# Milestone

## 发布 *1.0.0*

**MA Yuhai**

**2020 年 06 月 12 日**

# Contents

# 1 技术背景

## 1.1 仿真的基本概念

本文档中，仿真是指用计算机计算，对物理过程进行模拟的方法。通过计算机仿真的手段，能够对被设计产品的数学模型进行模拟研究，估计预期的动态性能，实现基于模型的设计。

首先，需要对被模拟对象的动态物理过程进行数学建模（简称建模），获得简化的数学模型，如常微分方程（ordinary differential equation，ODE），或微分代数方程（differential algebra equation，DAE）等形式。然后，复杂的数学模型表现为多变量的方程组，难以直接给出各变量随时间变化的解析形式，往往需要借助计算机上的通用数值算法，对数学模型进行数值求解（简称求解）。

求解方面，已经有众多成熟的仿真软件工具，将通用数值算法形成模型求解器（solver），如 MATLAB/Simulink、LabVIEW、ADAMS、AMESim、SimulationX、Silver 等等。

建模方面，随着被研究对象数学模型的复杂度增加，为便于开发、测试与功能复用，将其进行模块化分割成为必须的手段。

反过来，对模块化的部件模型进行接口的连接，才能形成代表被模拟对象的系统仿真模型，称该过程为模型集成（简称集成）。

前面提到的仿真软件工具，一些兼具建模功能，一些仅对模型进行集成，多数软件则兼具建模和集成功能。在对航空、航天器，汽车等复杂的被模拟对象进行数学建模时，其往往包含力学、热学、电学、自动控制和软件等不同领域的设计特征，是一个多学科交叉系统；不同仿真软件所针对的物理领域往往不同，单一软件难以方便地实现多学科交叉系统的建模。因而，需要在不同仿真软件的模块化部件模型集成方面，形成可交换的标准接口，使得在不同仿真软件内建立的模型可以相互导入、导出，实现对多源异构模型的集成和系统仿真。

## 1.2 本系统着力解决的问题

目前，主要的模型标准接口为 S-Function 与 Functional Mock-up Interface, FMI。S-Function 为 Simulink 的模型开发接口，应用较为广泛；对于集成已有的 C/C++ 代码，具有 Legacy Code Tool, LCT 代码导入工具以及图形化的向导工具 S-Function Builder；但由于 Simulink 是商业工具，其他工具仅支持 S-Function 模型的导出，S-Function 模型的导入和求解仅能在 Simulink 中进行。

FMI 是一种还在发展之中的开放接口标准，由国际性的产业联盟维护，已在汽车行业广泛支持。符合 FMI 接口标准的模型为 Functional Mock-up Unit, FMU，其模型端（model slave）与求解端（solver master）的实现都有开源软件实现范例，因而越来越多的工具支持 FMU 模型的导入、导出以及系统集成

仿真。但是，FMI 标准的技术细节多，相关工具链自动化程度低，对于技术人员的编程水平要求较高；对于集成已有的 C/C++ 代码，需要熟悉 C 代码模板的运行流程，定义一维展开的接口变量，完成与已有模型接口数据结构的相互转换，编制 FMU 描述文件（modelDescription.xml），编译各运行平台下的动态链接库，FMU 目录结构的创建，FMU 的打包及测试等等工作。一般用户难以实现复杂的功能，使用的便利性不足。

针对使用 FMI 接口标准集成已有的 C/C++ 代码自动化程度低，开发门槛高等问题，对 FMI 及 S-Function 模型接口标准的运行流程进行提炼和抽象，在 FMI 接口代码模板基础上进行扩展，提供外部资源和定时任务的统一管理接口，形成通用的中间层（图 1.1）；开发图形向导式的接口适配工具，能够自动完成已有模型接口数据结构与 FMI 中一维展开的接口变量间的相互转换，自动生成模型描述文件，并能够一次编码同时支持 FMI 及 S-Function 模型接口标准。

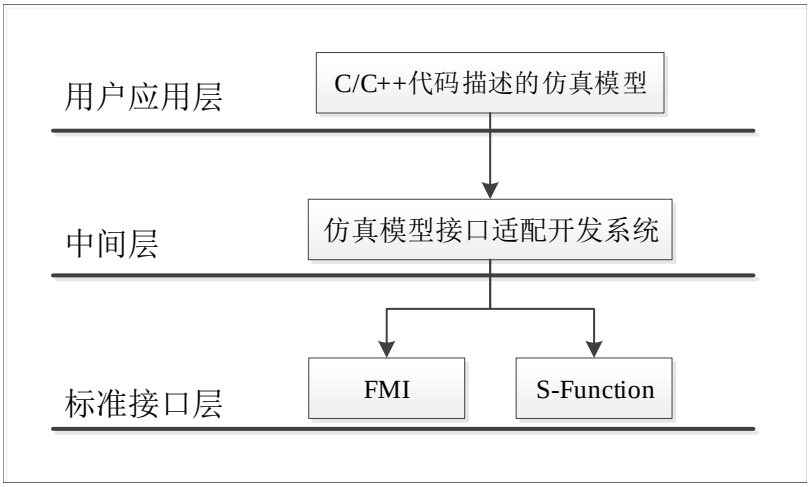


图 1.1: 系统功能定位

## 1.3 主要功能特性

1. 用于模型在环仿真（MiL）的 C/C++ 代码集成工具
2. 简捷、统一，一次编码同时支持 FMI 及 S-Function 模型接口标准
3. 对外部资源及定时任务提供了自动化的管理接口，便于模型中引用
4. 使用 CMake 构建系统，可自动适配大多数编译器
5. 使用 Qt 图形界面，可运行于大部分操作系统的桌面环境
6. 与来自 IBK 的 MasterSimulator 环境集成发布，便于 FMU 集成仿真
7. 生成平台相关的代码，兼容 Windows 及 Linux
8. 生成的模型可运行在基于 Linux 的半实物仿真（HiL）系统

对 FMI 及 S-Function 模型接口标准的运行流程进行提炼和抽象，在 FMI 接口代码模板基础上进行扩展，形成通用的中间层；开发了向导式图形界面，能够辅助用户建立中间层接口函数模板，包括实例化、初始化，步进，重置以及终止，还扩展了外部资源和定时任务的统一管理接口，便于用户在模型代码中引用；开发了接口解析及代码生成工具，能够递归地解析用户模型定义文件中的接口数据定义，建立已有模型接口数据结构与 FMI 中所需的一维展开的接口变量间的映射关系；用户只需一次编写模型定义及实现代码，能够同时支持 FMI 及 S-Function 模型接口标准，自动生成 FMU 模型对应的 FMI 接口代码文件以及 FMU 描述文件，同样的信息也用于自动生成 S-Function 模型对应的 LCT 接口代码文件以及 LCT 模型定义文件。

# 2 安装与配置

## 2.1 软件授权

提供运行系统的 MAC 地址，联系 `WeChat:latitude_vocal` 获取授权文件。

### 2.1.1 Windows

Windows 下查看本机 MAC 地址的方式：查看本地网络连接中的适配器属性，或者在命令提示符窗口（组合键 WIN+R，键入 cmd）中运行 `ipconfig /all` 。

### 2.1.2 Linux

在 X Window 配置界面中查看网络适配器属性，或在控制台窗口中运行 `ifconfig` 。

## 2.2 软件安装

工具包解压缩得到主目录结构。将授权文件放置在 license 目录中。

## 2.3 目录结构

| 目录/文件 | 内容 |
|---|---|
| bin | 可执行文件及脚本，请勿修改 |
| build | 模型构建过程中的临时目录 |
| env | 所依赖的软件环境安装程序 |
| export | 导出的 FMU 及测试工程目录 |
| include | 模型模板的头文件目录，请勿修改 |
| license | 授权文件的放置目录 |
| model | 模型代码的存放目录 |
| CMakeLists.txt | CMake 工程文件模板，界面程序运行时请勿修改 |
| SFcnLists.m | S-Fcuntion 构建脚本文件，界面程序运行时请勿修改 |
| Readme.txt | 简要的使用说明文件 |

## 2.4 外部依赖环境配置

### 2.4.1 编译器

#### 2.4.1.1 Windows

推荐 Visual Studio 中的 cl 编译器，注意不要使用绿色安装。示例代码在 VS2010 及以上版本中经过测试，但推荐使用 VS2013 及以上的版本，以支持 C99 中的编码习惯。

#### 2.4.1.2 Linux

推荐使用 gcc/g++ 或 clang/clang++ 编译器，推荐在系统的包管理器中安装。在 Linux 下使用图形界面需要配置 Qt 运行环境，若系统的 Qt 环境不满足要求，可单独安装 Qt 开发环境，并在运行 GUI 程序前设置环境变量：export LD_LIBRARY_PATH=/home/user/Qt5.12.5/5.12.5/gcc_64/lib/（应替换为用户本地的安装路径）然后，执行 GUI 程序./MasterSimulatorUI

### 2.4.2 CMake

开发工具包执行需要部署 CMake 运行环境。此外 Windows 下还要部署 VC++ 运行时环境。完整版工具包的 env 中包含相应的安装文件。Linux 下，需要授予 bin 目录下程序执行权限（chmod 777 ./bin/*）。

### 2.4.3 路径配置

在 MasterSim 中选择当前系统中安装的 CMake 路径以及 milestone 可执行文件的路径，如 图 2.1 。

### 2.4.4 切换界面语言

在 MasterSim 中切换 Milestone 的界面语言，重新启动后生效，如 图 2.2 。

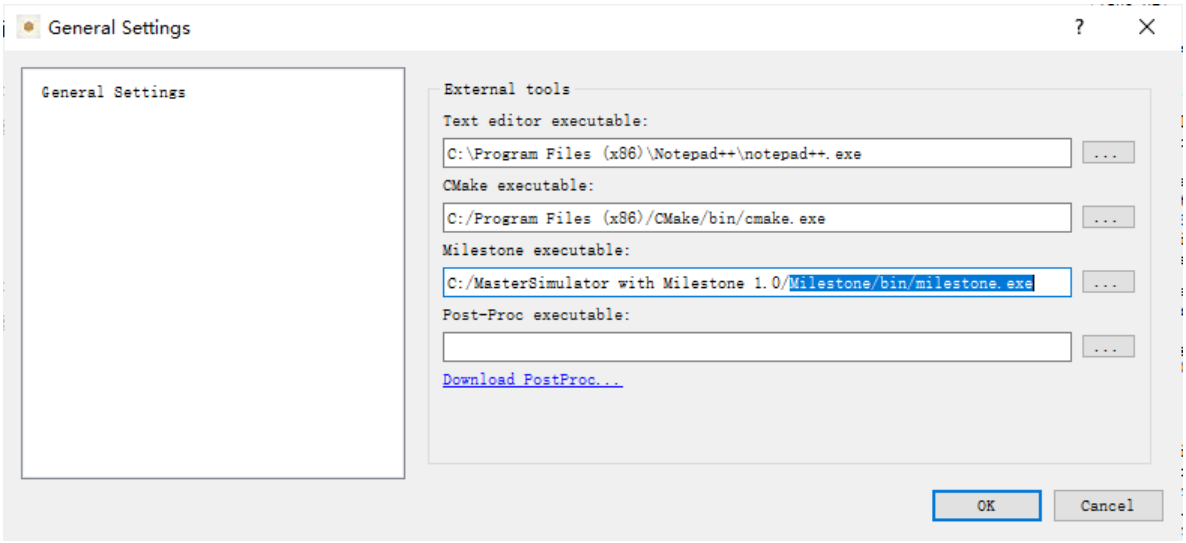


图 2.1: 配置 Milestone 相关工具路径

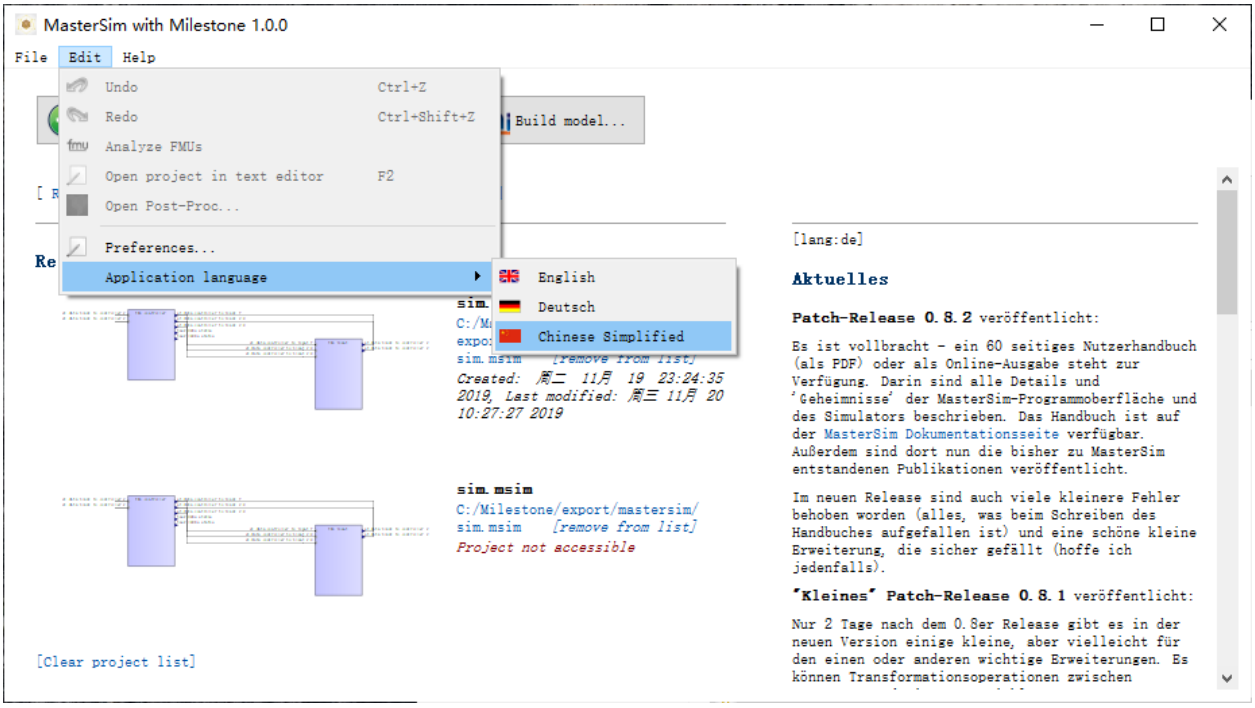


图 2.2: 设置 Milestone 界面语言

# 3 示例教程

## 3.1 运行示例

在了解仿真模型实现的细节之前，可以使用工具包自带的测试用例验证环境部署的正确性，我们先在 GUI 中快速完成这些操作。

- 运行 MasterSimulatorUI.exe，启动 MasterSim 主界面，如 图 3.1 。


图 3.1: MasterSim 主界面

- 点击 “*Build model*”，打开 Milestone 工具主界面，如 图 3.2 。
- 此处我们直接运行示例模型，切换至 “代码生成” 标签页，点击 “创建工程”，如 图 3.3 。
- 弹出模型选择对话框，在模型列表中选择 controller 和 plant 模型，如 图 3.4 。
- 点击 “创建工程”，完成编译工程的创建，如 图 3.5 。
- 点击 “构建模型”，调用系统的编译环境，如 图 3.6 。
- 在工具包的 export 目录中，查看生成的 FMU 文件，如 图 3.7 。

