

EDDI EDITOR

User Guide

Contents

1. Introduction	3
2. Visual Studio solution structure	4
2.1 ODELib	4
2.2 ODEConverterUnitTests	4
2.3 ODEConverterConsole	4
2.4 ODEConverter	4
3. Test models	7
4. Examples	8
4.1 Importing and converting a HiP-HOPS model	8
4.2 Replacing a subsystem	9
4.3 Importing a state machine (or other failure model)	11
4.4 Adding Events and Actions	12

1. Introduction

The EDDI Editor is a small tool intended to act as glue between different safety analysis tools and runtime EDDI generators. It can:

- Import HiP-HOPS models (both system architectures and analysis results, i.e., FMEA/FTA) and convert them to ODE models
- Import Dymodia state machines and convert them to ODE models
- Load ODE models directly (currently, system architectures, fault trees, FMEAs, and state machines)
- Allow rudimentary editing of most properties of the entities in each model (both before and after conversion)
- Merge ODE models, e.g. by importing a model/subsystem hierarchy to replace an existing (empty) placeholder system, or to add new failure models etc
- Perform some simple validation on the merged models
- (Experimental) Execute a state-based EDDI model to allow testing of events, actions, and triggers etc

Note that it is not intended as a modelling or analysis tool in itself. Models must still be initially created in appropriate modelling tools first. Similarly, some degree of further post-processing is expected to produce an actual runtime EDDI.

2. Visual Studio solution structure

2.1 ODELib

This project contains the 'data' (or model of the MVVM architecture) and performs the actual conversion process. The `ConverterModel` is the parent class of three different model types, representing an ODE model (`ode::`), a HiP-HOPS model (`hip::`), and a Dymodia state machine model (`dym::`), which can be found in the corresponding subdirectories. The `ConverterXtoY` classes (all children of `Converter`) convert from one model type to the other, though `ODEtoHIP` is only a limited prototype. To convert, instantiate the relevant converter, call `Convert()`, and pass it the model to be converted. The ODELib is independent of the other projects, including the GUI, so it can be imported and used directly to convert models.

The actual conversion happens within the internal functions of the `ConverterXtoY` classes, primarily `ConverterHIPtoODE`.

File handling is handled via serialisation for the most part. `hip` and `ode` classes have XML attributes that allow them to be directly serialised/deserialised to their respective formats. Dymodia uses JSON, so the `dym` classes instead make use of JSON.NET to perform loading.

Notes:

- Some ODE members are currently missing. For example, `Hazards` do not include `Measures` and `Ports` do not include `DependabilityRequirements`. `StateMachine` also omits the `TransitionMatrices` and quite a few things are missing from `System`.

2.2 ODEConverterUnitTests

Unit tests for the ODELib classes and converters. Should probably actually write some tests for it...

2.3 ODEConverterConsole

For simple console-only testing of the converter.

2.4 ODEConverter

This is the GUI project, using WPF. Originally it was designed purely for conversion, but it has since grown into a sort of editor with additional features (hence "EDDI Editor"). It uses the Model-View-ViewModel (MVVM) architecture; the `ODELib` serves as the model (data layer), the WPF windows as the View, and the viewmodel layer is in the Viewmodels subdirectory, which echoes the structure of `ODELib`. A `ViewModel` object acts as an interface layer between the data, which should be ignorant of the UI, and the UI itself. This facilitates e.g. conversion of data to a more display-ready form or error checking of user input without adding UI-specific code to the data layer.

Windows include:

- **AddAction.xaml**: a small dialog for adding new Actions to States and Causes.
- **AddEvent.xaml**: a small dialog for adding new Events to Failures.
- **App.xaml**: this is just the top-level application.

