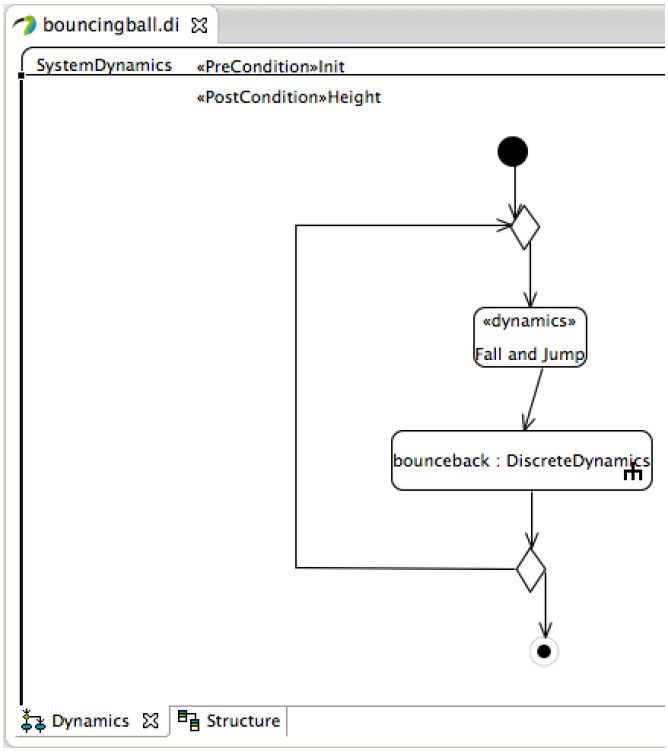


Figure 4.12: Overview of bouncing ball dynamics

In the next step, we will specify the discrete dynamics of the system. The discrete dynamics of the bouncing ball is rather simple: if the ball hits the ground, it bounces back (i. e., the discrete dynamics inverts the ball's velocity and reduces it according to the damping factor), else it just keeps falling or jumping (i. e., the discrete dynamics does nothing).

1. In the Model Editor view, add a *New* → *Activity Diagram* to the DiscreteDynamics behavior created above using the context menu.

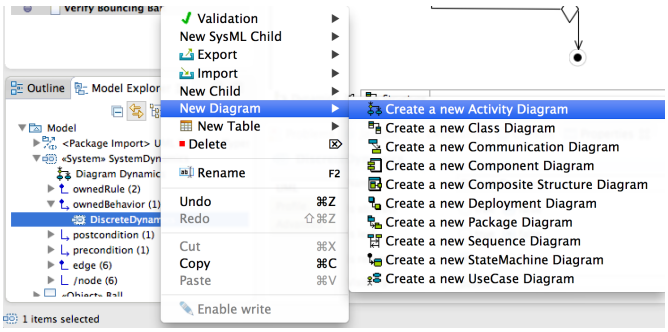


Figure 4.13: Use the context menu to add a new activity diagram to a UML *behavior*

2. Add an *Initial Node* to denote the start of the discrete dynamics.

3. Connect the initial node to a decision node. This is the start of the if-condition.

4. Add an *Opaque Action* to set the new velocity, and another one to reset time. Tag both with the stereotype *Deterministic Assignment* to define that they will set the value of variables in a deterministic manner. Click the label of the action and set the values of velocity ($v := -c \cdot v$) and time ($t := 0$).