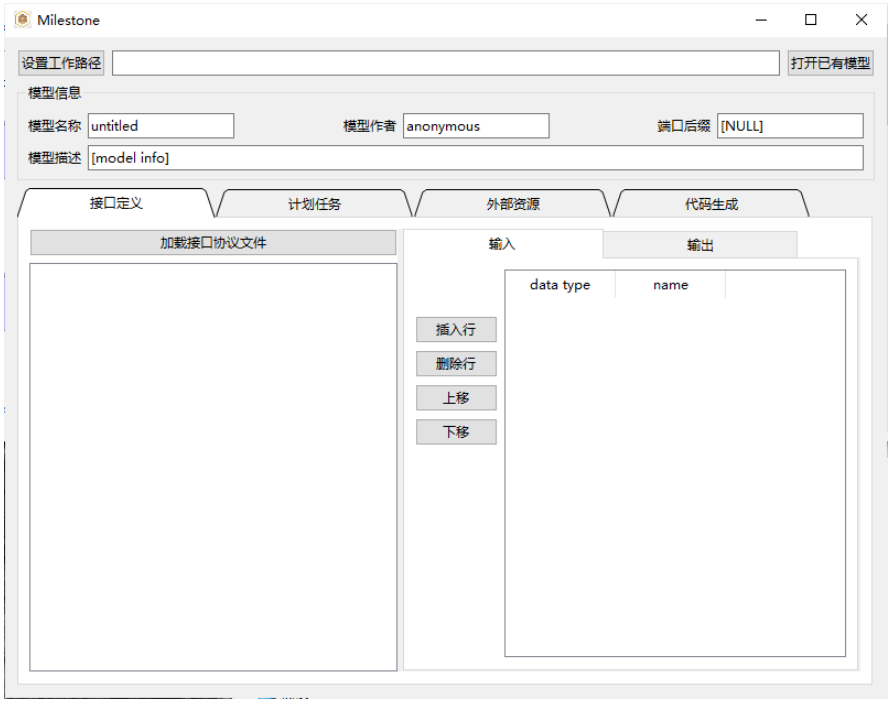


图 3.2: Milestone 主界面

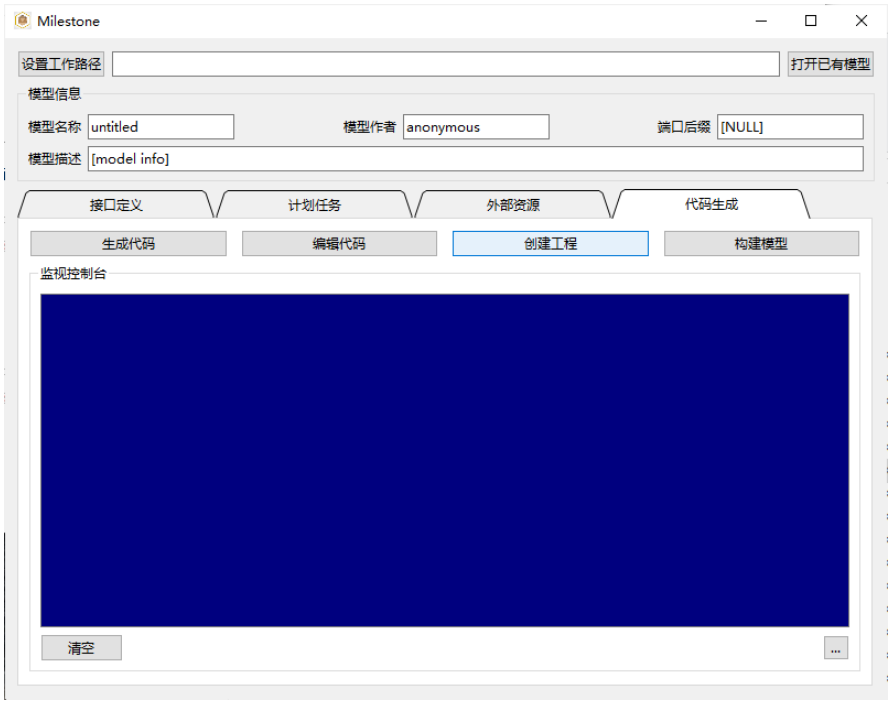


图 3.3: 创建工程

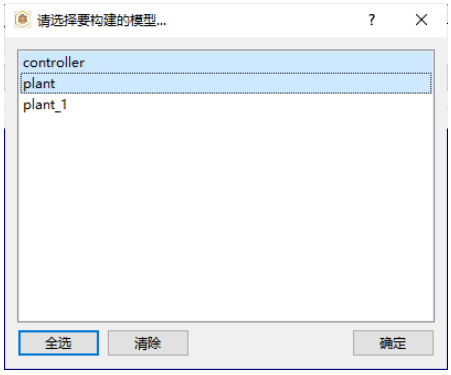


图 3.4: 选择模型

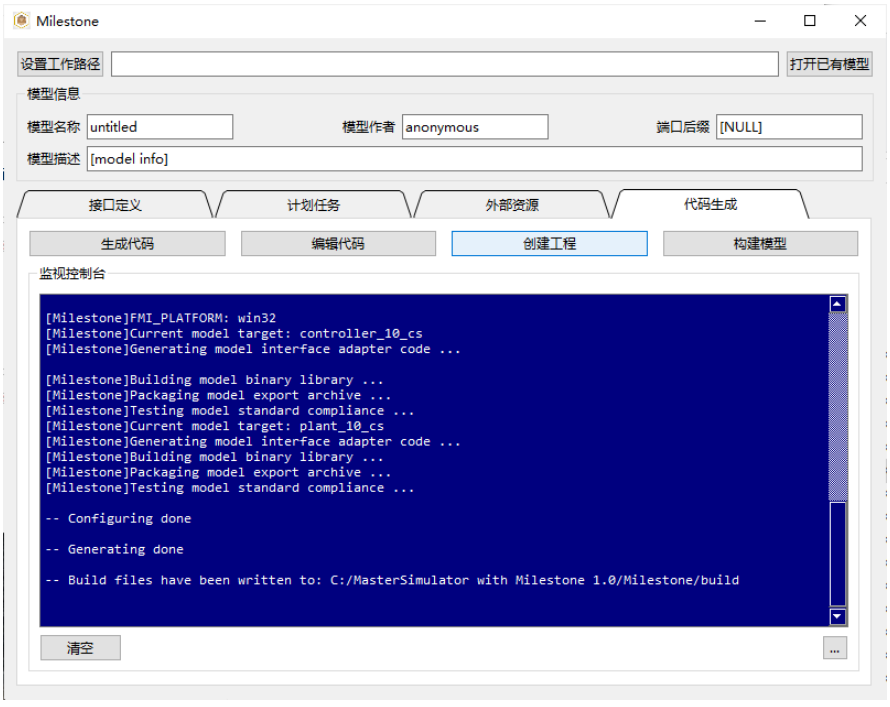


图 3.5: 完成编译工程创建

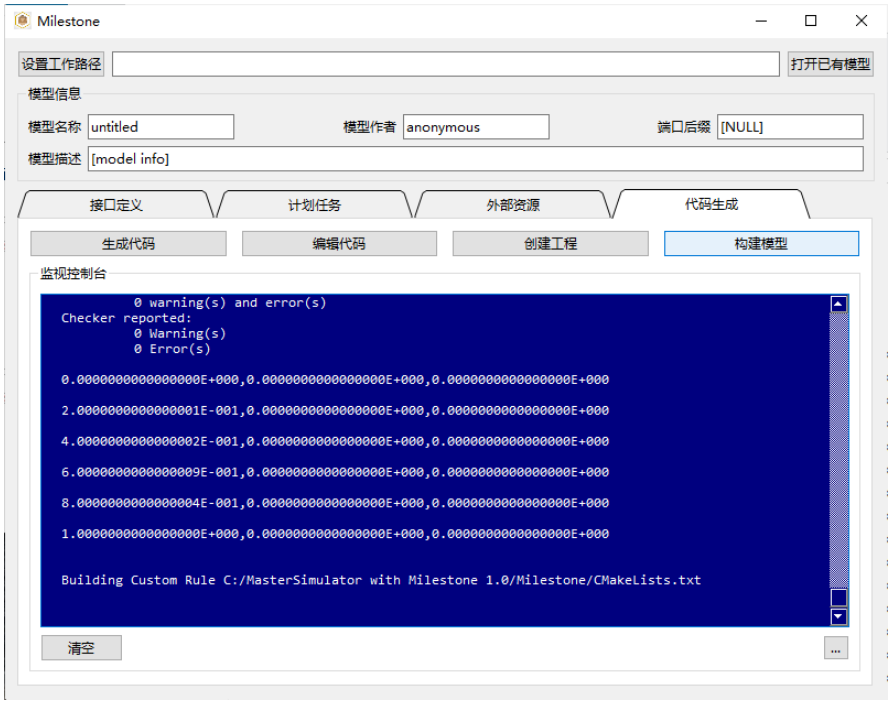


图 3.6: 构建模型


图 3.7: 查看导出的 FMU

- 在 MasterSim 主界面中打开自带的测试工程，如 图 3.8 。

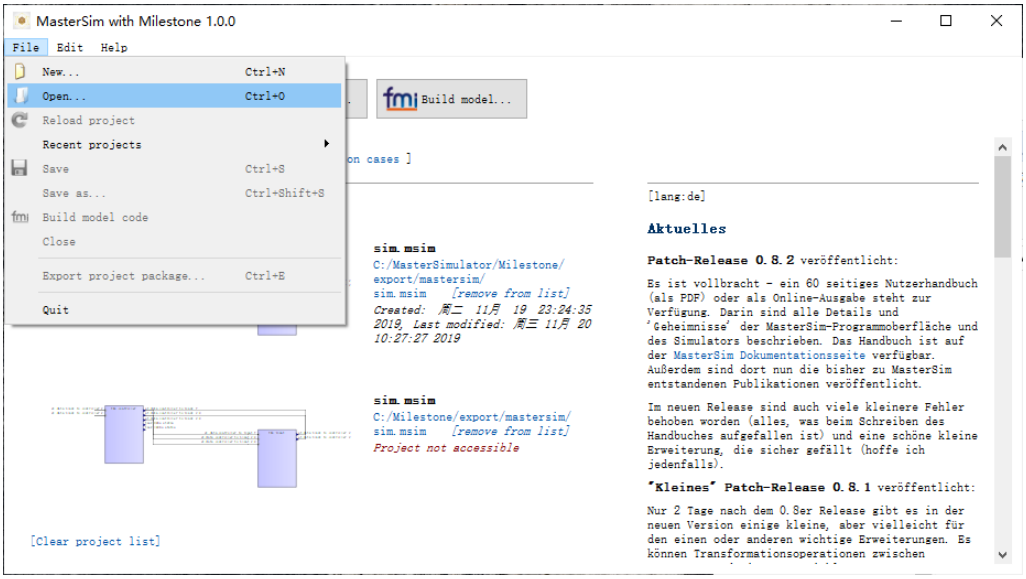


图 3.8: 打开 MasterSim 测试工程

- 测试工程在 export/mastersim 路径下，如 图 3.9 。

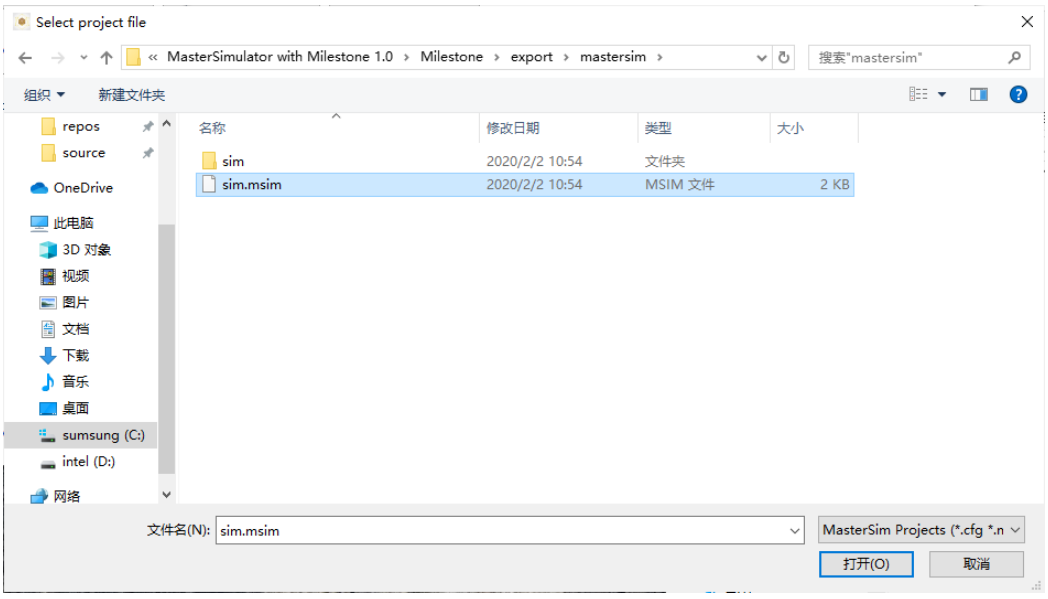


图 3.9: 测试工程默认路径

- 我们这里重新生成了其中的 FMU 模型，但他们的连接保持测试工程中的关系不变，如 图 3.10 。
- MasterSim 左侧的功能选择按钮可以启动 Milestone，测试 FMU 中的信息，启动后处理程序，以图或表格的方式配置模型间的连接，以及配置仿真求解器参数等，请参考其 官方文档[1] 获得更详细的信息。此处可直接使用示例中配置好的参数，点击“开始仿真”按钮，如 图 3.11 。
- 观察仿真器的监控信息，以及打印模型中输出的信息，如 图 3.12 。

[1] https://bauklimatik-dresden.de/mastersim/html_en/MasterSim_manual.html

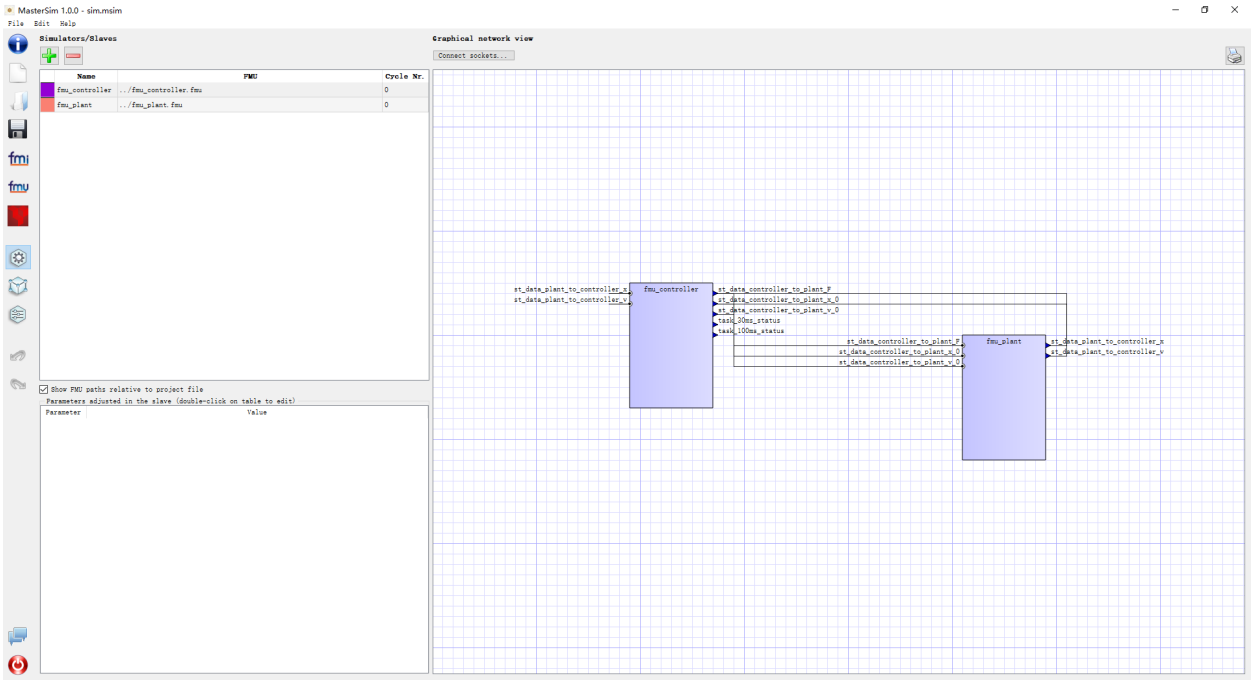


图 3.10: MasterSim 中的模型连接

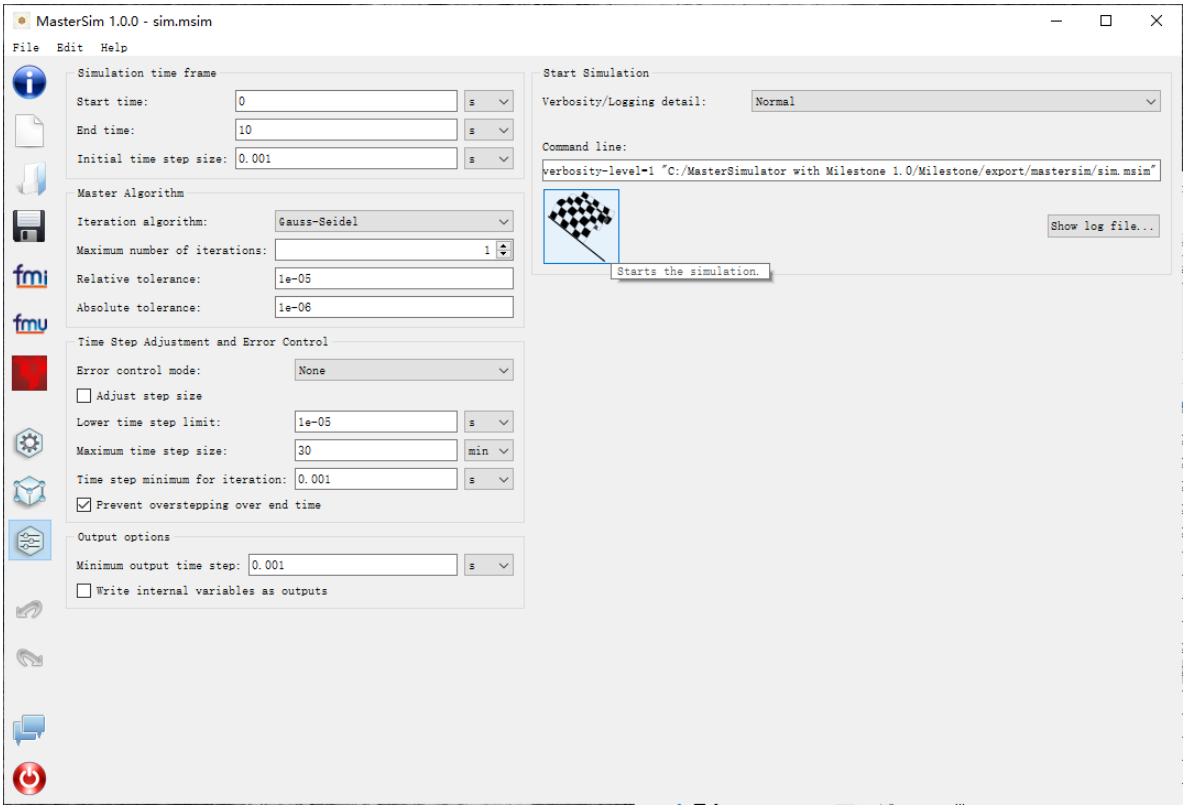


图 3.11: MasterSim 中的仿真配置

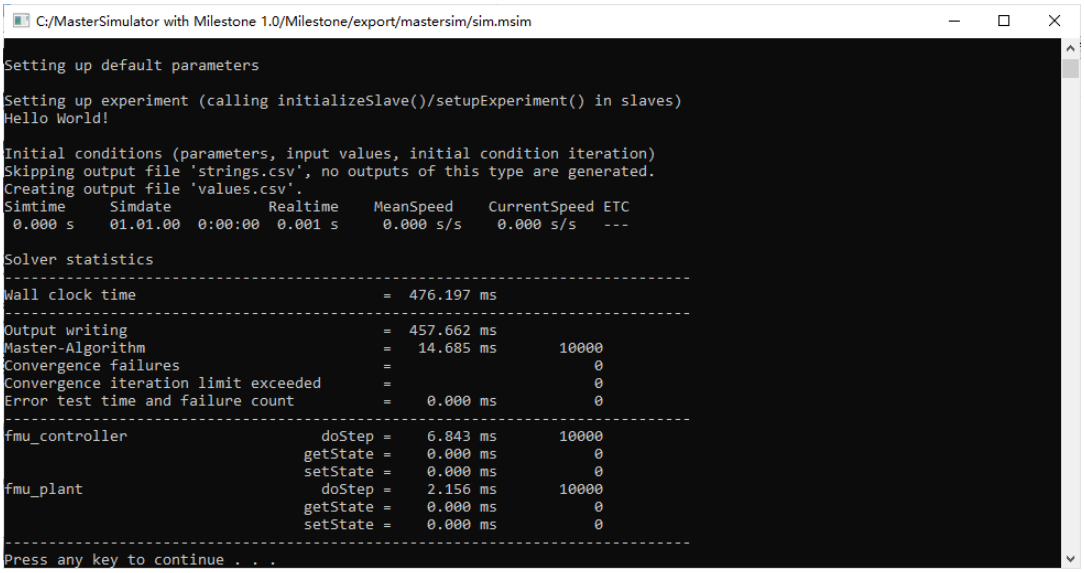


图 3.12: MasterSim 仿真过程监控

- 生成的结果数据文件存放在 results 路径下，如 图 3.13 。

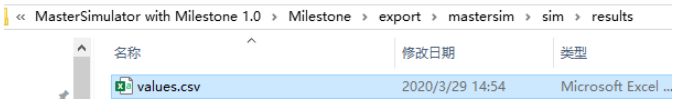


图 3.13: MasterSim 仿真结果路径

- 识别该 csv 文件的分隔符，用 Excel 等工具格式化查看，如 图 3.14 。
- 绘制仿真结果，验证模型的正确性，如 图 3.15 。

## 3.2 示例模型说明

控制器模型为 PD 测速反馈控制器，如 Eq.3.1 。

$$F = k_p\left(r_x - x\right) - k_d v \tag{3.1}$$

被控对象为简单的一维质量块模型，如 Eq.3.2 。

$$\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v \\ \frac{F}{m} \end{bmatrix} \tag{3.2}$$