- **ImportModel.xaml**: this window opens when importing (rather than opening) a model. Designed to let you view/edit a model before completing the import. Click 'convert' to convert it to ODE (if not already an ODE model) and import it. Remember to change file type if opening Dymodia files.
- **ImportSystem.xaml**: a simple prompt that lets you select a file to import (or two files, if HiP-HOPS). Not used.
- **MainWindow.xaml**: the main interface. Allows you to open/import/edit/merge ODE (EDDI) models.
- **MainWindowOld.xaml**: the old converter interface. The idea was that you import a non-ODE model (e.g. from HiP-HOPS) in one tab, then convert it and view/edit it in the second tab. No longer used, but kept just in case.

Menu commands in the main window:

- **New**: creates an empty ODE model.
- **Open**: opens an existing ODE model (replacing the current model, if any).
- **Import**: opens a new dialog that allows you to import a model, tweak it, and convert it to ODE. You can also open an existing ODE model this way and import it directly (no conversion), though it offers no benefits to do so.
- **Save**: saves the model to the current file (or opens Save As if no current filename).
- **Save As**: enter a new filename and save the model to it.
- **Export As**: in progress, but in theory will allow export to other model types (e.g. convert ODE to HiP-HOPS).
- **Exit**: ...exits.
- **Validate model** (unfinished): performs a limited validation of the model. Will be implemented when I figure out what should be validated (and how).

While in the main window, you can also right-click on any ODE model, system, or failure model to open up another import menu:

- **Import model as subsystem**: opens the import dialog and lets you import a model as a new subsystem under the current (right-clicked) system.
- **Import model and replace this subsystem**: opens the import dialog and lets you import a model to replace the current (right-clicked) system. Used for replacing placeholders/dummy systems with full ones.
- **Import new failure model**: imports a new failure model and adds it to the current system.

Models imported this way can be existing ODE models or models from another tool (which will get converted). Thus you can e.g. open a high-level ODE model from one tool, then add subsystems or replace empty subsystems with e.g. HiP-HOPS models, or import state machines from Dymodia etc.

You can also right-click on any State to add Actions to the entry/exit action lists, or Causes to add generic Actions. This opens up a small dialog to set some basic info for each Action; further (type-specific) information can be added via the properties information in the main window.

Similarly, you can right-click Failures to add Events. A similar dialog opens up to set the name, description, and type. Further information is added via the properties pane.

3. Test models

A small selection of pre-made model files are included for testing and demonstration purposes:

- **KIOS Example**, consisting of a HiP-HOPS analysis and converted ODE files. Analyses a single drone and the base station.
 - `hip_kiosDrone_analysis.xml`: the HiP-HOPS system architecture file for the KIOS model.
 - `hip_kiosDrone_results.xml`: the HiP-HOPS results file for the KIOS model.
 - `kiosDrone.slx`: the KIOS drone Simulink model annotated for HiP-HOPS.
- **Locomotec example** is a HiP-HOPS model for the old Locomotec use case.
 - `hip_locomotec_results`: HiP-HOPS results file for this model.
 - `hip_locomotec_analysis`: HiP-HOPS architecture file for this model.
 - `locomotec.mdl`: the Locomotec Simulink model annotated for HiP-HOPS.
- **Standby Recovery example** is a simple HiP-HOPS example consisting of a standby-recovery block, making it suitable as a state-based example.
 - `hip_standbyRecovery_results.xml`: the HiP-HOPS results file for the SR model.
 - `hip_standbyRecoveryEmptyStandby_analysis.xml`: this is the HiP-HOPS architecture file for a model with a placeholder standby subsystem.
 - `hip_standbyRecoveryFull_analysis.xml`: this is the full HiP-HOPS architecture file for the SR model (no placeholders).
 - `hip_standbyRecoveryOnlyStandby_analysis.xml`: this is the HiP-HOPS architecture file for just a standby subsystem. The idea is to import into the EmptyStandby model and replace the placeholder.
 - `dym_primaryStandby.usm`: the standalone state machine model from Dymodia (for inspection purposes).
 - `dym_primaryStandbySM.uproj`: the Dymodia project file containing the state machine model.
 - `ode_standbyRecoveryFull.xml`: the ODE exported version of the full model (including results).
 - `ode_standbyRecoveryFullWithSM.xml`: the ODE exported version of the full model (including results) with the imported state machine.
 - `ode_standbyRecoveryEmptyStandby.xml`: the ODE exported version of the SR system with a placeholder/dummy standby subsystem (and no results).
 - `ode_standbyRecoveryOnlyStandby.xml`: the ODE exported version of just the standby subsystem (no results).