5. Add a *Merge Node*, an *Activity Final Node*, and set the control flows to get the final dynamics as below.

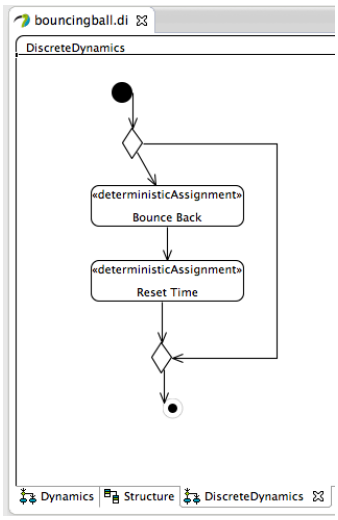


Figure 4.14: The discrete dynamics of the bouncing ball example

6. To set up navigation from the dynamics overview to the detailed discrete dynamics, switch back to the tab *Dynamics*.

7. Double-click the behavior call action *bounceback:DiscreteDynamics* to open the hyperlink configuration window.

8. Click the button + on the tab *View Hyperlinks* to open the *Edit Hyperlink Window*.

9. Click the button *Search* and select the *DiscreteDynamics* activity diagram.

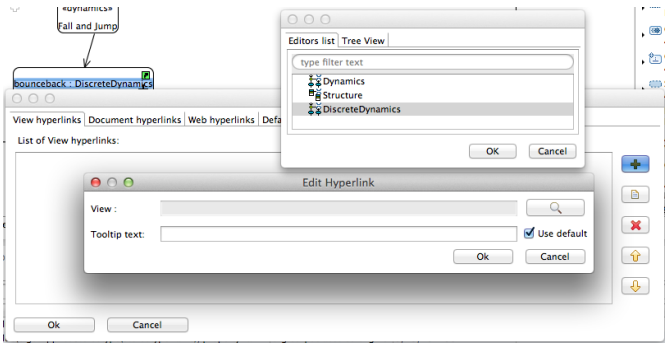


Figure 4.15: Create a new hyperlink

10. Switch to the tab *Default Hyperlinks*, select the *DiscreteDynamics* hyperlink and press the *right-arrow* button to add it as default.

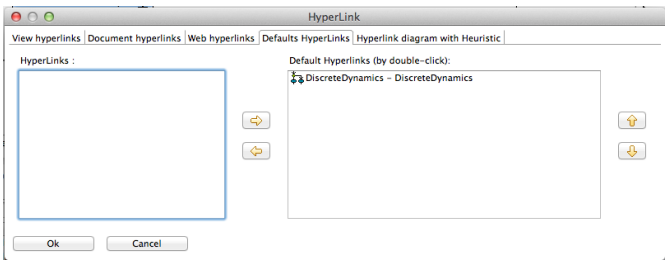


Figure 4.16: Set a hyperlink as default action for double-click on a diagram element

4.4 GENERATE TEXTUAL MODEL

1. Open the context menu of bouncingball.uml in the package explorer.
2. Click *Acceleo Model to Text → Generate DifferentialDynamicLogic*

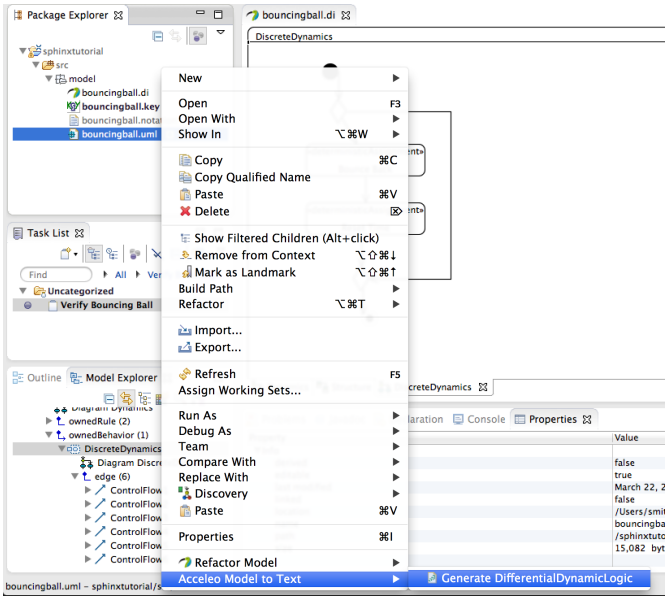


Figure 4.17: Click Acceleo Model to Text → Generate DifferentialDynamicLogic to generate a textual dℒ model from a graphical UML model

5 HYBRID SYSTEM ANALYSIS

Abstract. This chapter introduces the analysis features of Sφnx: Correctness properties about hybrid programs can be formally verified in KeYmaera. In order to develop an intuition about the actual behavior of a hybrid program, Sφnx additionally simulates hybrid programs using Mathematica and displays the traces of variable valuations.