所组成控制系统的原理框图如 图 3.16 所示。

系统的传递函数为 Eq.3.3 。

$$\frac{X\left(s\right)}{R\left(s\right)} = \frac{k_p \frac{1}{s} \frac{\frac{1}{ms}}{1+k_d * \frac{1}{ms}}}{1 + k_p \frac{1}{s} \frac{\frac{1}{ms}}{1+k_d * \frac{1}{ms}}} = \frac{\frac{k_p}{ms^2+k_d s}}{1 + \frac{k_p}{ms^2+k_d s}} = \frac{k_p}{ms^2 + k_d s + k_p} = \frac{1}{\frac{ms}{k_p}s^2 + \frac{k_d}{k_p}s + 1} \tag{3.3}$$

$$D\left(s\right) = \frac{ms}{k_p}s^2 + \frac{k_d}{k_p}s + 1 = \frac{1}{\omega_n^2}s^2 + 2\frac{\zeta}{\omega_n}s + 1 \tag{3.4}$$

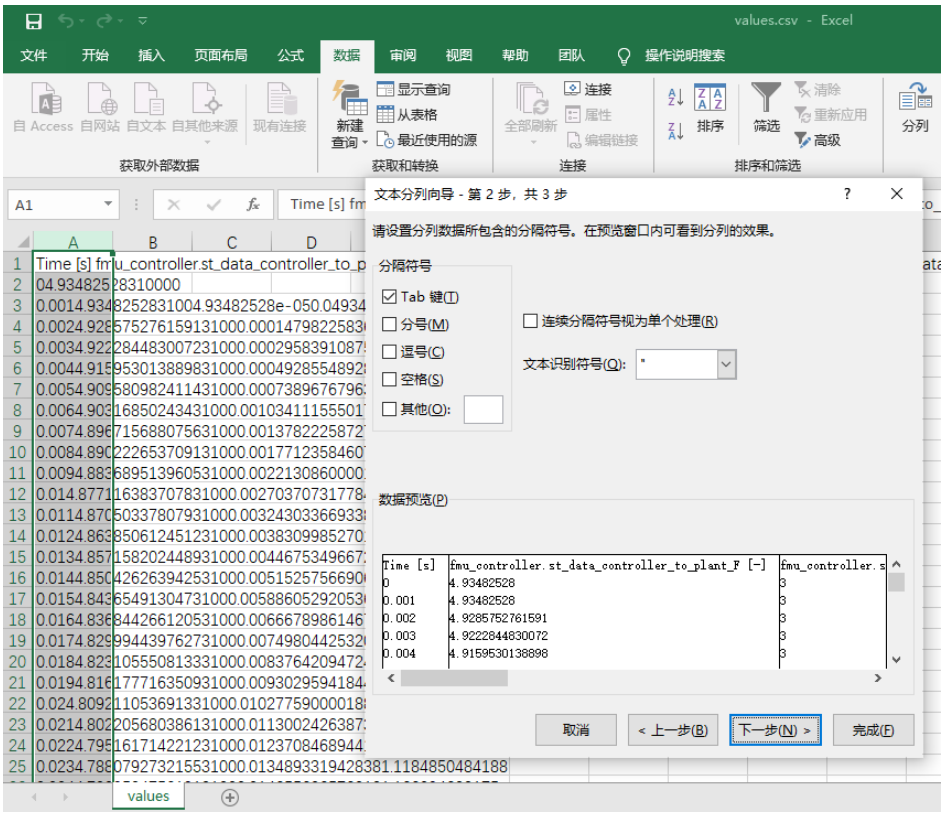


图 3.14: Excel 中查看结果 csv 数据文件

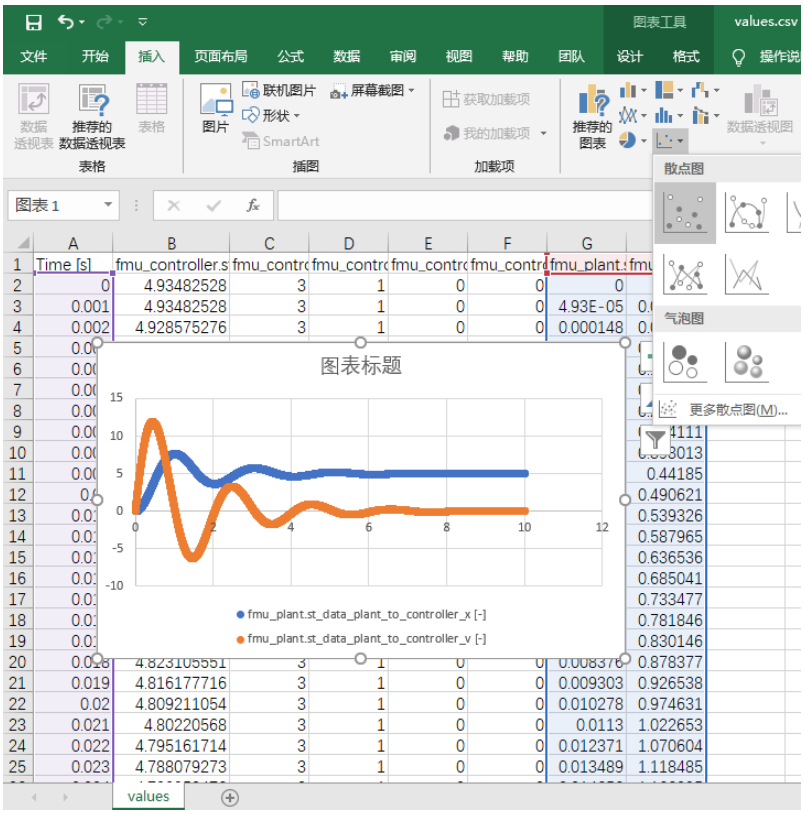


图 3.15: 系统时域响应曲线

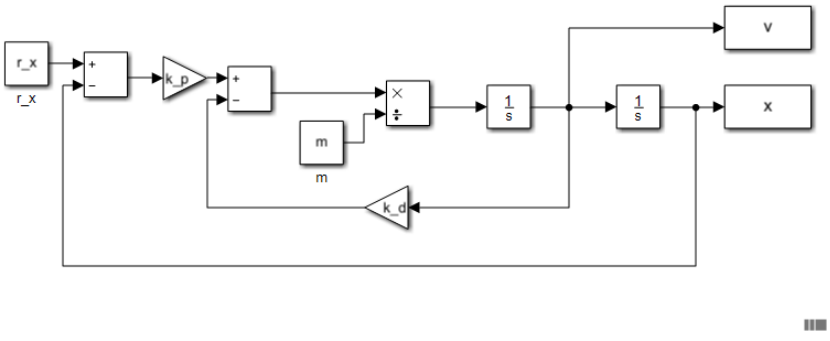


图 3.16: 控制系统框图

可见该反馈控制系统为典型的二阶系统，其特征方程为 Eq.3.4 ，为使得系统的响应具有较明显的动态过程，选择系统参数使得阻尼比较小且振荡频率为 0.5Hz 即取 $\zeta = 0.2, \omega_n = 2\pi \times 0.5$ ，则系统的反馈增益为 $k_p = \omega_n^2 m, k_d = 2\zeta\omega_n m$ 。对于这个简单的模型，当然没有必要分别使用不同的仿真工具对其不同部分进行建模，但我们为了快速验证系统运行的正确性，可以以此作为测试用例，容易给出在 Simulink 中的仿真结果与设计的预期相一致，如 图 3.17 。

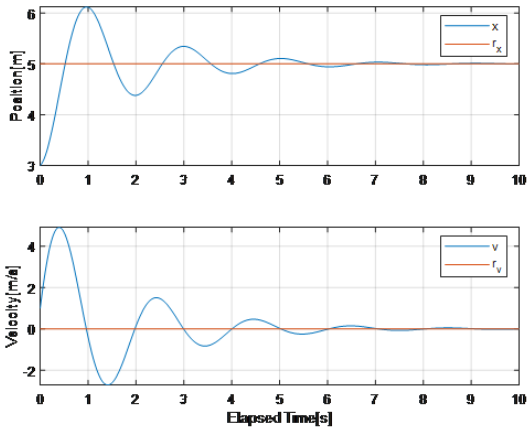


图 3.17: 预期的系统响应

接下来，将控制器和被控对象分别实现为两个模型，通过接口的连接实现该系统的仿真，则系统的接口关系为 图 3.18 。

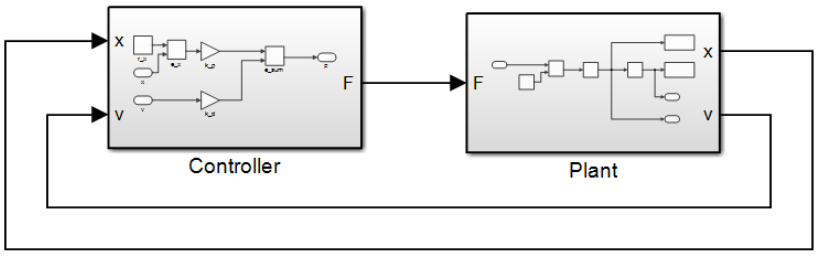


图 3.18: 系统的接口关系

# 4 图形用户界面操作

## 4.1 新建模型并生成模板

请先参考编辑模型的操作，如打开已有的模型头文件 。

若要全新创建模型，请先备份并复制示例模型的全局接口头文件与模型目录结构，编辑全局接口头文件中的接口数据类型定义；然后，选择工作路径到工具包中创建好的模型代码目录：点击“设置工作路径”按钮，打开系统的路径选择对话框，拾取工具包中的 model/model_name/sources 目录，如 图 4.1 。然后，通过界面中的相应区域，填写模型头文件中的信息，生成代码模板，其他操作同编辑已有模型。

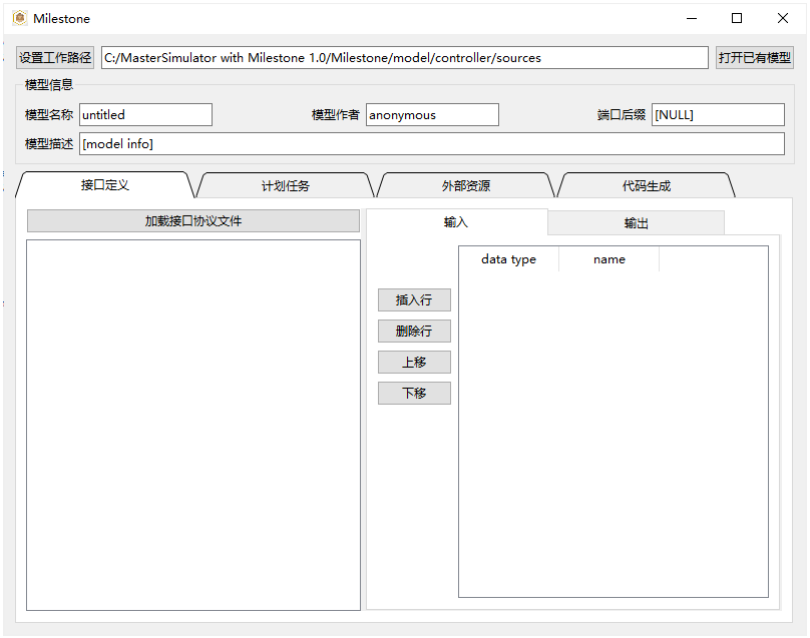


图 4.1: 设置工作路径

## 4.2 打开已有的模型头文件

通过界面中的相应区域，修改从模型头文件中载入的信息。

- 点击“打开已有模型”按钮，打开系统的文件选择对话框，拾取模型的头文件，如 图 4.2 。

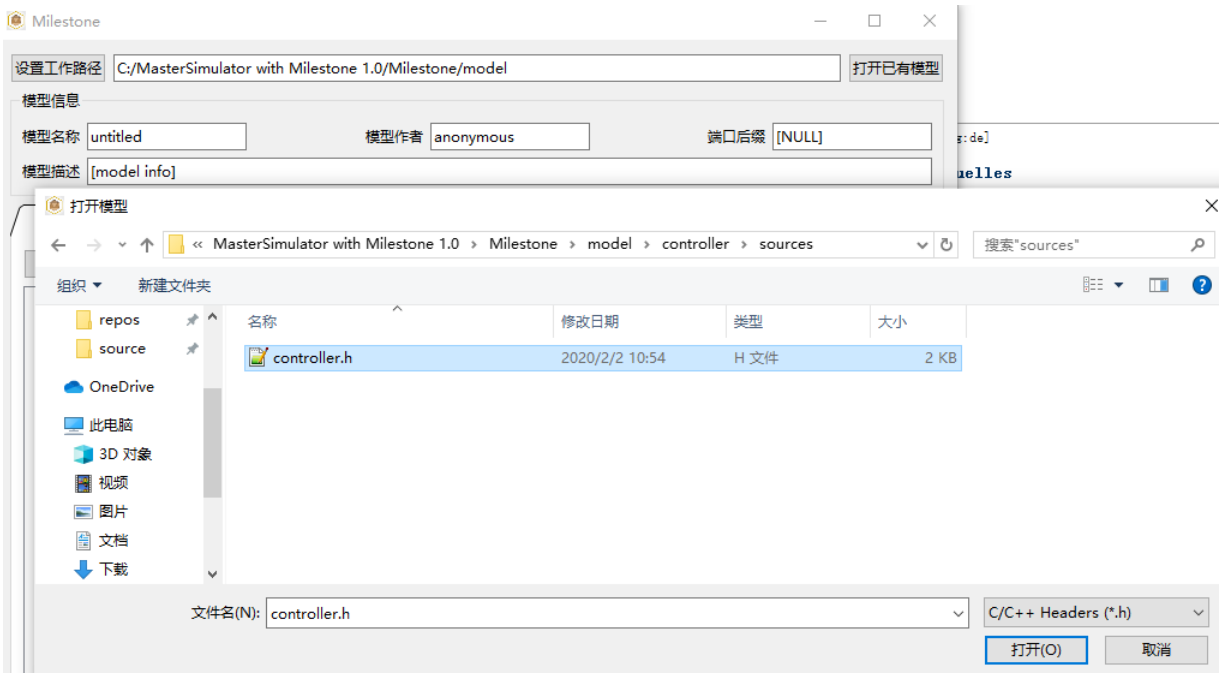


图 4.2: 打开已有模型

- 点击“加载接口协议文件”按钮，打开系统的文件选择对话框，拾取 interface.h 全局接口头文件，如 图 4.3 。
- 从左侧列表解析出的接口数据结构中，点选拖动结构体定义到右侧输入、输出接口定义区域，如 图 4.4 。
- 也可以直接键入接口定义表格中的内容，录入后可以通过中间的编辑按钮，对输入、输出列表中的行数据进行编辑，如 图 4.5 。
- 点击上方标签页，切换至“计划任务”，如 图 4.6 。其中 task ID 为可选的录入区域，系统会自动从零开始编号；按照表头键入任务的定时周期和启动偏移时间；左侧的编辑按钮可对已经录入的行进行整体编辑。
- 点击上方标签页，切换至“外部资源”，如 图 4.7 。其中 resource ID 为可选的录入区域，系统会自动从零开始编号；按照表头键入资源文件的文件名（运行时文件需要预先放置到模型的 resources 目录）；左侧的编辑按钮可对已经录入的行进行整体编辑。
- 点击上方标签页，切换至“代码生成”。点击“生成代码”将在当前工作路径生成模型的头文件和源文件模板，对于打开的已有模型，仅更新头文件，不会覆盖已经实现的模型源文件，并给出提示，如 图 4.8 。
- 点击“编辑代码”，将使用配置中选择的编辑器打开已经生成的模型头文件和源文件，如 图 4.9 。
- 点击“创建工程”，弹出模型选择对话框，在列表中选择需要构建的模型，如 图 4.10 。确定后开始创建工程。
- 观察创建工程过程中控制台输出的信息，如 图 4.11 ，请确认对系统中编译环境的测试是否通过。