4. Examples

4.1 Importing and converting a HiP-HOPS model

Click **File** then **Import**. This opens the Import dialog. At the top-left, click **Open** and select a HiP-HOPS file (for this example, use `hip_kiosDrone_Analysis.xml`). The first file is always the main system/model architecture file.

This will load, populating the model hierarchy tree view and the properties pane. The "Detected type" box should say that it is a HiP-HOPS model. You may notice that the Results in the hierarchy view is empty, but if the load was successful there should be hazards and a default perspective with components and failure data etc.

Because HiP-HOPS uses two separate files, one for the system architecture (technically, this is an input file for HiP-HOPS) and one for the results (the output file), we also need to load the results. Click the second **Open** button in the top-left and select `hip_kiosDrone_Results.xml`.

Note that the `outputtype=RESULTS` option should be used with HiP-HOPS to generate a single output XML file rather than one per fault tree.

This will reset the hierarchy and properties but will merge the results with the rest of the model.

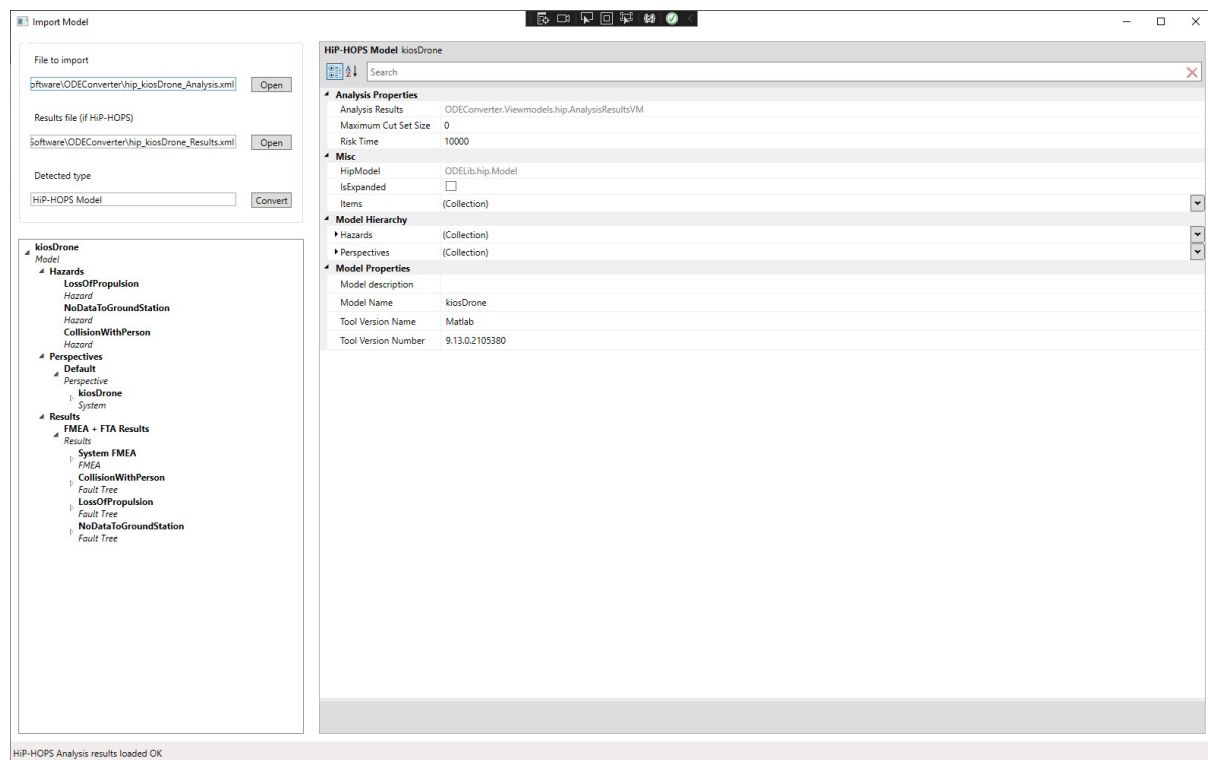


Figure 1: Import dialog

At this stage, you can choose to edit some or all of the properties of the model. Note that editing is restricted to the properties of existing model entities; you cannot change the model structure by adding/removing entities.