Contents

5.1 PLOT SIMULATED TRACES OF HYBRID PROGRAMS

Sφnx uses Mathematica to create simulated traces of hybrid programs. The simulation traces are displayed in the simulation view, which can be activated by clicking *Window* → *Show View* → *Other...* → *Sφnx* → *Hybrid Simulation*. You can create a simulation run for the model in the active editor by pressing the gears icon in the simulation view. The context menu on the simulation plot lets you select the displayed variables, scale and zoom the plot, save a bitmap image or print the simulation trace.

Figure 5.1: Inspect simulated traces of hybrid programs

5.2 VERIFY A MODEL WITH KEYMAERA

Sφnx uses KeYmaera to prove correctness properties about models in dℒ. You can start KeYmaera from the context menu of a .key or .key.proof file in the package explorer, or from the context menu of a textual editor.

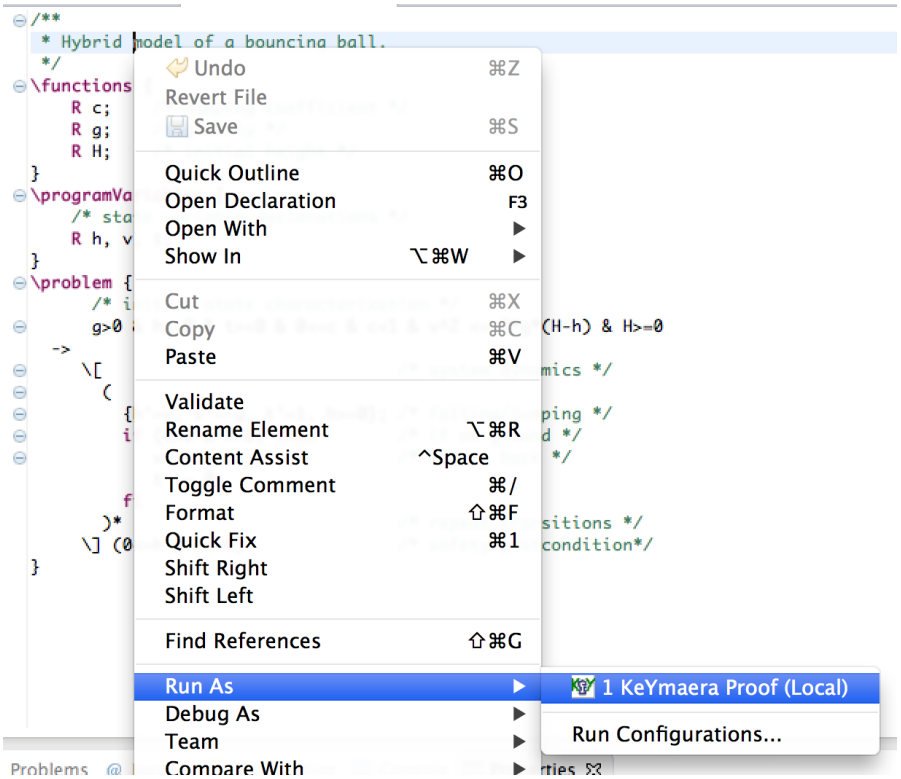


Figure 5.2: Run KeYmaera from the context menu of any .key or .proof file

Sφnx will start KeYmaera as external application and show KeYmaera's information output in a console view. If absolutely necessary, you can stop KeYmaera from the console using the red stop button (not encouraged). Note, that this will force-quit KeYmaera without saving your work.