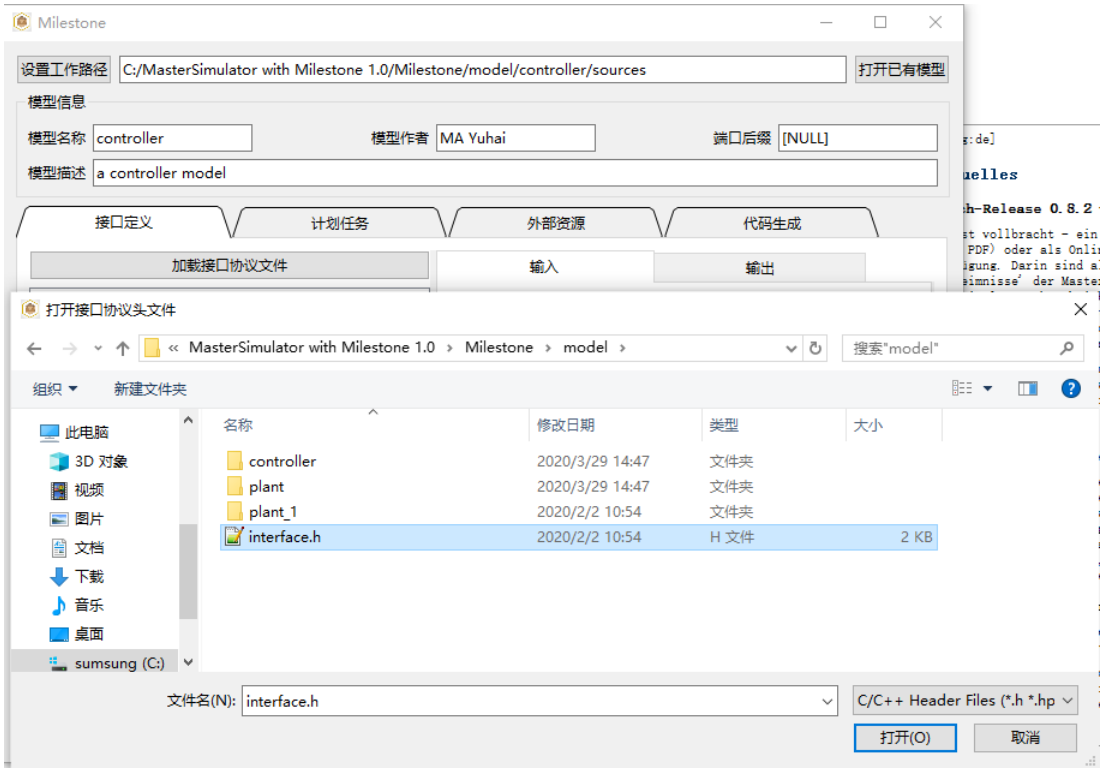


图 4.3: 加载接口协议文件

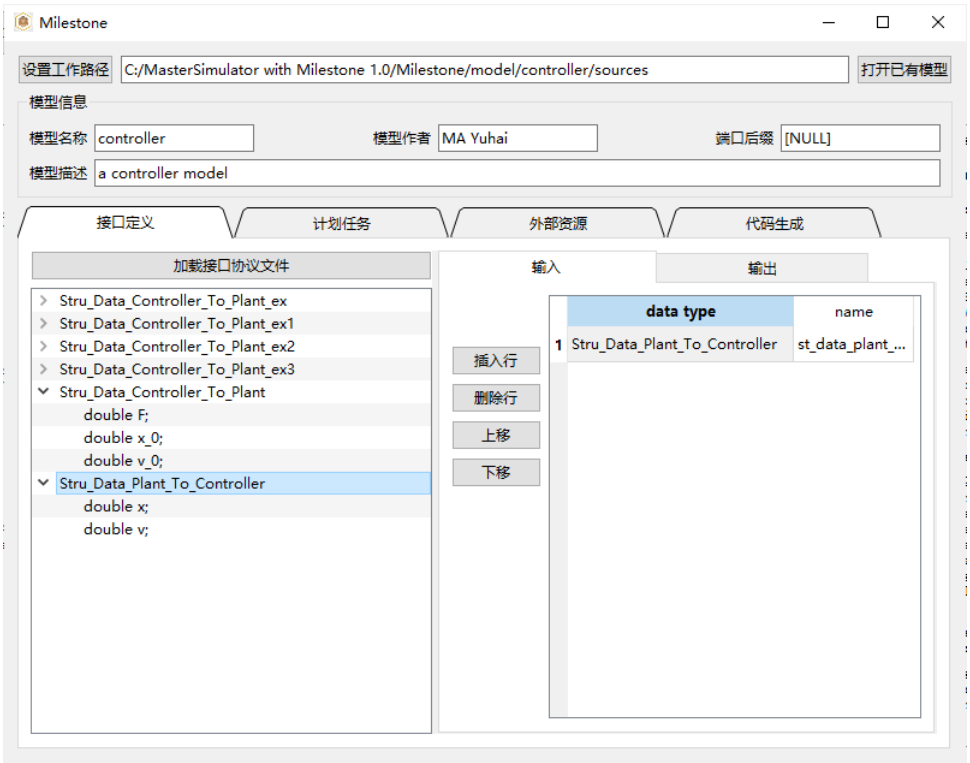


图 4.4: 拖动结构体定义到输入

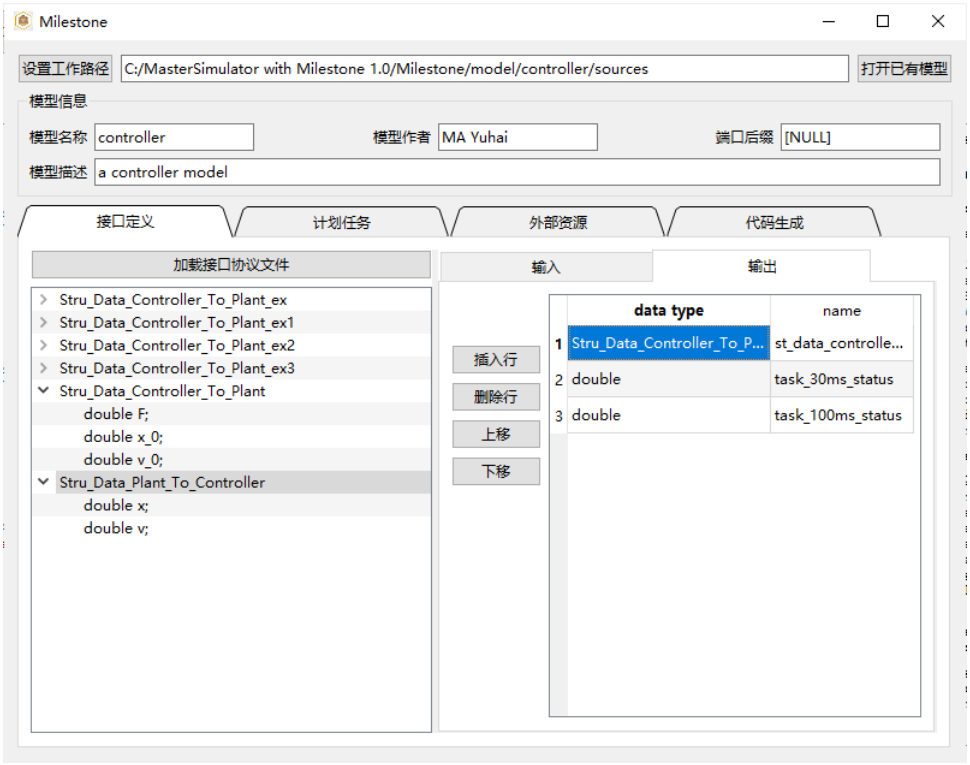


图 4.5: 拖动结构体定义到输出

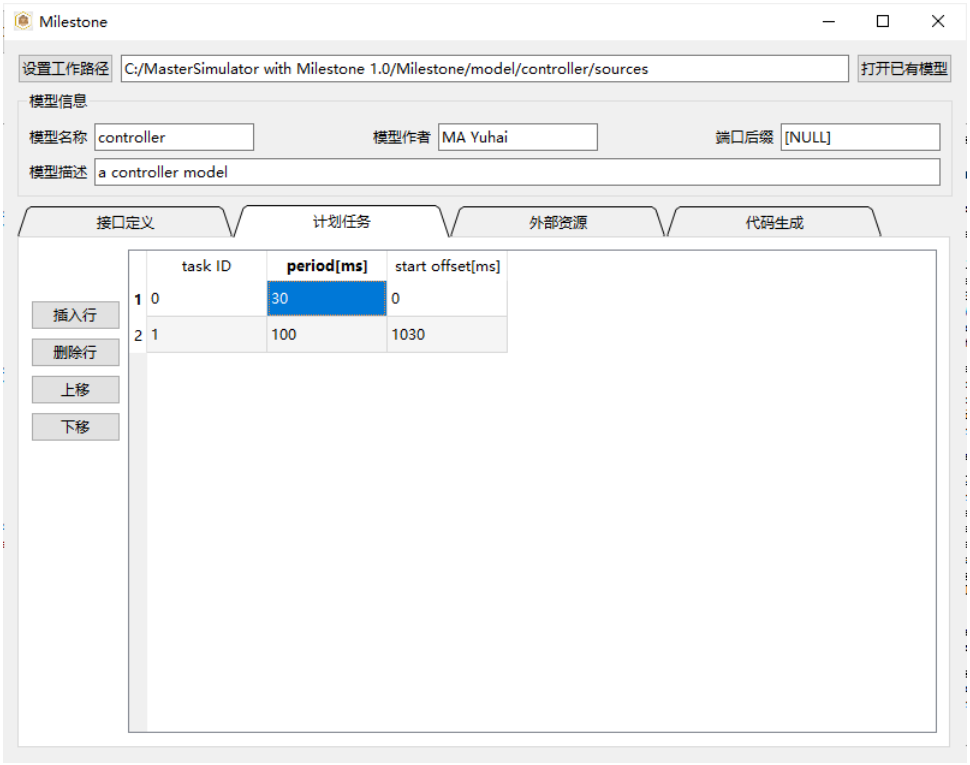


图 4.6: 定义计划任务

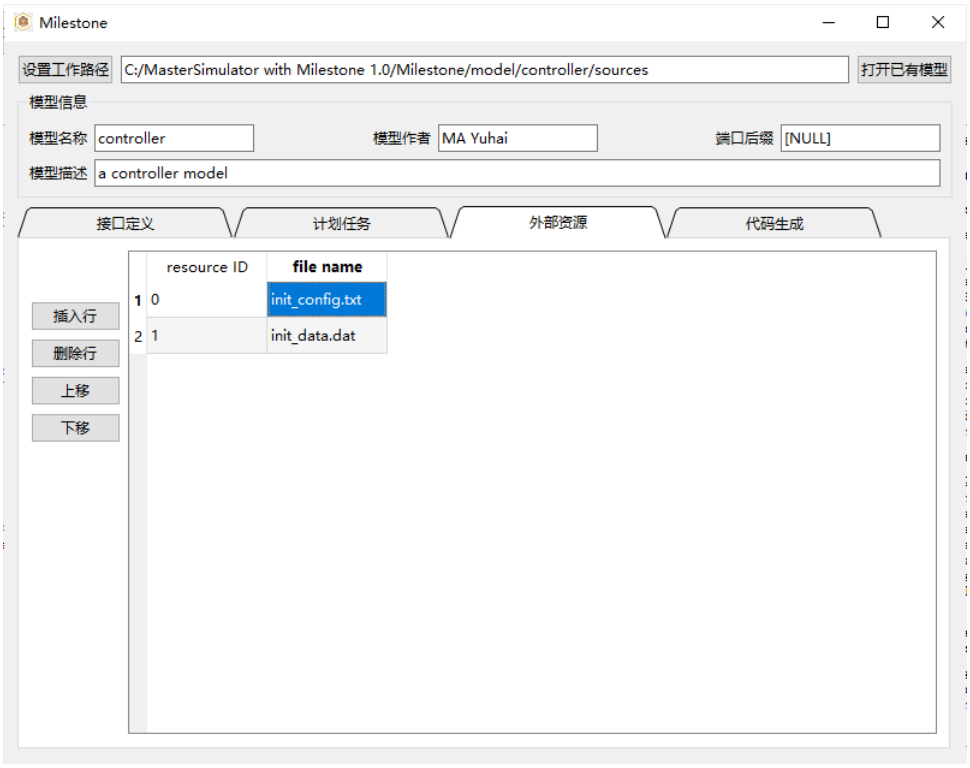


图 4.7: 定义外部资源

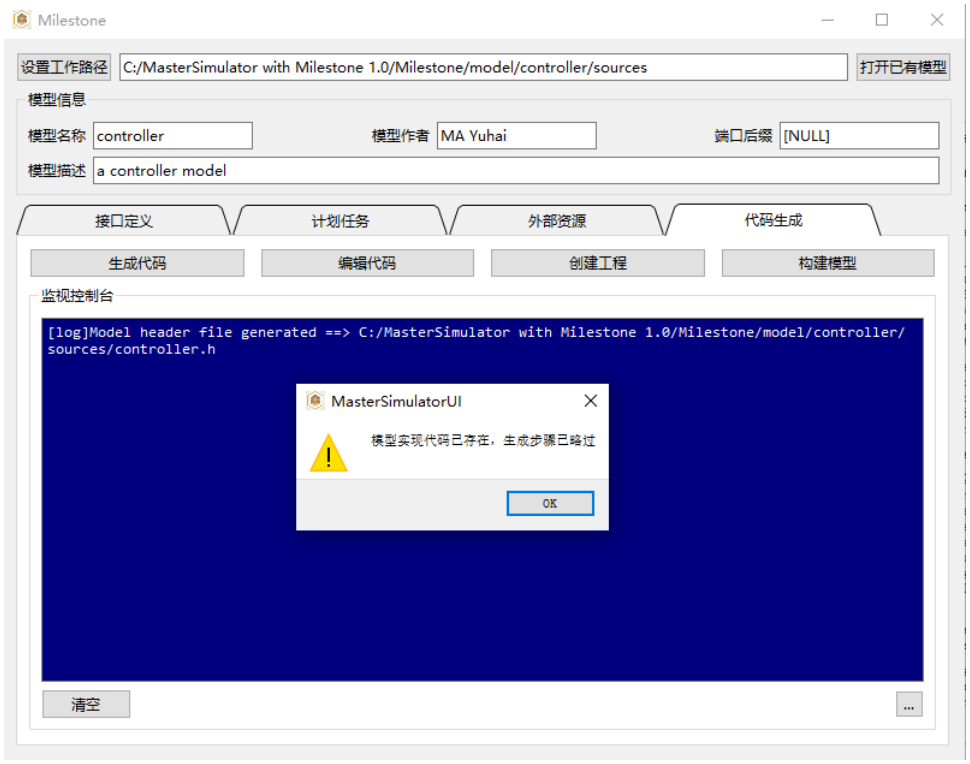


图 4.8: 生成代码

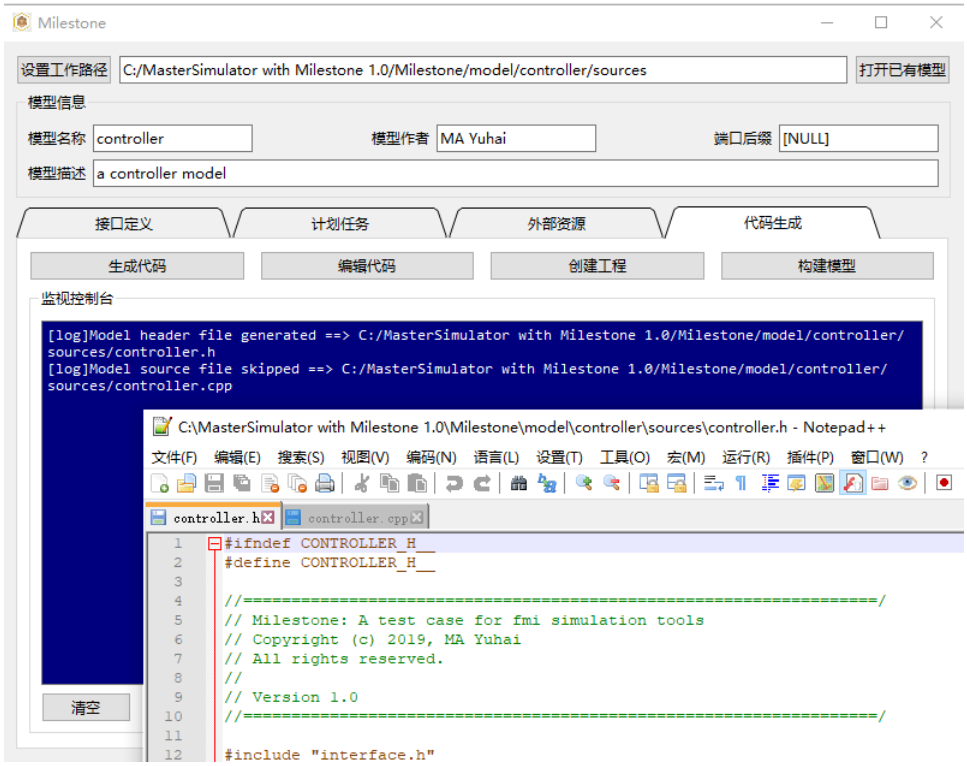


图 4.9: 编辑代码

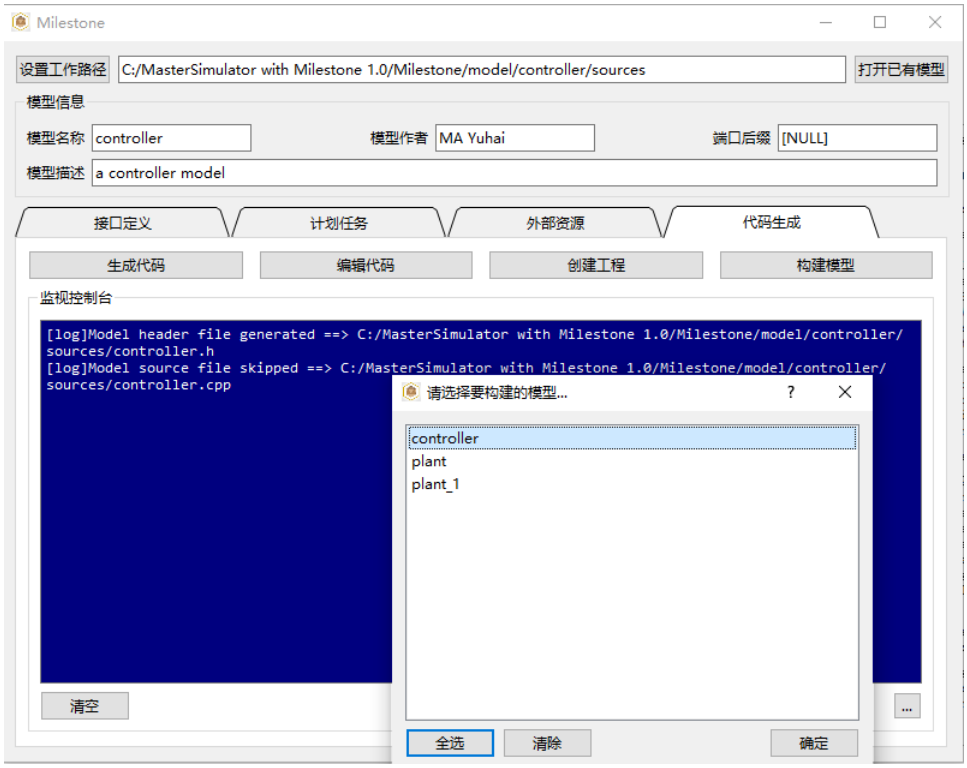


图 4.10: 选择要构建的模型

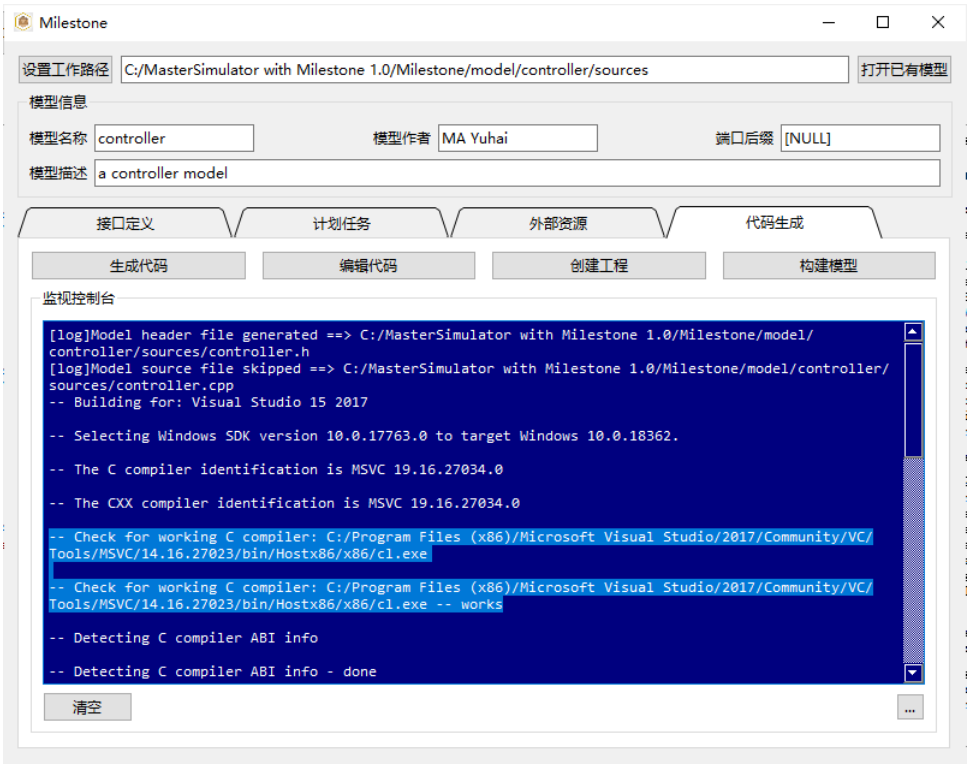


图 4.11: 开始创建，确认编译环境测试通过

- 构建完成后控制台输出信息结束，如 图 4.12 。
- 点击“构建模型”，开始编译、模型打包与测试过程，如 图 4.13 。请确认工具包授权检测通过，并生成了相关文件。
- 构建完成后将对生成的 FMU 进行零输入测试，给出运行报告，如 图 4.14 。