Pressing the **Convert** button next to the detected type will convert the imported model to ODE and return to the main view. At this stage, you can again edit the resulting ODE model if you wish, though again you cannot edit the structure, only the properties.

4.2 Replacing a subsystem

From the main window, click **File** then **Open**. Using **Open** rather than **Import** requires you to load an ODE model. Select the `ode_standbyRecoveryEmptyStandby.xml` file. This is a simple primary-standby model with three components: a sensor for input, a primary actuator/processor, and a standby actuator/processor that takes over if the primary fails. The 'empty standby' version has only a placeholder where the standby should be.

This simulates a high-level architecture model where detailed information about subsystem implementation is not available yet, e.g. because the design is still immature or because the subsystem is produced by a third party contractor.

The EDDI Editor allows you to add or replace subsystems and thereby merge models (potentially from different tools).

To demonstrate, use the hierarchy tree view to navigate down to the Standby system:

- Default (model)
 - System Elements
 - Default (system)
 - Subsystems
 - Standby (System)

You should notice that although ports are defined (to establish the interface), there is no failure data defined.

Right click the Standby system in the hierarchy view then select **Import model and replace this subsystem**. This will open up the import dialog once more. This time, select `hip_standbyRecoveryOnlyStandby_analysis.xml`. This is a small model that only defines the standby subsystem (matching the port interface designed in the parent model). Press **Convert** to import, convert, and merge this with the main model.

This should return to the main view, except now the previous placeholder has been replaced by the imported model with failure data. In this case, we have also merged an ODE model with a HiP-HOPS model.

You can also add new subsystems to either the top-level model or any subsystem/component. However, at present the failure models cannot be merged, only added to, which may not be intended behaviour.

