Working with the Model Checker API

Version 4.1 – 02/04/2023

AUTODESK

Getting started

The Model Checker API is designed to provide a simple system for automating the tasks that the full UI version of Model Checker does. It is designed to read and set check sets, local configurations, run checks, and get and save reports.

Model Checker uses XML-based checkset files to store the data needed to run a set of checks and generate reports. These files can be created and modified using the Model Checker Configurator tool. Each Revit model can have a single path to a checkset file stored in its extensible storage. The Model Checker API has a **CheckSet** class that represents a single checkset file. The CheckSet can collect the path to the XML checkset file from the model's extensible storage or simply be a valid path.

Model Checker can also save model specific preferences (such as what items are checked and unchecked, user specific filter values, etc.) without needing to modify the source. To do this, Model Checker uses **RunState** objects to store and retrieve the model specific settings for a check set.

When running checks, a **ReportRun** object will be returned. This object represents a single run of reports that was done and can contain reports for one or more models. The ReportRun has a property that holds a collection of **FileReport** objects; each of these represents all checks that were run on a single Revit file and possibly its links.

Project setup

To use the Model Checker API 2 dll files need to be referenced. They are Revit release specific and can be found in the Model Checker installation folder that corresponds to the Revit release:

1. **BIT.ModelChecker.API.dll** – This contains the core data model and classes that do not require Revit interaction directly. The base namespace is **ADSK.BIT.ModelChecker.API** and under that you will find basic data model and most service classes.
2. **BIT.ModelChecker.Revit.API.dll** – This contains all API classes that require direct Revit access. You will need to reference the version built to the Revit version you wish to target. The base namespace is **ADSK.BIT.ModelChecker.Revit.API** and under that you will find basic data model and most service classes.

Configuration

Storing the checkset path

A model can store the path to the XML checkset file in its extensible storage. The API manages this by creation a **CheckSetLocationRepository** for the document and then storing the path. Paths can be valid URL, drive letter-based, or UNC path. The code for this would look as follows:

```
var locationRepository = new CheckSetLocationRepository(doc);
locationRepository.SaveCheckSetLocation("http://mycompany.com/checksets/
someCheckset.xml");
```

Note that a transaction should NOT be created for this; Model Checker will create its own transaction.

Getting a CheckSet

To run a check, there needs to be a **CheckSet** object which the Model Checker API can successfully load from a disk or web-based path. To obtain the CheckSet object, create a **CheckSetService** instance and call **GetCheckSet**, passing in the path of the check set you wish to load. This will return the CheckSet object. Note that you will also need to construct and pass a **PrebuiltOptionsService** to the **CheckSetService** constructor. This is a simple parameterless constructor.

If the path to the checkset is known and not store in the Revit model's extensible storage, this is sufficient. To obtain the location of the checkset stored in the model, an instance **CheckSetLocationRepository** must be created, using the document you wish to call from. Then call **GetCurrentCheckSetLocation()** can be called to obtain the current stored location. This may be null or empty if a path has not been saved in the model.

Obtaining a checkset file that is not stored in the model's extensible storage by the Model Checker:

```
CheckSet GetCurrentCheckSet(Document doc)
{
    var checkSetPath =
    "http://mycompany.com/checksets/someCheckset.xml";

    var optionsService = new PreBuildOptionsService();
    var service = new CheckSetService(optionsService);
    return service.GetCheckSet(checkSetPath);
}
```

Getting the checkset location from the model and retrieving the check set object:

```
CheckSet GetCurrentCheckSet(Document doc)
{
    var locationRepository = new CheckSetLocationRepository(doc);
    var checkSetPath = locationRepository.GetCurrentCheckSetLocation();
```

```
    if (string.IsNullOrEmpty(checkSetPath)) return null;

    var optionsService = new PreBuildOptionsService();
    var service = new CheckSetService(optionsService);
    return service.GetCheckSet(checkSetPath);
}
```

Using the RunState

If the desire is to use the checkset in its default state stored in the XML, simply creating the CheckSet object successfully is sufficient.

However, the Model Checker can save a local **RunState** object to the model which stores the user's settings for this checkset. This may be specific sections or individual checks that have been turned off, or user input values for specific checks. To update the check set to reflect this information, a RunState object must be created and applied to the CheckSet.

To obtain the run state construct a **RunStateRepository** for the document and call **GetRunState().** This will give you the run state object. If no RunState is stored, null is returned. Then use the static 'Utility' class to update the CheckSet with the values from the RunState:

```
CheckSet GetCurrentCheckSet(Document doc)
{
    var locationRepository = new CheckSetLocationRepository(doc);
    var checkSetPath = locationRepository.GetCurrentCheckSetLocation();

    if (string.IsNullOrEmpty(checkSetPath)) return null;

    var optionsService = new PreBuildOptionsService();
    var service = new CheckSetService(optionsService);
    var checkSet = service.GetCheckSet(checkSetPath);

    var runStateRepository = new RunStateRepository(doc);
    var runState = runStateRepository.GetRunState();

    if (runState != null)
    {
        Utility.SetCheckSetFromRunState(checkSet, runState);
    }

    return checkSet;
}
```

Running checks

Once a valid CheckSet object is created, the Model Checker API can run checks against a Revit model. Running checks can be done in one of two ways, model by model, or in batch. Both result in a **ReportRun** object that collects the results from the check(s) run against the Revit files.