## 4.3 模型模板的代码结构及资源接口

参见*FMI 接口及其实现* 。

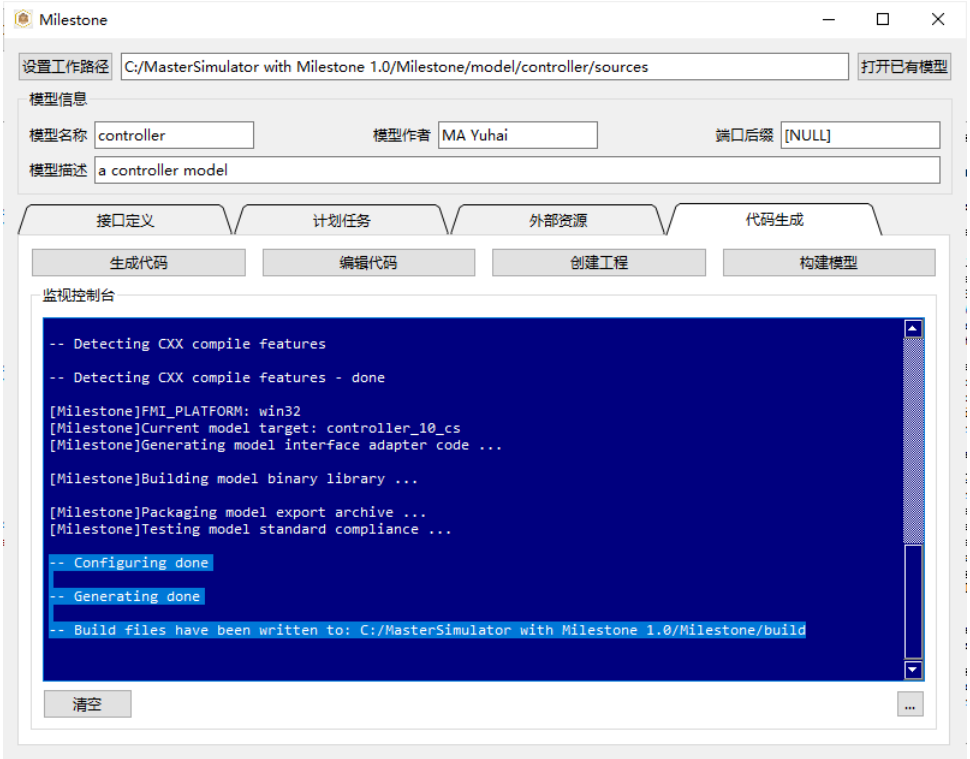


图 4.12: 完成创建，确认生成编译工程

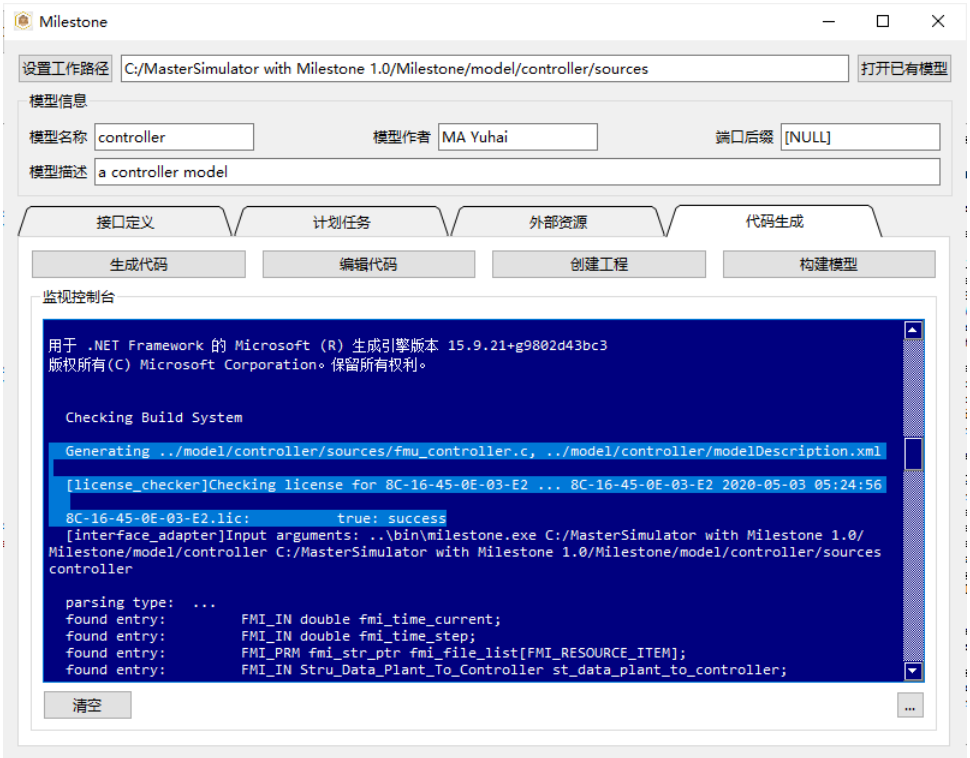


图 4.13: 开始构建，确认授权通过

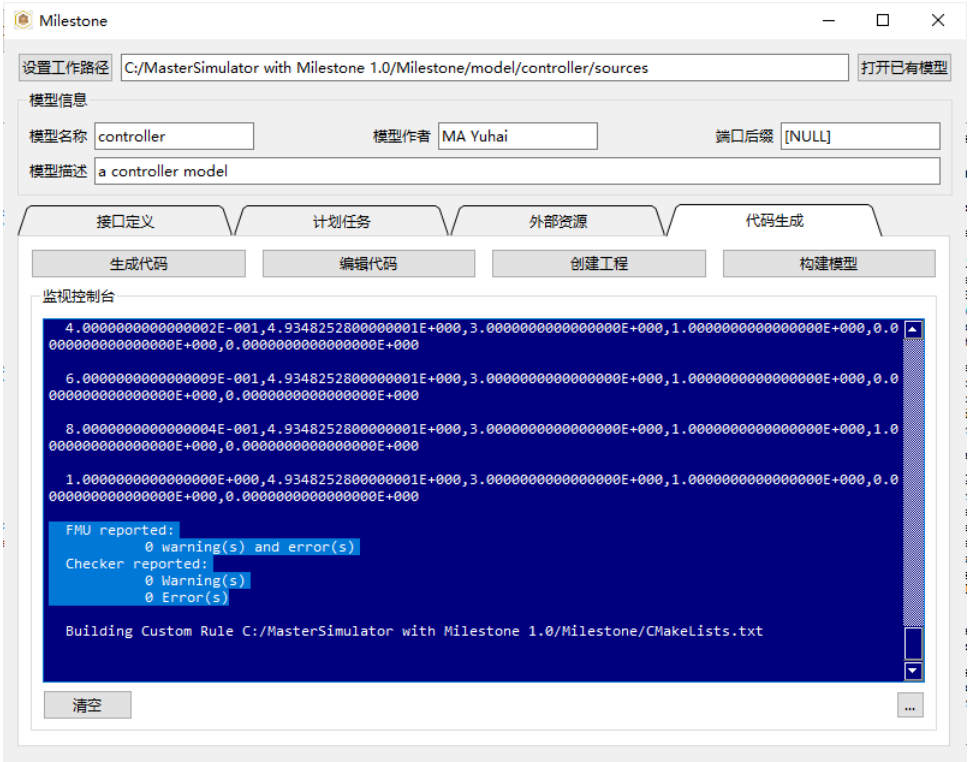


图 4.14: 完成构建，确认测试通过

# 5 命令行操作

## 5.1 运行示例

为使开发者从整体上把握工具包的组织方式，这里具体地给出测试用例在 CLI 中的执行方式如下。

1. 在工具包根目录下执行”mkdir build”//建立单独的构建目录，名称任意，用于将临时文件与工具分开
2. 在工具包根目录下执行”cd build”//切换到创建的构建目录
3. 在创建的构建目录下执行”cmake ..”//在构建目录下，指定代码目录在上层目录，生成编译工程文件（Windows 下为 MSVC sln，Linux 下为 Makefile）
4. 在创建的构建目录下执行”cmake –build . ”//执行构建，注意”.”为当前目录，附加”–config Release”或”–config Debug”参数切换 Release 版和 Debug 版，默认为 Debug 版
5. 所导出的 fmu 模型在 export 目录中
6. 依次在不同系统下执行工具包，将 model 目录（保留已生成的中间文件）或整个工具包复制到其他系统继续构建，将获得同时支持多系统的 fmu 文件

## 5.2 添加模型

1. 复制 model 内部的模型目录结构（内部 sources 文件夹为必须），实现与模型文件夹同名称的.h 及.cpp 模型代码文件
2. 若增加新的模型间接口，在 model/interface.h 中定义接口数据结构体
3. 在顶层 CMakeLists.txt 中”foreach (MODEL_NAME controller plant plant_1) # add model to this list”语句处，将新的模型添加在列表中
4. 重新执行上述构建操作，系统将执行增量构建

## 5.3 构建模型

为避免污染程序目录结构，支持 shadow 构建，运行 CMake，生成编译工程，如 图 5.1 。

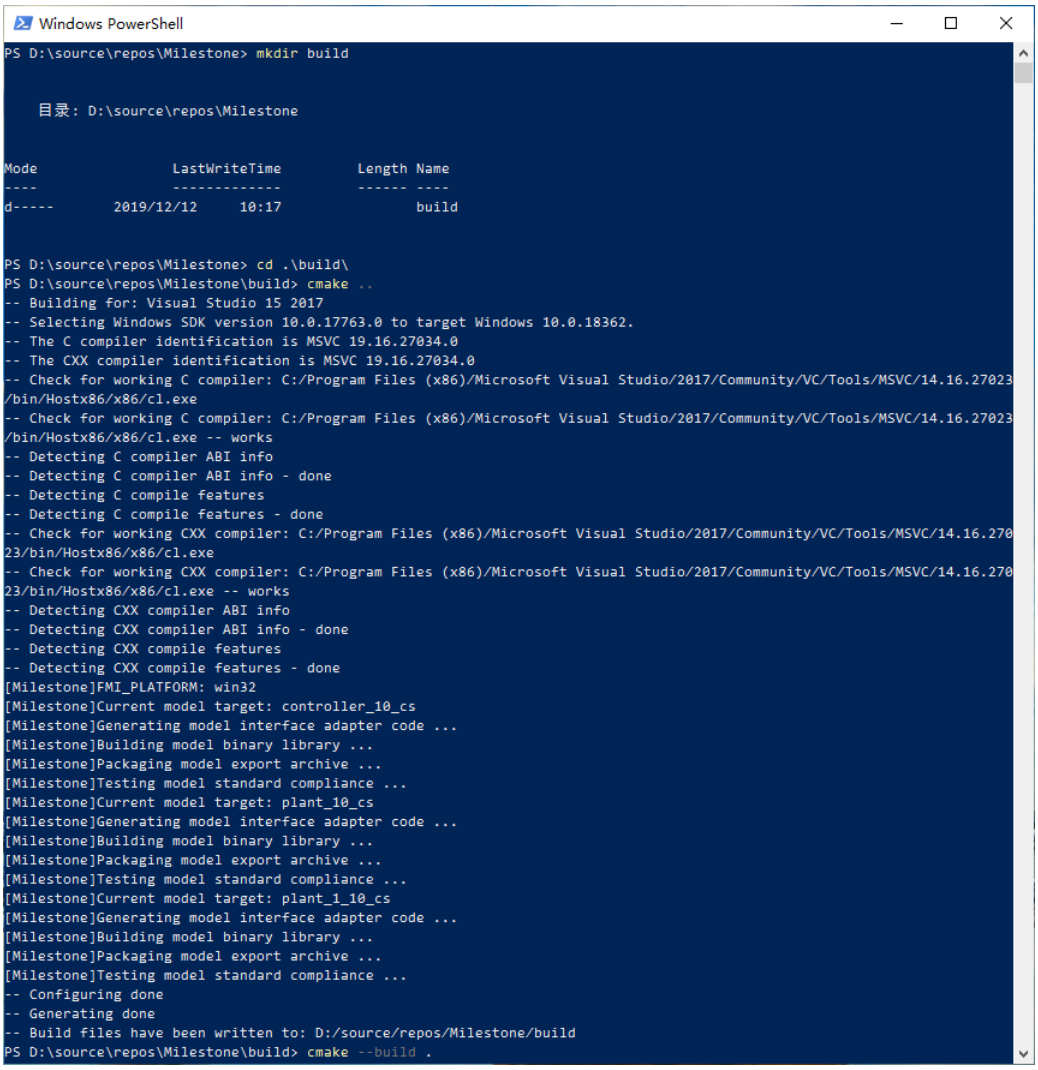


图 5.1: 创建 shadow build 目录，生成构建工程（Windows）

程序将完成结构多层次递归展开，FMU 结构代码生成，xml 描述文件生成，如 图 5.2 。

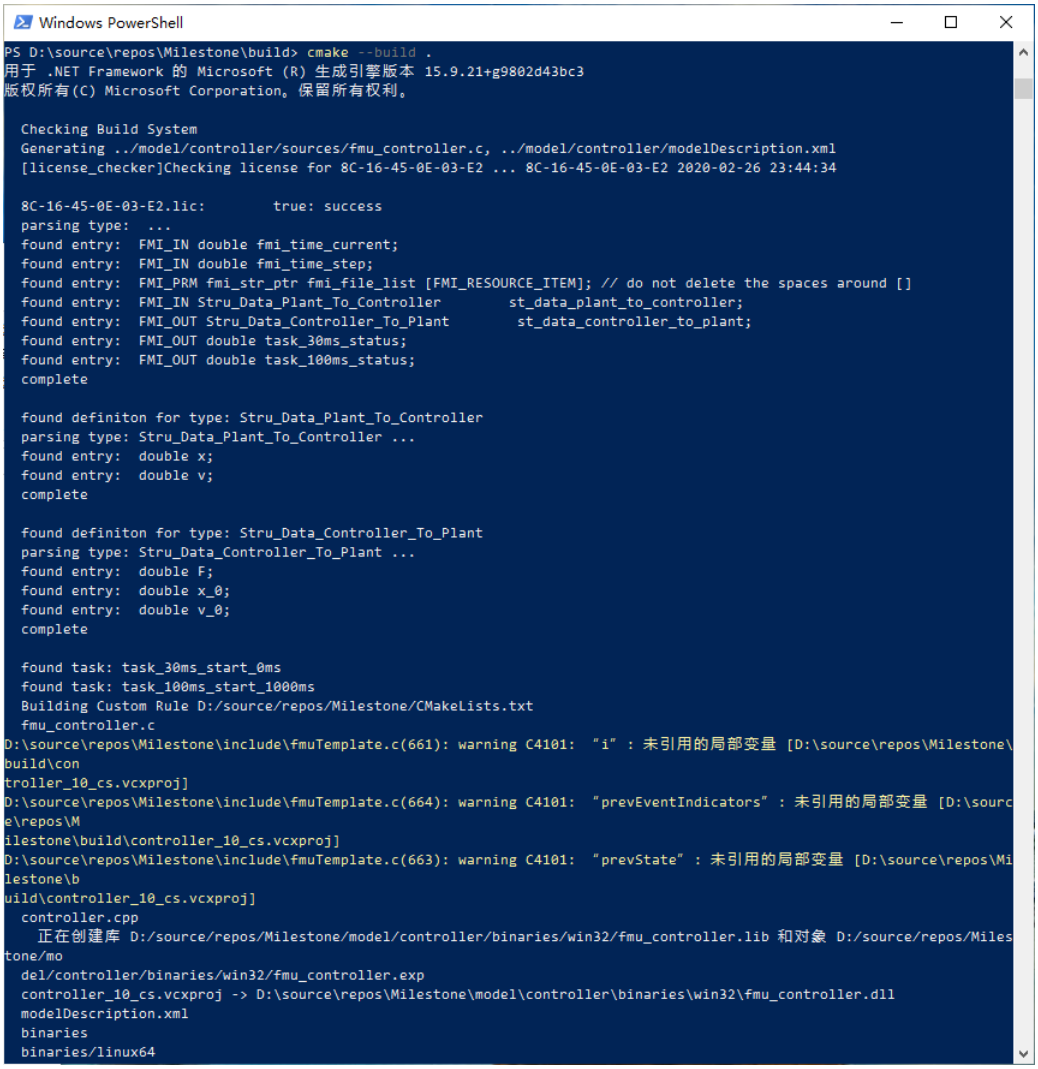


图 5.2: 通过 CMake 执行编译及后处理

程序完成 FMU 链接库构建，打包及测试流程，如 图 5.3 。

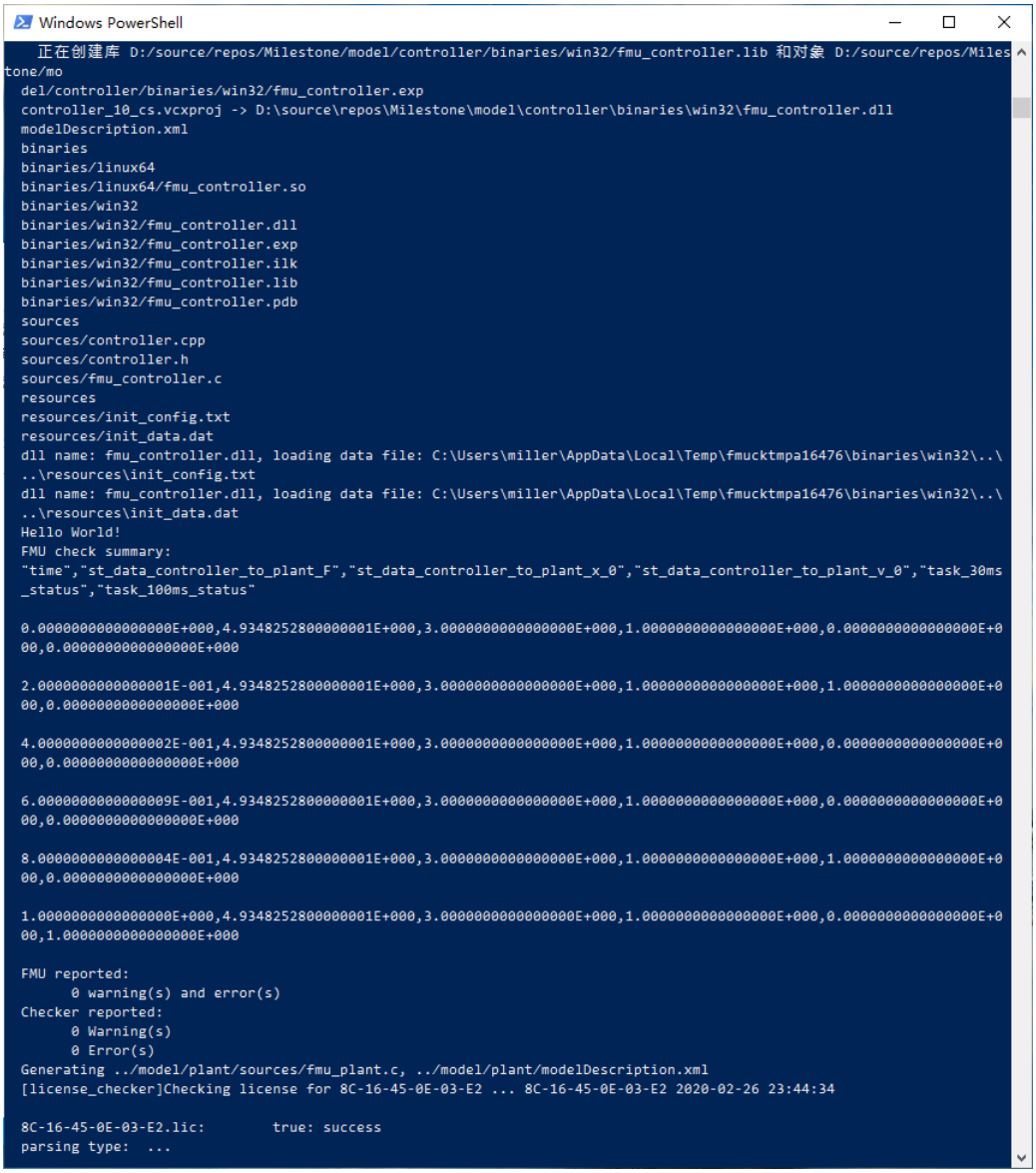


图 5.3: FMU 的自动打包和测试

# 6 FMI 接口及其实现

本工具在 FMI 接口代码的基础上，简化了多数固有的流程性代码以及繁琐的接口定义操作，并添加了一些实用功能，代码模板的运行流程如 图 6.1 。代码的实现请阅读代码结构剖析 相关注释。