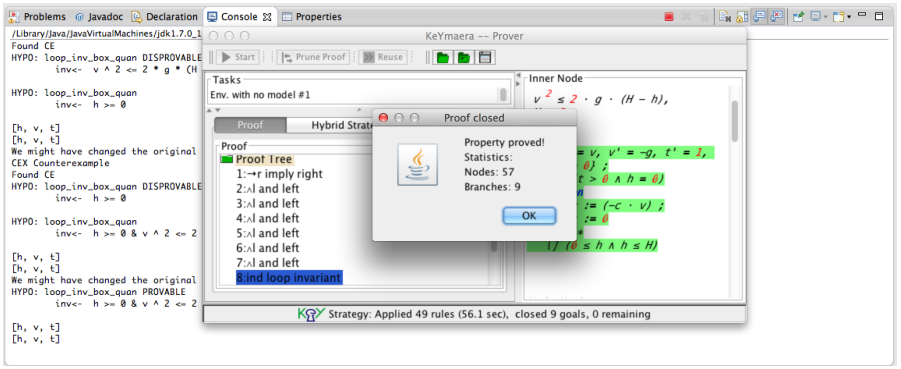


Figure 5.3: The KeYmaera console.

6 PROOF COLLABORATION

Abstract. This chapter introduces the proof collaboration features of Sφnx. Sφnx uses Eclipse for model and proof versioning, and is thus compatible with common versioning systems such as SVN and GIT.

Contents

6.1 SHARE AND COLLABORATE ON TEXTUAL MODELS

Textual models in dℒ, just as ordinary source code, can be versioned in a source code repository. Differences between versions (updates, deletions, and conflicts) can be compared and resolved using the standard Eclipse tools.

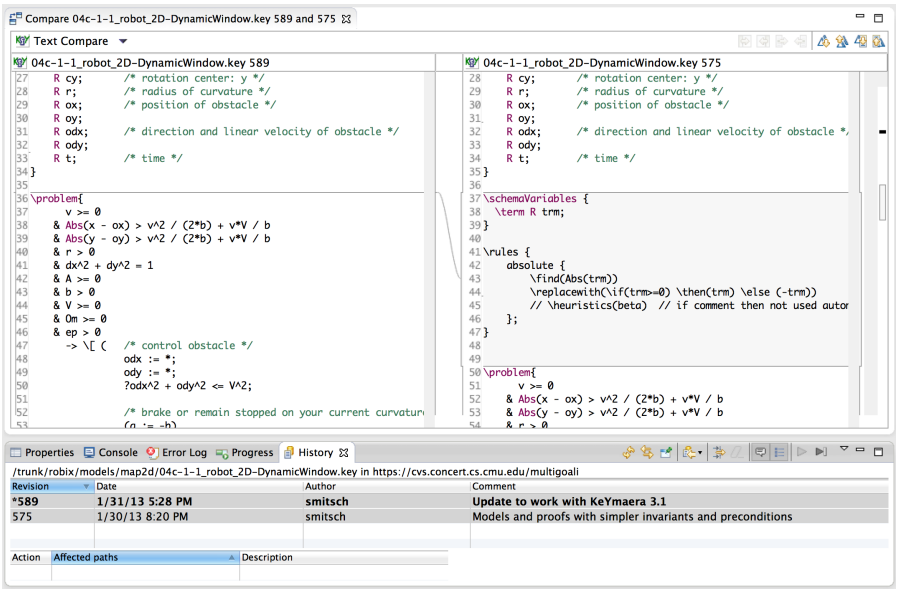


Figure 6.1: Comparison of textual models

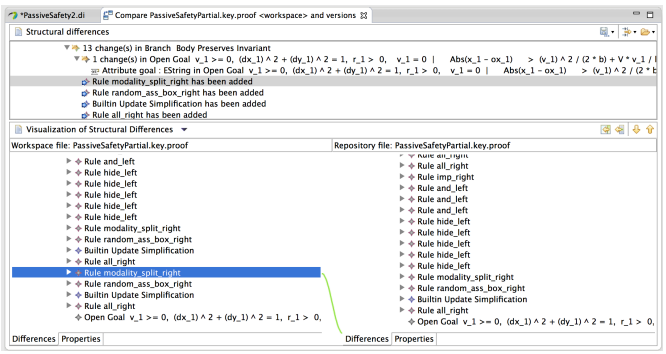
6.2 SHARE AND COLLABORATE ON A PROOF

Just as for textual models, Sφnx supports any source code repository connected to Eclipse to version proofs. You can compare changes between your

local version of a proof and the latest version in the source code repository as follows.