Batch checks

To mirror the functionality of the Model Checker UI, the BatchCheckRunner can be used to run the same CheckSet against a collection of Revit models. This takes a simple set of inputs and returns you a result object for the entire run as well as a list of errors encountered (if any).

To construct a **BatchCheckRunner** object; you will need to pass in the Revit application object.

Second, create a collection of **FileCheckSetting** objects; these are a simple structure that stores the file path and a Boolean value indicating if linked models should be checked as well.

Then call the **RunChecks** method on the check runner. Since error handling needs to be automated for running checks against a potentially large list of Revit models, the Model Checker will collect exceptions and RunChecks will return a List of them.

The **ProgressChanged** event can be used to be notified of progress updates in the run.

A batch run would look as follows:

```
ReportRun RunChecksAndGetReport(CheckSet checkSet, Application revitApp)
{
    var optionsService = new PreBuildOptionsService();
    var checkRunner = new BatchCheckRunner(revitApp);

    var files = new List<FileCheckSetting>()
    {
        new FileCheckSetting(@"C:\Some\Path\model.rvt") {CheckLinks =
        false}
    };

    checkRunner.ProgressChanged += progressData =>
    {
        // Do something to report progress
    };

    var run = checkRunner.RunChecks(files, checkSet, out var errors);

    if (errors.Any())
    {
        // Do some error handling
```

```
    }

    return run;
}
```

Document checks

Alternately, individual documents can be checked. This requires that the model be open but may afford more control.

To do a document check construct a **DocumentCheckRunner** object with the document you wish to check. Then call RunChecks() with the proper arguments. Note that there are some optional arguments on this method that allows the check to be added to an existing ReportRun object.

Running a document check would look as follows:

```
ReportRun RunChecksAndGetReport(CheckSet checkSet, Document doc)
{
    var optionsService = new PreBuildOptionsService();
    var checkRunner = new DocumentCheckRunner(doc);

    checkRunner.ProgressChanged += progressData =>
    {
        // Do something to report progress
    };

    var run = checkRunner.RunChecks(false, checkSet);

    return run;
}
```

Saving and Exporting Results

Once a ReportRun object has been created, the data needs to be saved in some way. There are three ways to deal with the ReportRun:

1. Save to the model
2. Export to HTML
3. Export to Excel

Save to the model

This will save the report data in the model's extensible storage and it will be available to the user as their "Last Run Report" accessible through the Model Checker UI. Saving this does not require a transaction as Model Checker will create its own transaction. However, the model will need to be saved before closing it or this data will not be saved. Note that it is not recommended to save a ReportRun that holds reports for multiple models to a single model this way.
Saving the report to the model is done through the **ReportRunRepository** object; code for this option is as follows:

```
void SaveRunResult(ReportRun result, Document doc)
{
    var optionsService = new PreBuildOptionsService();
    var CheckSetService = new CheckSetService(optionsService);
    var repository = new ReportRunRepository(doc, checkSetService);

    repository.SaveRun(result);
}
```

Export to HTML

Results can be exported to an HTML file for archiving or analysis. To do so construct a **ResultExporterHtml** object and export with the required options:

- **ExportLocation** is a user editable folder path. The Model Checker will create the folder if needed, and if possible. In this folder a single or collection of HTML report files are saved.
- **ExportLists** corresponds to the option in the Model Checker UI to 'Export list elements' that will save the individual Revit elements that match to the checks.

```
void ExportRunResult(ReportRun result)
{
    var exporter = new ResultExporterHtml();
    var options = new ExportOptions()
    {
        ExportLocation = @"C:\Some\Export\Folder",
        ExportLists = true
    };
    exporter.ExportReport(result, options);
}
```

Export to Excel

Like HTML, results can be exported to Excel using the **ResultExporterExcel** class. The required options are:

- **ExportLocation** is the path to an Excel file. The Model Checker will create the file if needed, and if possible. It will overwrite an Excel file that matches the name. The Model Checker API will not confirm the file extension. It will throw an exception if the ExportLocation is a folder path such as “C:\temp\myreport\” but it will write a file if the ExportLocation could be identified as a file such as “C:\temp\ myreport”.
- **ExportLists** corresponds to the option in the Model Checker UI to ‘Export list elements’ that will save the individual Revit elements that match to the checks.

```
void ExportRunResult(ReportRun result)
{
    var exporter = new ResultExporterExcel();
    var options = new ExportOptions()
    {
        ExportLocation = @"C:\Some\Export\Folder\myfile.xlsx",
        ExportLists = true
    };
    exporter.ExportReport(result, options);
}
```