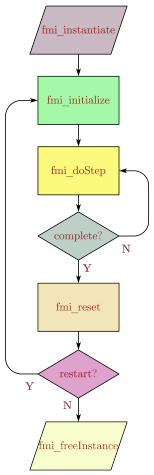


图 6.1: 模型代码模板运行流程

## 6.1 代码结构剖析

### 6.1.1 全局接口头文件

称 interface.h 文件为全局接口头文件，用于对模型间公用的接口数据类型进行定义。对于我们的测试用例，从 图 3.18 中可以看出，控制器需要从被控对象反馈当前的位置和速度，运算后输出推力，而被控对象受到力的作用后，经过其动力学微分方程，其状态量（位置和速度）发生变化。因而，我们在 interface.h 中定义了如下的数据结构用于传递两个模型间需要通信的数据。

```
typedef struct _struct_name
{
    data_type struct_field;
```

（下页继续）

(续上页)

```
    ...
}struct_name;
```

全局接口头文件 interface.h 中的内容：

```
#ifndef INTERFACE_H__    /* 避免重复包含 */
#define INTERFACE_H__
//====================================================================/
// A test case for fmi simulation tools
// Copyright (c) 2019 马玉海
// All rights reserved.
//
// Version 1.0
//====================================================================/
#define _CRT_SECURE_NO_WARNINGS    /* 抑制 cl 编译器对传统标准库的警告 */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>
#include <float.h>

#define IO_PORT_FLUSH(data_type, var_name) \    /* 工具宏定义，用于重置端口数据 */
    do{\
        memset(&(p->var_name), 0, sizeof(data_type));\
    } while(0);

#ifndef __cplusplus    /* 针对 C89 的兼容性定义 */
#define FMI_EXPORT
#define bool unsigned char
#define true 1
#define false 0
#else
#define FMI_EXPORT extern "C"    /* 针对 C++ 的兼容性定义 */
#endif

#ifndef _WIN32    /* 针对 Linux 的兼容性定义 */
#include <limits.h>
#define _MAX_PATH PATH_MAX
#define _MAX_FNAME NAME_MAX
#define _MAX_EXT NAME_MAX
#endif
// non-standard interface definition
#define FMI_IN    /* 输入端口格式标记 */
#define FMI_OUT    /* 输出端口格式标记 */
#define FMI_PRM    /* 参数端口格式标记（暂未使用） */
typedef const char * fmi_str_ptr;    /* 运行时的内部资源文件路径类型 */

FMI_EXPORT void *fmi_instantiate(void);    /* 导出接口函数声明 */
FMI_EXPORT int fmi_initialize(void *);
FMI_EXPORT int fmi_doStep(void *);
FMI_EXPORT int fmi_reset(void *);
FMI_EXPORT void fmi_freeInstance(void *);

#pragma pack(push, 8)    /* 强制结构体内部字节对齐 */
typedef struct _Stru_Data_Controller_To_Plant_ex    /* 测试定义格式 0 */
```

(下页继续)

(续上页)

```
{
    double y;
    double z;
}Stru_Data_Controller_To_Plant_ex;

typedef struct _Stru_Data_Controller_To_Plant_ex1{    /* 测试定义格式 1 */
    double y;
    double z;
}Stru_Data_Controller_To_Plant_ex1;
typedef struct _Stru_Data_Controller_To_Plant_ex2 {    /* 测试定义格式 2 */
    double y;
    double z;
}Stru_Data_Controller_To_Plant_ex2;
typedef struct _Stru_Data_Controller_To_Plant_ex3   {    /* 测试定义格式 3 */
    double y;
    double z;
}Stru_Data_Controller_To_Plant_ex3;

typedef struct _Stru_Data_Controller_To_Plant{    /* 接口结构体定义 1 */
    double F;
    double x_0;
    double v_0;
}Stru_Data_Controller_To_Plant;

typedef struct _Stru_Data_Plant_To_Controller    /* 接口结构体定义 2 */
{
    double x;
    double v;
}Stru_Data_Plant_To_Controller;

#pragma pack(pop)    /* 恢复结构体内部字节对齐 */

#endif // INTERFACE_H__
```

### 6.1.2 控制器模型

为实现 Eq.3.1 中描述的控制器数学模型并与被控对象模型相连接，分解后的控制器原理框图如 图 6.2 。

模型头文件 controller.h 的内容：

```
#ifndef CONTROLLER_H__
#define CONTROLLER_H__

//====================================================================/
// Milestone: A test case for fmi simulation tools
// Copyright (c) 2019, MA Yuhai
// All rights reserved.
//
// Version 1.0
//====================================================================/

#include "interface.h"

#define FMI_MODEL_AUTHOR "MA Yuhai"    /* 模型作者 */
```

(下页继续)

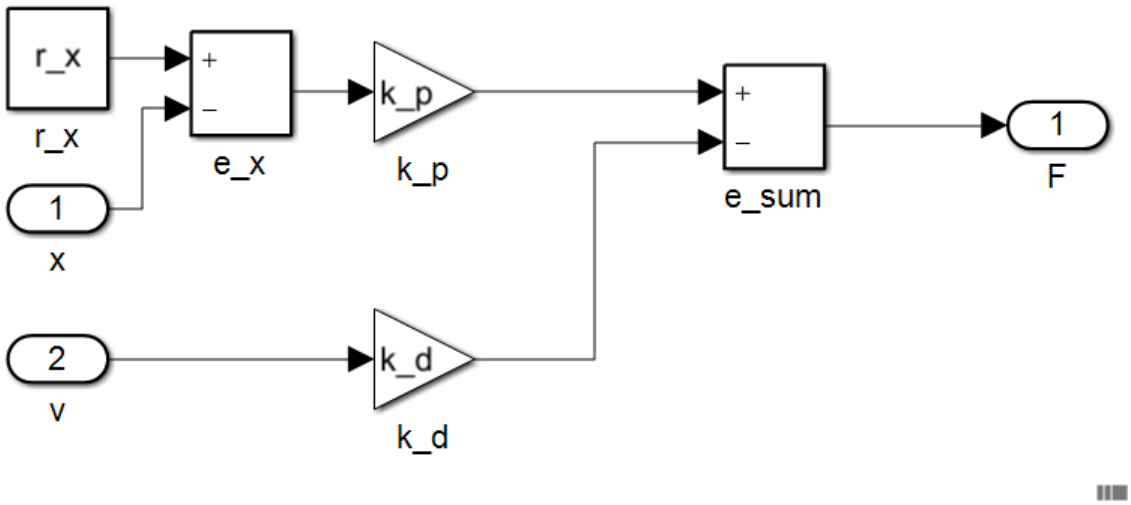


图 6.2: 控制器原理框图

(续上页)

```
#define FMI_MODEL_NAME "controller"    /* 模型名称 */
#define FMI_MODEL_DISCRIPTION "a controller model"    /* 模型描述 */
#define FMI_PORT_POSTFIX ""    /* 端口后缀 */

// resource file definition if any    /* 外部资源文件需要放置在模型的 resources 目录下 */
#define FMI_RESOURCE_ITEM 2    /* 外部资源文件数量定义 */
#if FMI_RESOURCE_ITEM>0 && defined EN_RES_ACCESS
const char *resource_file_list[FMI_RESOURCE_ITEM] = {
    "init_config.txt", "init_data.dat" };    /* 外部资源文件名称定义列表 */
#endif                                       /* 列表中超出 FMI_RESOURCE_ITEM 的项目将被忽略 */

// task definition if any in the unit of [ms]    /* 格式 task_[period]ms_start_[offfset]ms */
#define FMI_TASK_ITEM 2    /* 计划任务数量定义 */
FMI_EXPORT void task_30ms_start_0ms(void);    /* 计划任务触发函数 1 */
FMI_EXPORT void task_100ms_start_1030ms(void);    /* 计划任务触发函数 2 */

// define interface variables by an fmi object
// one statement per line, no extra semicolons allowed
// do not modify internal variables
typedef struct st_fmi_object_t {
    // internal variables    /* 必须的内部变量，当前时间、步长和文件列表 */
    FMI_IN double fmi_time_current;
    FMI_IN double fmi_time_step;
#if FMI_RESOURCE_ITEM>0
    FMI_PRM fmi_str_ptr fmi_file_list[FMI_RESOURCE_ITEM];    /* 运行时可用的外部资源文件名列表 */
#endif

    // interface variables
    FMI_IN Stru_Data_Plant_To_Controller st_data_plant_to_controller;    /* 输入接口结构体 */
    FMI_OUT Stru_Data_Controller_To_Plant st_data_controller_to_plant;    /* 输出接口结构体 */
```

(下页继续)

(续上页)

```
    FMI_OUT double task_30ms_status;    /* 自定义输出 1, 30ms 任务监控 */
    FMI_OUT double task_100ms_status;    /* 自定义输出 2, 100ms 任务监控 */
}st_fmi_object;    /* 模型名称 */
#endif // CONTROLLER_H__
```

模型源文件 controller.cpp 中的内容:

```
#include "controller.h"
#include <iostream>    /* 不限制模型内部的实现方式, 可以使用 C++ 的类、STL 等特性 */
#include <fstream>
#include <string>
using namespace std;

double x;    /* 可以在模型内部自定义全局变量 */
double v;
double F;
double x_0;
double v_0;
int task_30ms_trigger;
int task_100ms_trigger;

void load_initial_data(fmi_str_ptr fmi_file_list[])    /* 可以在模型内部自定义函数 */
{
    ifstream init_file;

    init_file.open(fmi_file_list[0]);
    if (!init_file.is_open()) {
        cout << "open data file error: " << fmi_file_list[0] << endl;
    }
    else {
        string buff;
        getline(init_file, buff);
        cout << buff << endl;    /* 打印资源文件中的内容 */
    }

    init_file.close();

    init_file.open(fmi_file_list[1]);
    if (!init_file.is_open()) {
        cout << "open data file error: " << fmi_file_list[1] << endl;
    }
    else {
        init_file >> x_0;    /* 读取资源文件中的内容作为初始值传递给被控对象 */
        init_file >> v_0;    /* 注意! 一般情况下这并不会生效, 和求解器的实现方式有关 */
    }    /* 一般情况下, 初始化阶段不会按照模型的连接关系按顺序执行, 并交换接口变量 */

    init_file.close();
    return;
}

void task_30ms_start_0ms(void)    /* 头文件中定义的定时任务触发函数必须实现 */
{
    task_30ms_trigger = task_30ms_trigger ? 0 : 1;
}

void task_100ms_start_1030ms(void)
```

(下页继续)

(续上页)

```
{
    task_100ms_trigger = task_100ms_trigger ? 0 : 1;
}

void* fmi_instantiate(void)    /* 实例化函数，在模型加载后被调用 */
{
    st_fmi_object *p =    /* 模板内容均为必须的操作，请勿删除 */
        (st_fmi_object *)calloc(1, sizeof(st_fmi_object));
    if (!p) {
        fprintf(stderr, "fmi_instantiate failed in model controller!\n");
        exit(EXIT_FAILURE);
    }
    /* 在模板代码后，可添加自定义的操作，如打印信息 */
    return p;
}

int fmi_initialize(void *fmi_object)    /* 初始化函数，在模型启动或重置时被调用 */
{
    st_fmi_object *p = (st_fmi_object *)fmi_object;

    load_initial_data(p->fmi_file_list);    /* 可通过 p 指针访问接口上的所有变量及文件资源 */
    p->st_data_controller_to_plant.x_0 = x_0;
    p->st_data_controller_to_plant.v_0 = v_0;

    return 0;
}

int fmi_doStep(void *fmi_object)    /* 步进函数，每一个步长推进的周期被调用 */
{
    st_fmi_object *p = (st_fmi_object *)fmi_object;
    const double pi = 3.1416;    /* 可在模型中自定义参数常量 */
    const double r_x = 5;
    const double m = 0.1;

    const double zeta = 0.2; // let it oscillates
    const double omega_n = 2*pi*0.5;
    const double k_p = omega_n*omega_n*m;
    const double k_d = 2*zeta*omega_n*m;

    x = p->st_data_plant_to_controller.x;    /* 可选择将接口内存变量赋值到较方便的名称 */
    v = p->st_data_plant_to_controller.v;

    F = k_p * (r_x - x) - k_d * v;    /* 执行模型计算 */

    p->st_data_controller_to_plant.F = F;    /* 将计算后的结果发布到接口内存上 */
    p->task_30ms_status = task_30ms_trigger;
    p->task_100ms_status = task_100ms_trigger;

    return 0;
}

int fmi_reset(void *fmi_object)    /* 复位函数，在重置模型时被调用 */
{
    st_fmi_object *p = (st_fmi_object *)fmi_object;
    IO_PORT_FLUSH(Stru_Data_Controller_To_Plant, st_data_controller_to_plant);    /* 清
空输出接口内存 */
```

(下页继续)

(续上页)

```
    return 0;
}

void fmi_freeInstance(void *fmi_object)    /* 释放函数，在模型卸载时被调用 */
{
    st_fmi_object *p = (st_fmi_object *)fmi_object;

    free(p);
}
```

### 6.1.3 被控对象模型

为实现 Eq.3.2 中描述的被控对象模型并与控制器数学模型相连接，分解后的被控对象原理框图如 图 6.3 。[2]

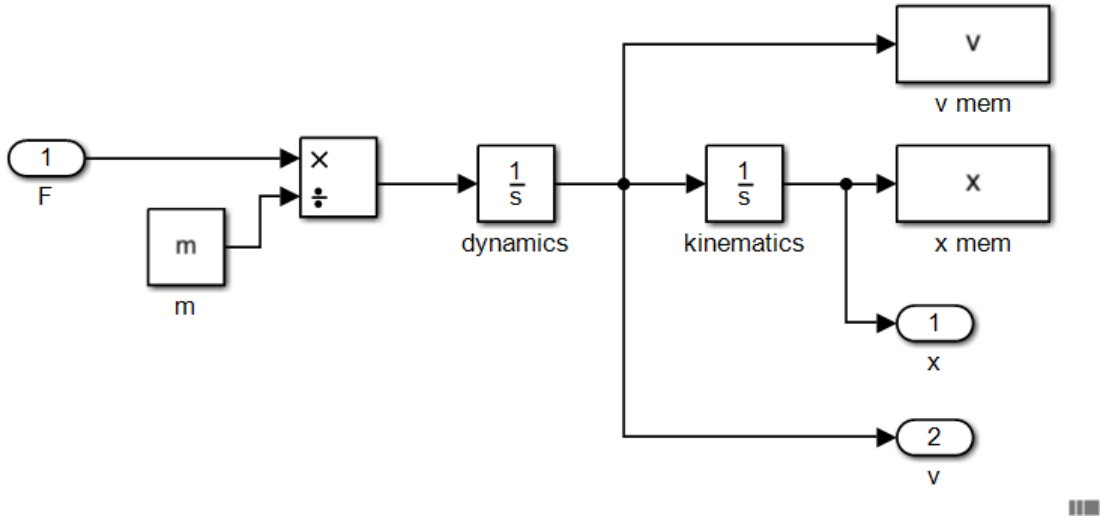


图 6.3: 被控对象原理框图

模型头文件 plant.h 中的内容：

```
#ifndef PLANT_H__
#define PLANT_H__
//====================================================================/
// A test case for fmi simulation tools
// Copyright (c) 2019 马玉海
// All rights reserved.
//
// Version 1.0
//====================================================================/

#include "interface.h"

#define FMI_MODEL_AUTHOR "MA Yuhai"
#define FMI_MODEL_NAME "plant"
```

(下页继续)

[2] 可参考示例模型 plant_1，查看构建后的代码及接口是否添加了期望的后缀。

(续上页)

```
#define FMI_MODEL_DISCRIPTION "a plant model"
#define FMI_PORT_POSTFIX ""    /* 指定端口后缀，避免在某些仿真工具中的命名冲突 */

// resource file definition if any
#define FMI_RESOURCE_ITEM 0    /* 外部资源文件数量为零，列表被忽略 */
#if FMI_RESOURCE_ITEM>0 && defined EN_RES_ACCESS
const char *resource_file_list[FMI_RESOURCE_ITEM] = {
    };
#endif

#define FMI_TASK_ITEM 0    /* 定时任务数量为零，触发函数被忽略 */
// task definition if any in the unit of [ms]
void task_30ms_start_0ms(void);

// define interface variables by an fmi object
// one statement per line, no extra semicolons allowed
// do not modify internal variables
typedef struct st_fmi_object_t{
    // internal variables
    FMI_IN double fmi_time_current;
    FMI_IN double fmi_time_step;
#if FMI_RESOURCE_ITEM>0
    FMI_PRM fmi_str_ptr fmi_file_list [FMI_RESOURCE_ITEM]; // do not delete the␣
↪spaces around []
#endif

    // interface variables
    FMI_IN Stru_Data_Controller_To_Plant st_data_controller_to_plant;
    FMI_OUT Stru_Data_Plant_To_Controller st_data_plant_to_controller;
}st_fmi_object;

#endif // PLANT_H__
```