1. Open the context menu of a proof and click *Compare with → Latest from Repository*
2. Click *Complete resource set(s)*
3. Browse the proof comparison

Figure 6.2: Proof comparison

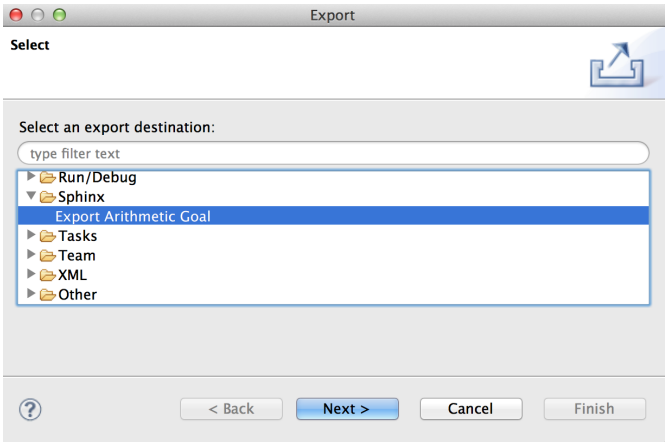


4. Use the buttons in the structural differences view header to copy changes between versions

6.3 EXPORT OPEN GOALS OF PARTIAL PROOFS

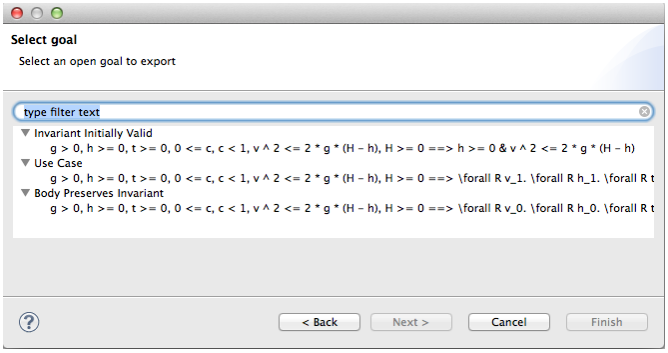
1. Open the context menu of a proof and click *Export...*
2. Select *Sphinx → Export Arithmetic Goal* and click *Next*