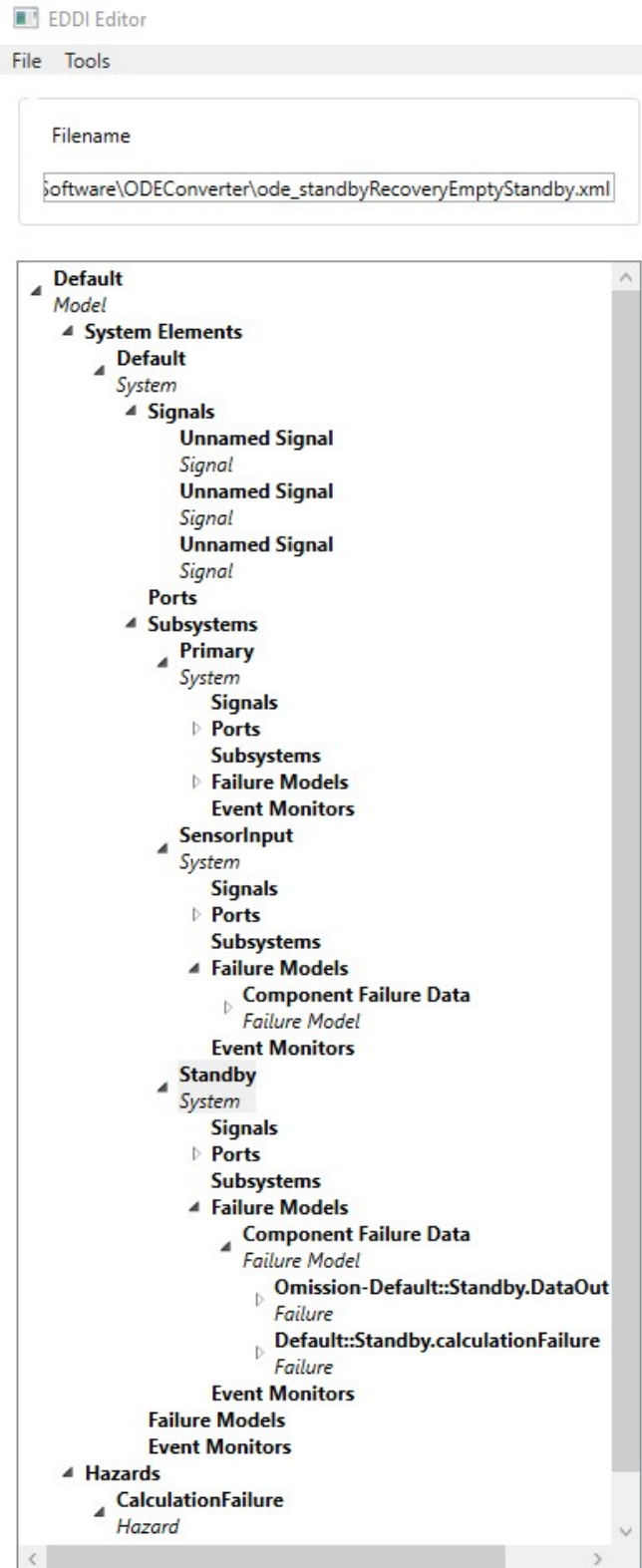


Figure 2: Merged Standby Recovery model

4.3 Importing a state machine (or other failure model)

Just as we can import and merge system architecture models, so can we import failure models such as state machines. Using either the model from the previous example or by loading `ode_standbyRecoveryFull.xml`, navigate to the top (the Default model) and right click. Select **Import New Failure Model**. (You can also do this by right-clicking any subsystem, but using the top-level model will guarantee that we add the failure model at the highest level).

This opens the import dialog as usual. This time, click **Open** and in the drop down menu to the bottom right, select a Dymodia project (.uproj) instead of an XML file. There should be a `dym_primaryStandbySM.uproj` file; import this. You can see how it defines a state machine consisting of three states and three transitions. Press **Convert**.

This will add the state machine to the existing model. In so doing we have also combined models of three different types (ODE, HiP-HOPS, Dymodia).