模型源文件 plant.cpp 中的内容：

```
#include "plant.h"

double x;
double v;
double F;

void* fmi_instantiate(void)
{
    st_fmi_object *p =
        (st_fmi_object *)calloc(1, sizeof(st_fmi_object));
    if (!p) {
        fprintf(stderr, "fmi_instantiate failed in model plant!\n");
        exit(EXIT_FAILURE);
    }

    return p;
}

int fmi_initialize(void *fmi_object)
{
    st_fmi_object *p = (st_fmi_object *)fmi_object;
```

(下页继续)

(续上页)

```

    x = p->st_data_controller_to_plant.x_0;
    v = p->st_data_controller_to_plant.v_0;

    return 0;
}

int fmi_doStep(void *fmi_object)
{
    st_fmi_object *p = (st_fmi_object *)fmi_object;
    const double m = 0.1;

    F = p->st_data_controller_to_plant.F;

    v += F / m * p->fmi_time_step;    /* 当前时间、步长会由仿真工具更新 */
    x += v * p->fmi_time_step;

    p->st_data_plant_to_controller.x = x;
    p->st_data_plant_to_controller.v = v;

    return 0;
}

int fmi_reset(void *fmi_object)
{
    st_fmi_object *p = (st_fmi_object *)fmi_object;
    IO_PORT_FLUSH(Stru_Data_Plant_To_Controller, st_data_plant_to_controller);
    return 0;
}

void fmi_freeInstance(void *fmi_object)
{
    st_fmi_object *p = (st_fmi_object *)fmi_object;

    free(p);
}
```

# 7 S-Function 接口及其实现

生成 fmu 的过程中，在模型的 sources 目录中也生成了支持 Simulink 导入的 S-Function 接口代码。

若要生成模型的 S-Function 模块，在顶层 SFcnLists.m 中”model_list = { ‘controller’ , ‘plant’ }; % add model to this list” 语句处，将新的模型添加在列表中。

启动 MATLAB，将工作路径切换至工具包根目录，运行 SFcnLists.m 脚本，将执行 S-Function 的代码生成和模块构建。

保存获得的 Simulink 模型模块，以及工作空间中的数据总线定义，分发模型时还需要附加 *.mexw32/*.mexw64 二进制文件，以及模型所需的数据文件。

下面的示例仍以教程中的模型为例，演示使用 S-Function 模块进行集成仿真的具体过程。

- 运行 SFcnLists.m 脚本后，将一一生成列表中模型对应的 S-Function 模块，如 图 7.1 。

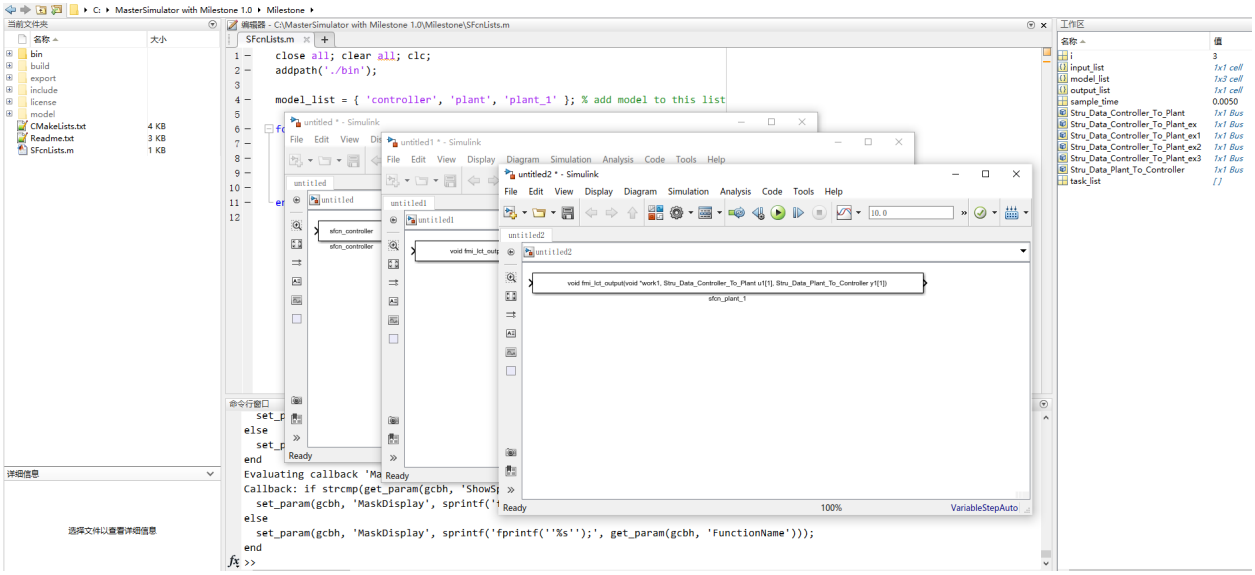


图 7.1: 生成的 S-Function 模块

- 在 MATLAB 的工作空间 (Workspace) 中，将生成的总线 (Bus) 定义备份，如 图 7.2 。

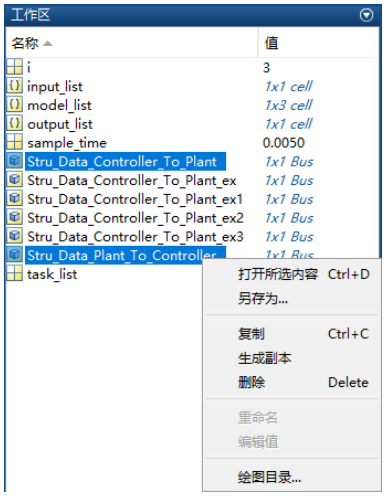


图 7.2: 工作空间中的总线数据定义

- 在模型对应的代码目录 (sources) 下，将生成的 S-Function 所对应的二进制文件一一备份，如 图 7.3 。
- 在 Simulink 中将多个 S-Function 模型连接为仿真工程，并显示其间的总线数据流，如 图 7.4 。
- 此时若直接运行仿真工程，可能出现代数环错误，如 图 7.5 。
- 在 Simulink 模块库中，找到并添加 “*Memory*” 模块到工程中，如 图 7.6 。
- 在 Simulink 模块库中，找到并添加 “*Scope*” 模块到工程中，如 图 7.7 。
- 在 Simulink 模块库中，找到并添加 “*Bus Selector*” 模块到工程中，如 图 7.8 。
- 将 Bus Selector 模块拖放到 plant 输出的数据总线上，点击选择要析取的信号，如 图 7.9 。
- 打开 Simulink 的仿真求解器配置页面，设置定步长求解器，步长 0.005s，如 图 7.10 。
- 将备份的总线数据结构文件 (.mat) 导入到工作空间中，并将 S-Function 的二进制文件复制到工程当前路径下，如 图 7.11 。
- 将模型依赖的其他数据文件复制到工程当前路径下，如 图 7.12 。
- 配置好的 Simulink 仿真工程及仿真结果，如 图 7.13 。
- 模型中打印到标准输出的调试信息，被重定向至 Simulink 状态栏中部的诊断监视窗口，如 图 7.14 。