Figure 6.3: Export an arithmetic goal via the export wizard



3. Search for and select the open goal to export, click *Next*

Figure 6.4: Search an open goal



4. Select or create a new file to export, click *Finish*.

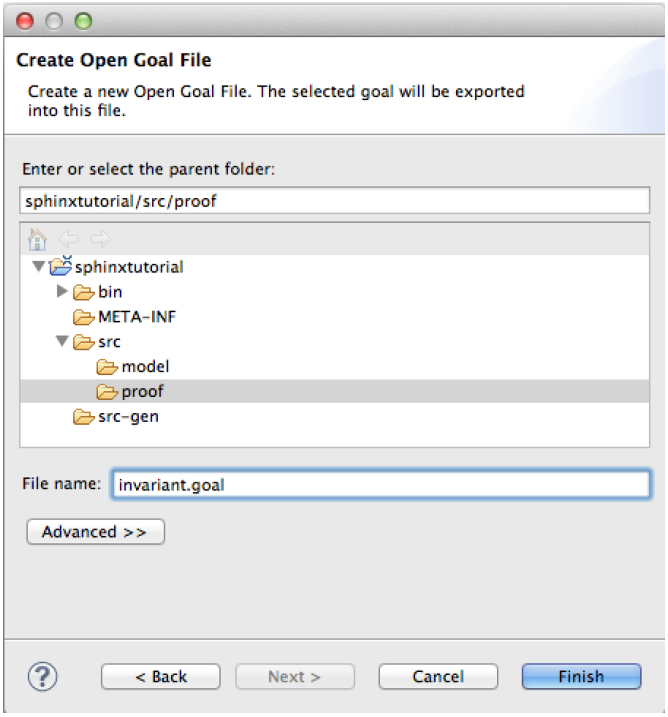


Figure 6.5: Select or create a new file to export

6.4 IMPORT GEOMETRIC RELEVANCE FILTERING RESULTS

1. Open the context menu of a proof and click *Import...*
2. Select *Sphinx* → *Geometric Relevance Filtering Result*
3. Select a file from the file system or from the Eclipse workspace
4. Select the hiding suggestions to import into the proof (usually all) and click *Next*.

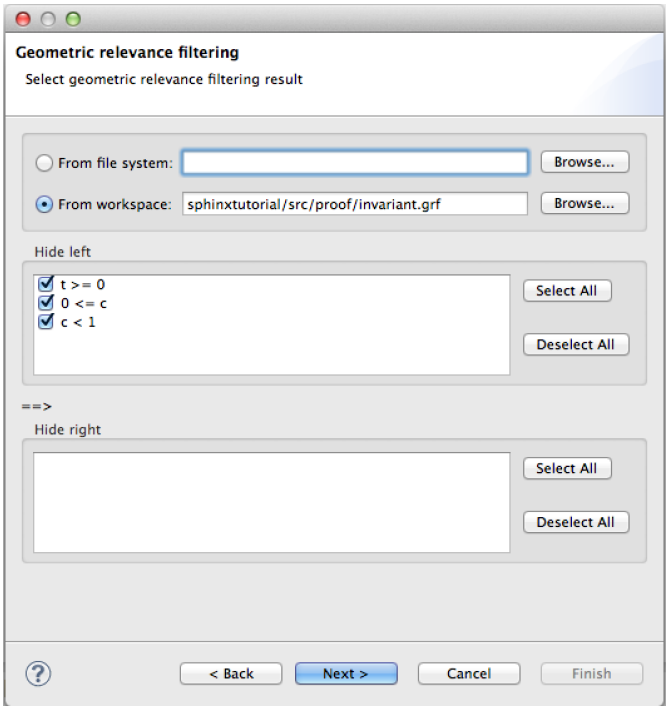


Figure 6.6: Select hiding suggestions

5. Select the applicable goals (usually all) and click *Finish*.

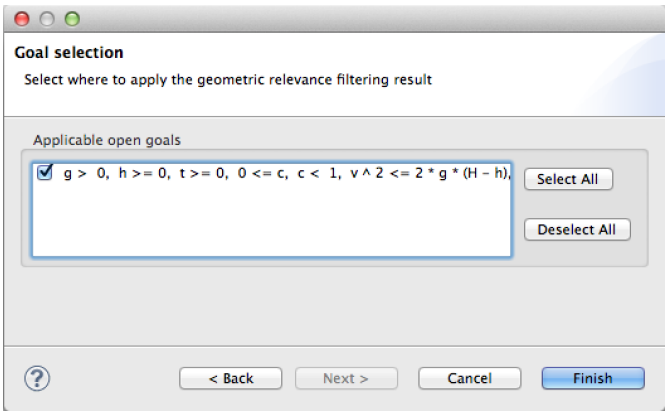


Figure 6.7: Select applicable goals

Commit the changes to the source code repository or load the partial proof in KeYmaera to continue proving.

BIBLIOGRAPHY

[1] Mitsch, S., Ghorbal, K., and Platzer, A. On provably safe obstacle avoidance for autonomous robotic ground vehicles. In *Robotics: Science and Systems* (2013).

[2] Mitsch, S., Loos, S. M., and Platzer, A. Towards formal verification of freeway traffic control. In *ICCPS* (2012), IEEE, pp. 171–180.

[3] Mitsch, S., Passmore, G. O., and Platzer, A. A vision of collaborative verification-driven engineering of hybrid systems. In *Do-Form* (2013), M. Kerber, C. Lange, and C. Rowat, Eds., AISB, pp. 8–17.