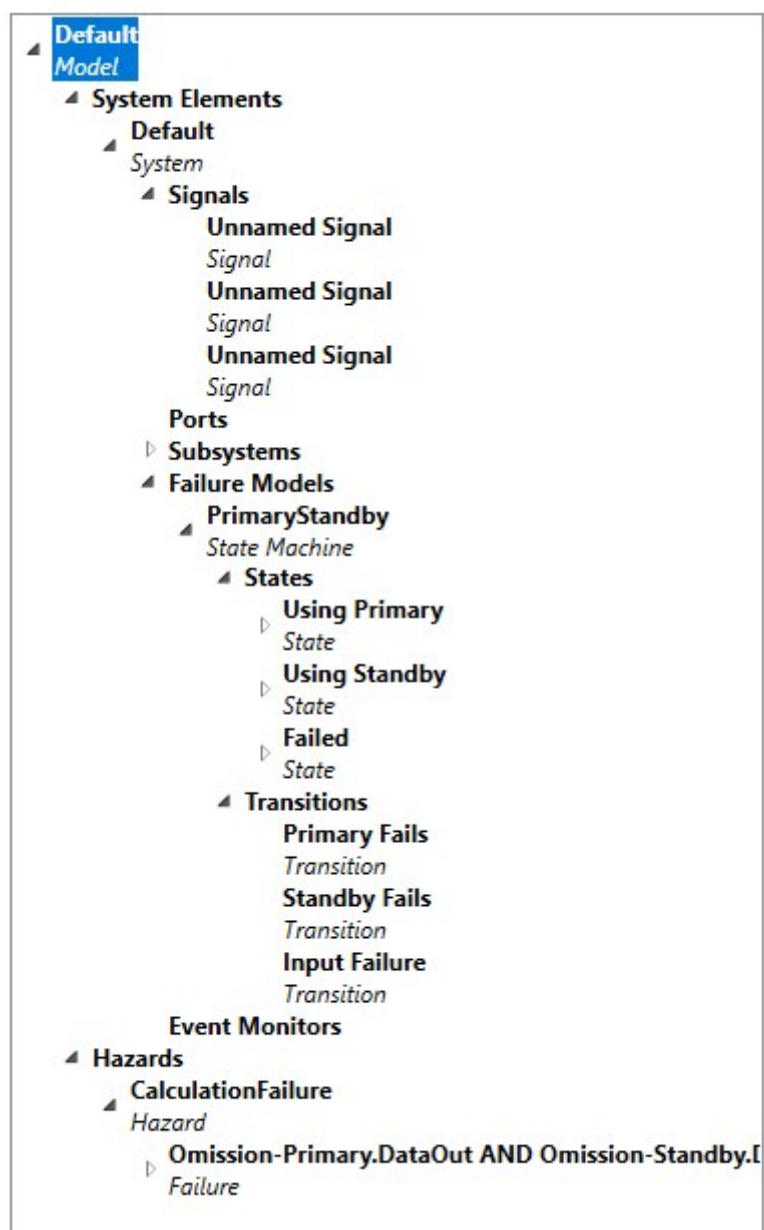


Figure 3: Imported state machine

Type-specific information (like the message in this case) can now be added via the properties pane to the right.

The process for adding Events is very similar. Right-click on any Failure (e.g. `Default::Standby.calculationFailure`) and you can find **Add New Event**. Clicking this opens up another dialog for defining an Event (including its type). Pressing Add Event will add it to the model.

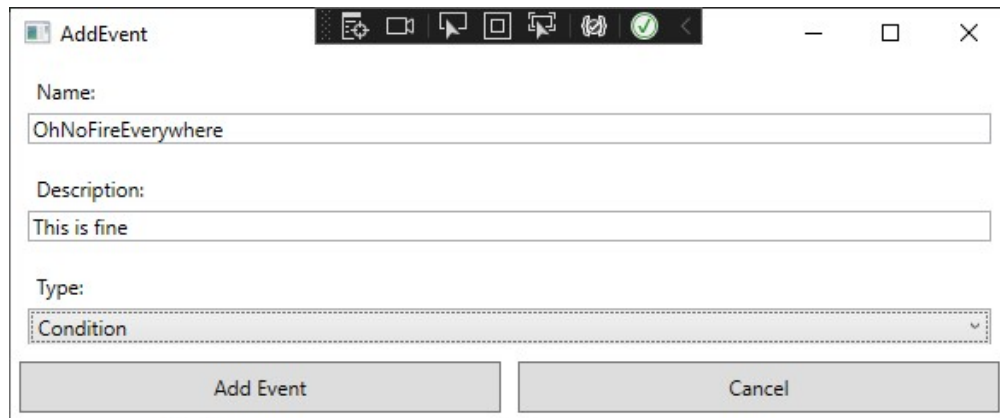
The image shows a software dialog box titled "AddEvent". It has a standard Windows-style title bar with a close button (X) and a maximize button (square). Below the title bar is a toolbar with several icons: a plus sign, a monitor, a speech bubble, a square, a double square, a speech bubble with a plus sign, a green checkmark, and a left arrow. The main area of the dialog contains three text input fields. The first is labeled "Name:" and contains the text "OhNoFireEverywhere". The second is labeled "Description:" and contains the text "This is fine". The third is labeled "Type:" and is a dropdown menu currently showing "Condition". At the bottom of the dialog are two buttons: "Add Event" on the left and "Cancel" on the right.

Figure 6: Add event dialog

Again, type-specific information (like the condition) can be added in the properties pane. To save time, by default the condition is set to the Failure that the Event was added to, but you can easily change this.