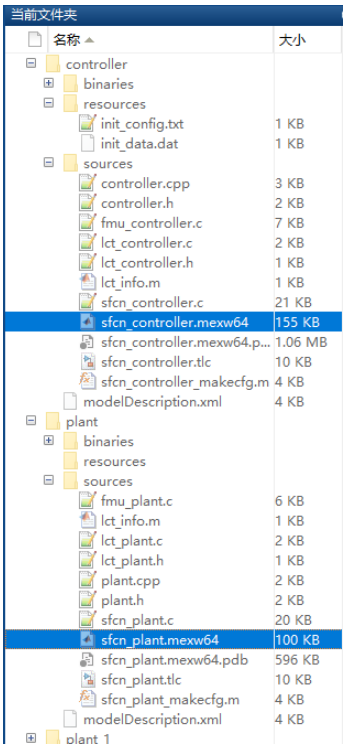


图 7.3: S-Function 所对应的二进制文件

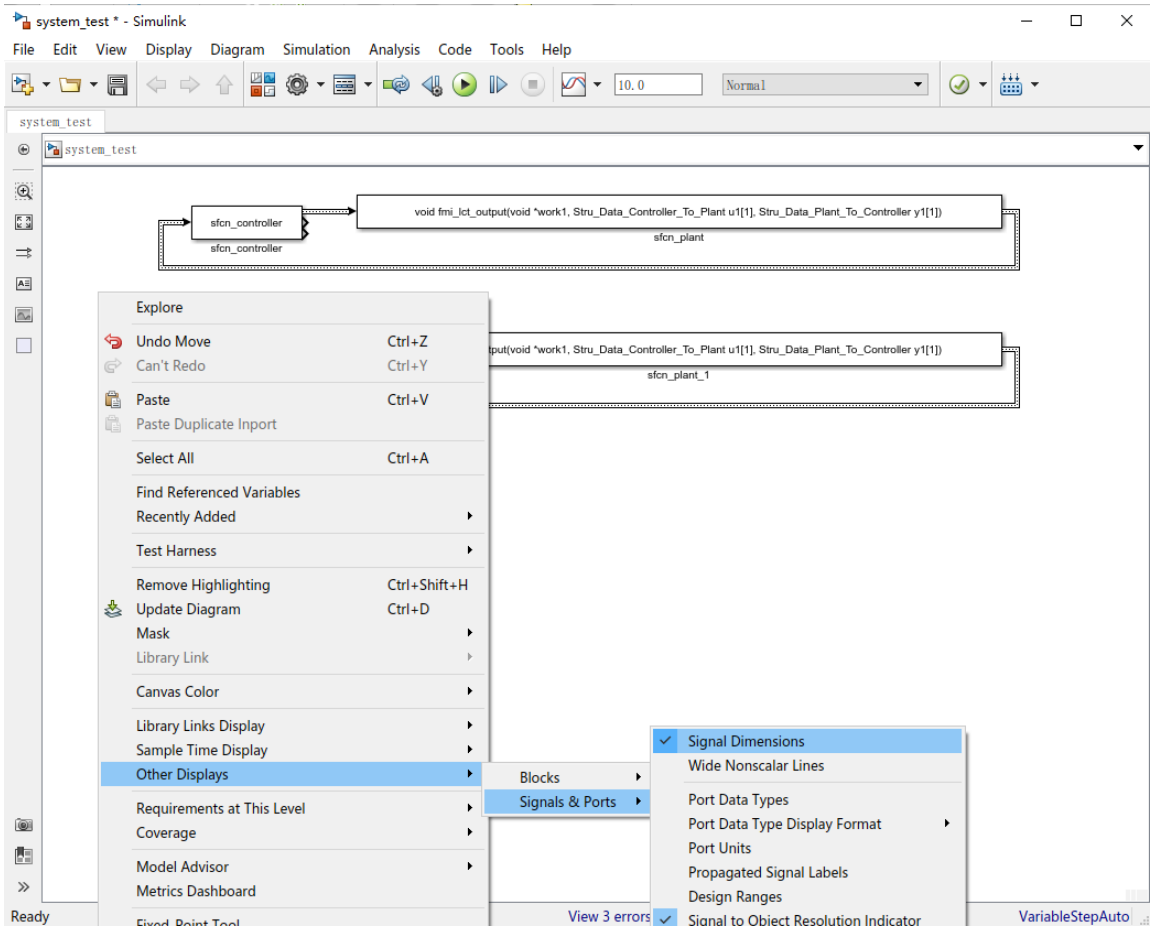


图 7.4: Simulink 中集成生成的 S-Function 模型

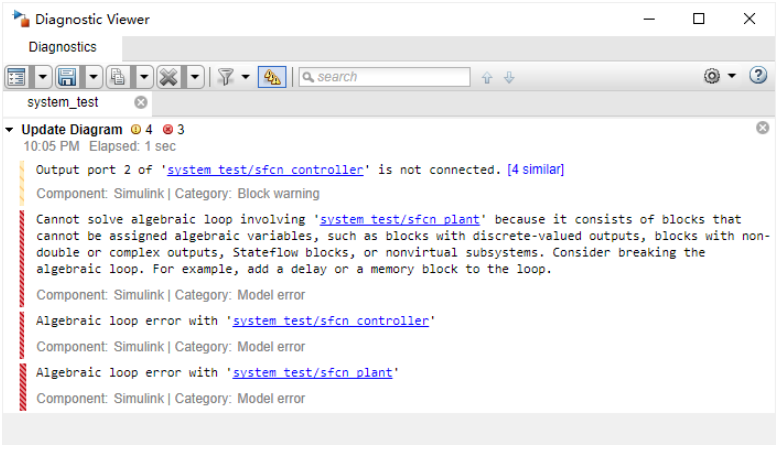


图 7.5: 代数环错误

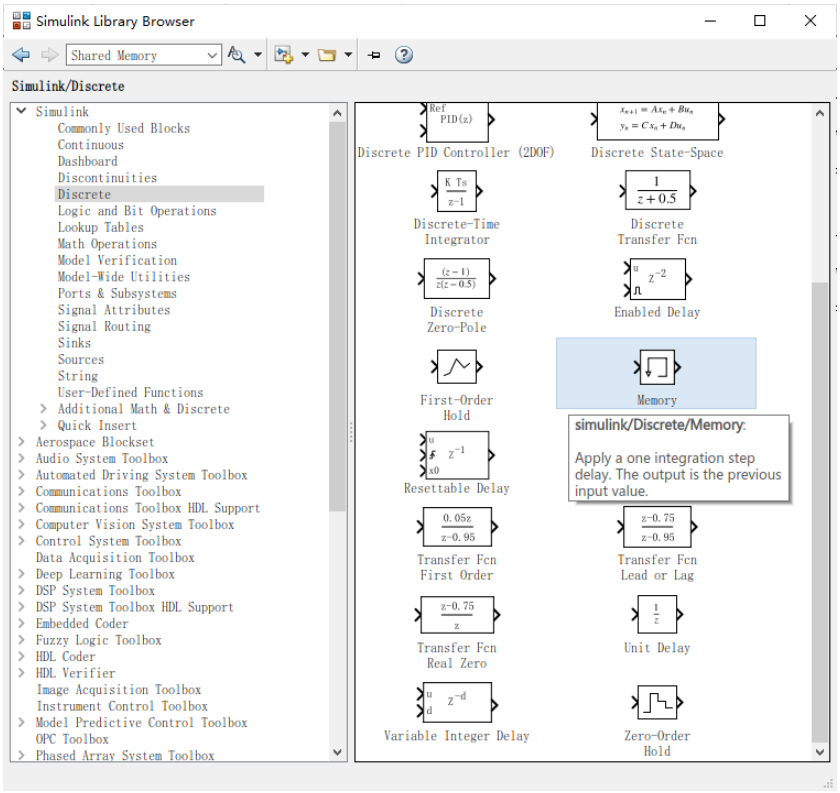


图 7.6: Memory 模块

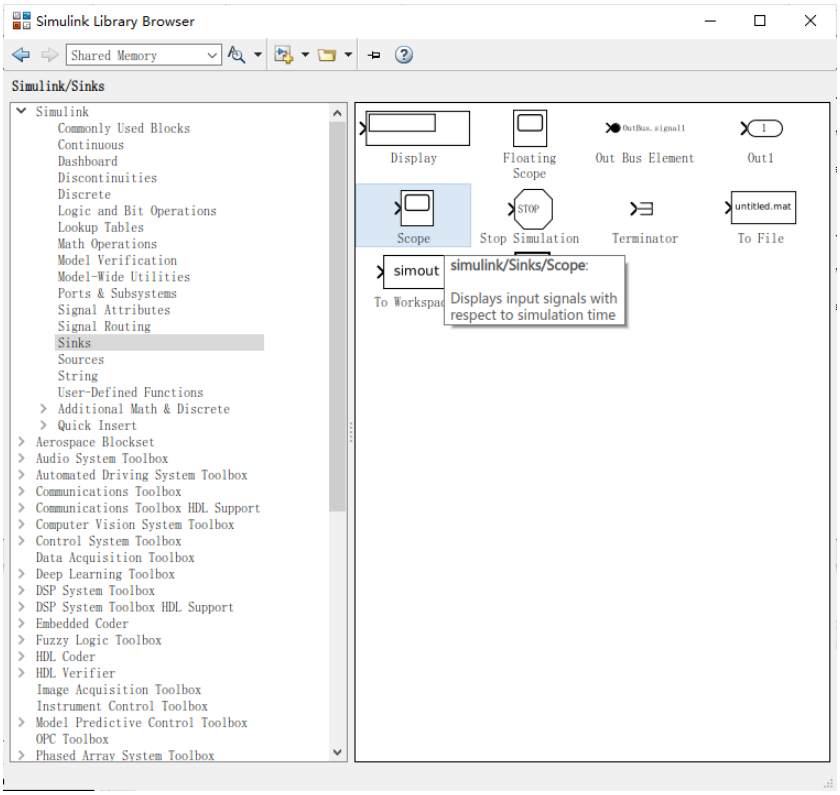


图 7.7: Scope 模块

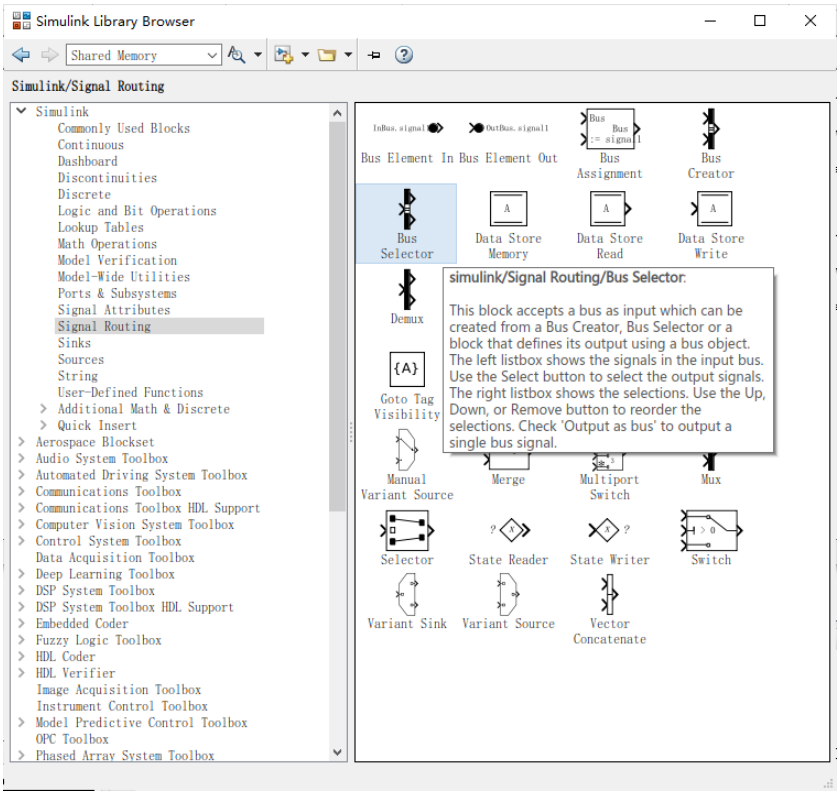


图 7.8: Bus Selector 模块

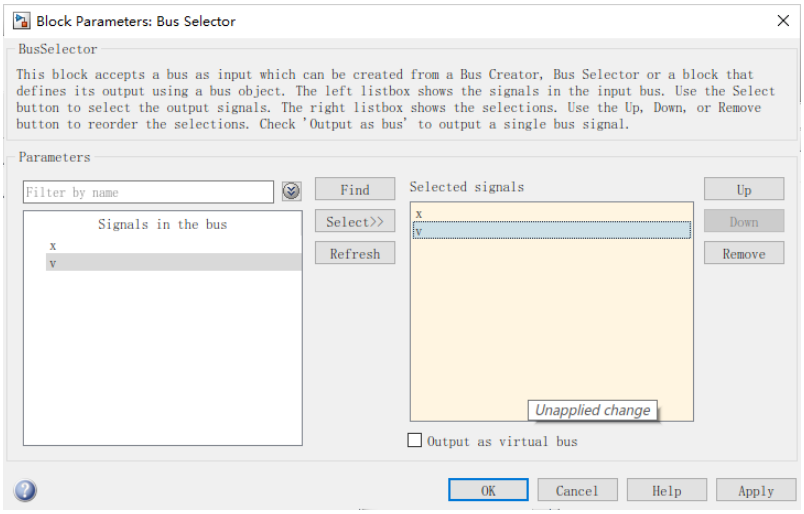


图 7.9: 通过 Bus Selector 析信号

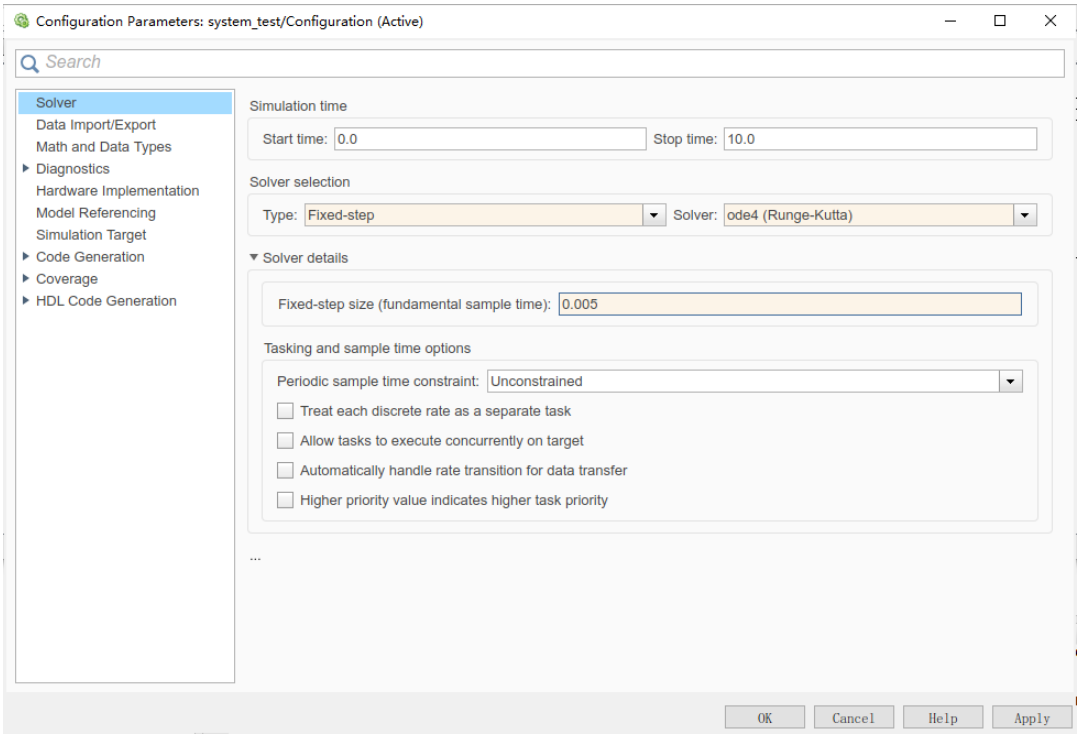


图 7.10: 仿真求解器配置

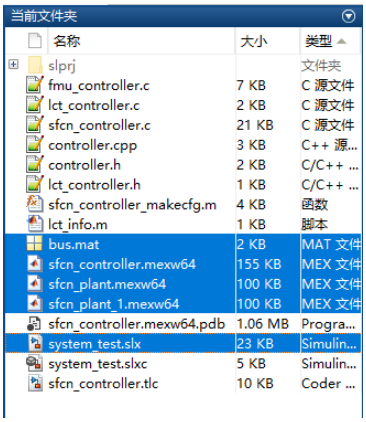


图 7.11: 准备总线及二进制文件

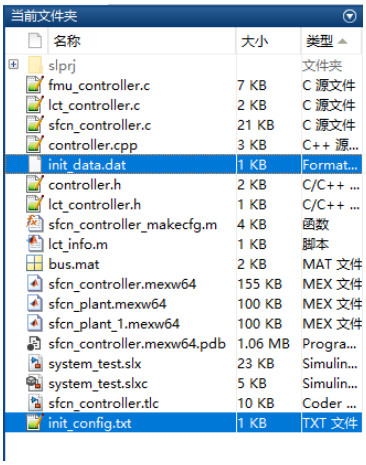


图 7.12: 准备数据文件

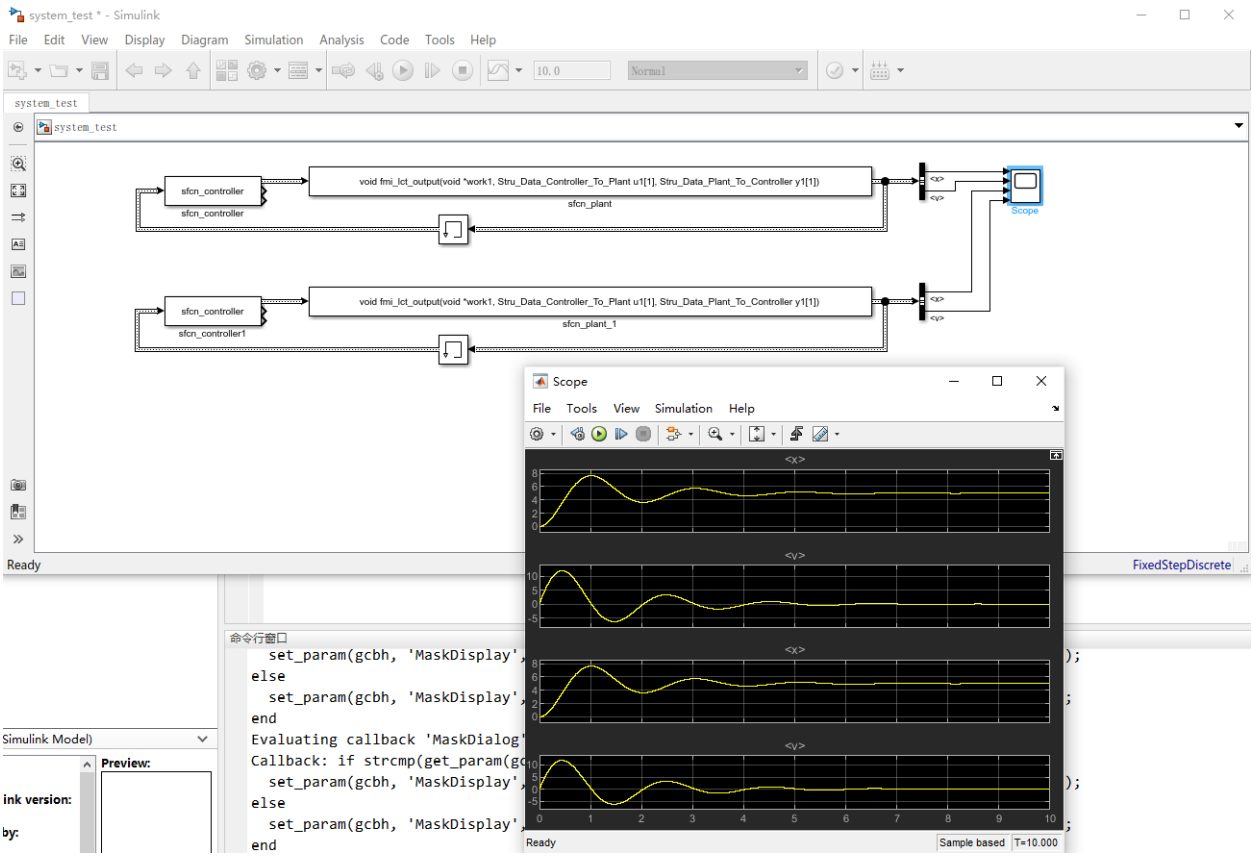


图 7.13: Simulink 仿真结果

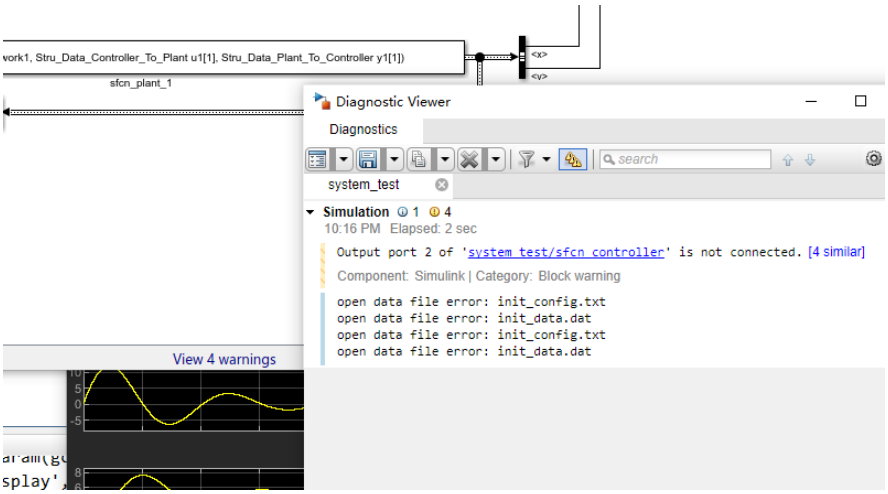


图 7.14: Simulink 诊断监视窗口

# 8 系统实现细节

## 8.1 功能及组成

仿真模型接口适配开发系统主要由向导式图形界面（简称界面程序）与接口解析及代码生成工具（简称接口程序）构成，并借助系统提供的代码编辑工具，编译构建工具（典型的如 make，nmake 或 CMake 等），C/C++ 编译环境等通用组件完成 FMU 模型及 S-Function 模型（统称为模型）的接口适配工作；对于生成 FMU 模型，还需要系统管理工具与 FMI 测试工具；对于生成 S-Function 模型，还需要 MATLAB/Simulink 软件。

具体的，系统组成关系如 图 8.1 。其中向导式图形界面主要由公用数据结构定义文件解析（用于提供用户可选择的数据类型）、用户模型定义文件解析（用于重新编辑用户已经配置过的模型）、FMU 构建脚本生成、S-Function 构建脚本生成以及模型用户代码模板生成等部分组成；此外还包括若干通用界面组件，主要有可交互表格控件、多语言界面样式及布局、操作信号传递与处理、系统配置持久化、外部进程调用及监测以及模型目录状态监测等。接口解析及代码生成工具主要由用户模型定义文件解析、公用数据结构定义文件解析、FMI 接口代码文件生成、FMU 描述文件生成、LCT 接口代码文件生成以及 LCT 模型定义文件生成等部分组成。

## 8.2 运行流程

系统架构及工作流程如 图 8.2 所示。具体的，系统的工作步骤为：

Step-1：用户操作界面程序，可以创建或编辑已有的模型定义文件内容，包括模型信息定义，外部资源定义，定时任务定义以及接口数据定义。其中模型信息包括模型名称，作者，备注，及端口后缀等；对于外部资源定义，包括路径，文件名，后缀名等；对于定时任务定义，包括任务周期，起始时间偏移等；对于接口数据定义，包括输入、输出及参数的数据类型和变量名称，用户可以直接编辑录入，也可以加载公用数据结构定义文件，界面程序解析其中的 C 语言结构体定义，并形成可供用户选择的数据结构列表，用户可以拖拽列表中的项目，将其添加至模型的接口数据定义区域内。

Step-2：用户填写完界面程序中的信息后，可以操作生成模型用户代码，包括用户模型定义文件及用户模型实现文件。其中用户模型定义文件完全存储了界面程序中的信息；用户模型实现文件则给出了模型受仿真程序调用运行时的接口函数模板，包括对应于 FMI 标准接口中的实例化代码（在模型加载时执行），初始化代码（在模型处于复位状态时执行），步进代码（每次仿真时间推进时执行），重置代码（将模型切换至复位状态时执行），以及终止代码（在模型退出时执行）；此外，还根据外部资源及定时任务定义，自动拓展给出了外部资源引用及定时任务处理的接口。用户可以操作界面程序，设置关联的代码编辑工具，在界面程序中可调用代码编辑工具，填写接口函数模板，即可完成模型代码内容的实现。

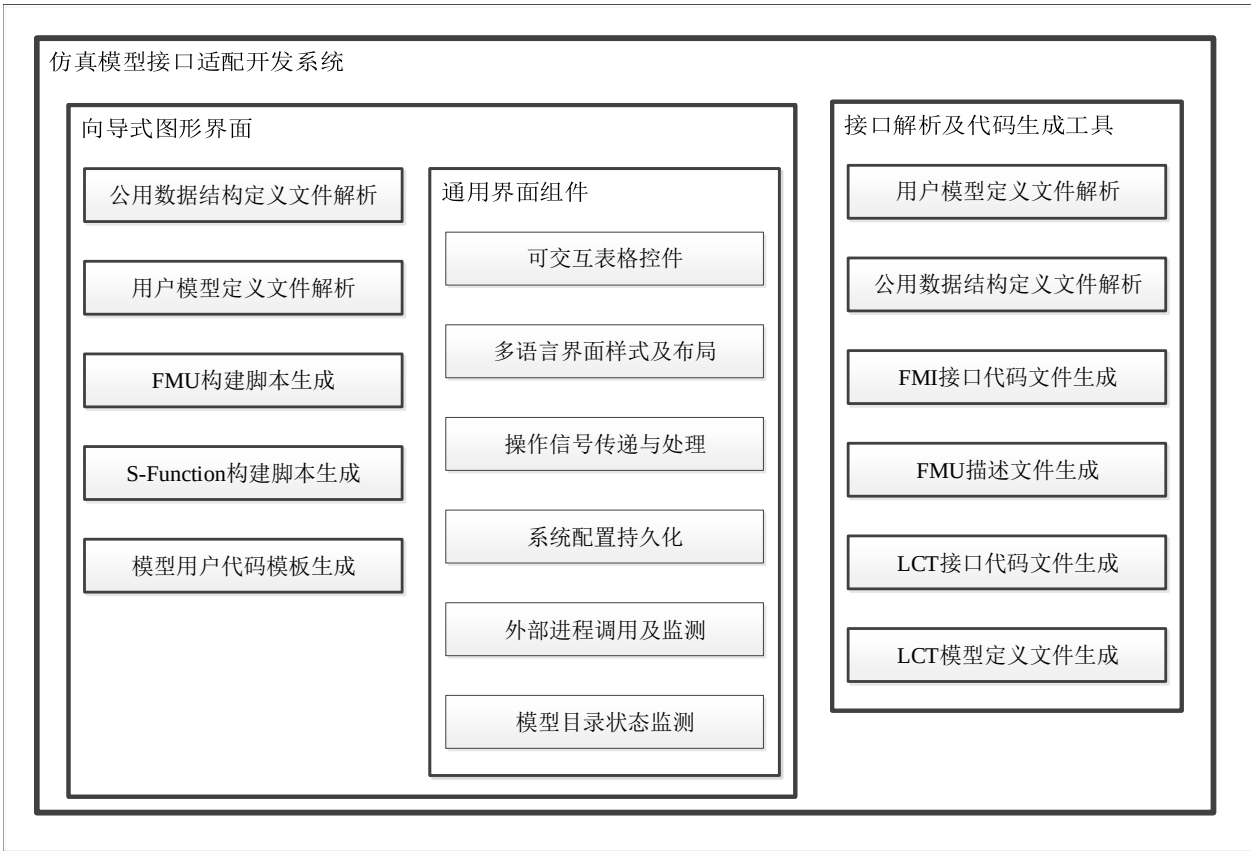
仿真模型接口适配开发系统
向导式图形界面
公用数据结构定义文件解析
用户模型定义文件解析
FMU构建脚本生成
S-Function构建脚本生成
模型用户代码模板生成
通用界面组件
可交互表格控件
多语言界面样式及布局
操作信号传递与处理
系统配置持久化
外部进程调用及监测
模型目录状态监测
接口解析及代码生成工具
用户模型定义文件解析
公用数据结构定义文件解析
FMI接口代码文件生成
FMU描述文件生成
LCT接口代码文件生成
LCT模型定义文件生成

图 8.1: 系统组成关系

Step-3：用户可操作界面程序，调用编译构建工具，完成编译过程。界面程序生成 FMU 构建脚本（典型的如 Makefile 或 CMakeLists.txt）以及 S-Function 构建脚本（MATLAB 语言）；FMU 构建脚本以及 S-Function 构建脚本均支持多个模型的批处理操作，界面程序在启动编译操作前，提示用户选择当前需要操作的模型。编译过程中，编译构建工具调用接口程序，根据用户模型定义文件中的接口数据定义，查询公用数据结构定义文件中的已有模型接口数据结构；接口程序递归地解析用户模型定义文件中的接口数据定义，建立已有模型接口数据结构与 FMI 中所需的一维展开的接口变量间的映射关系，从而能够自动填写 FMI 接口代码模板、FMI 描述文件模板，生成 FMU 模型对应的 FMI 接口代码文件以及 FMU 描述文件。同样的信息也用于自动生成 S-Function 模型对应的 LCT 接口代码文件以及 LCT 模型定义文件。

Step-4：用户可操作界面程序，调用编译构建工具，完成构建过程。构建过程中，编译构建工具首先调用系统管理工具，创建或清理出 FMU 模型目录结构；然后，调用 C/C++ 编译环境，完成 FMU 模型对应的 FMI 接口代码文件、FMI 标准通用文件（包含适应多种系统环境的代码，不需要用户修改）与用户模型实现文件的编译与链接，生成 FMU 动态链接库（在不同系统环境下生成对应版本的文件，如.dll/.so/dynlib 等，支持多文件共存），并将过程中可能的调试信息提示给用户；之后，将 FMU 模型所需的 FMU 动态链接库、FMU 描述文件、FMU 资源文件以及 FMU 源代码文件等汇集到 FMU 模型目录结构中，完成压缩打包工作；最后，调用 FMI 测试工具，尝试仿真所生成的 FMU 模型文件，将测试运行结果打印反馈给用户。

Step-5：用户可以启动 MATLAB 软件（需要带有 LCT 模块并配置了 C/C++ 编译环境），执行所生成的 S-Function 构建脚本，能够根据已生成的 LCT 接口代码文件以及 LCT 模型定义文件，创建 S-Function 动态链接库、总线定义文件、Simulink 模块以及 Target Link Compiler, TLC 脚本文件。

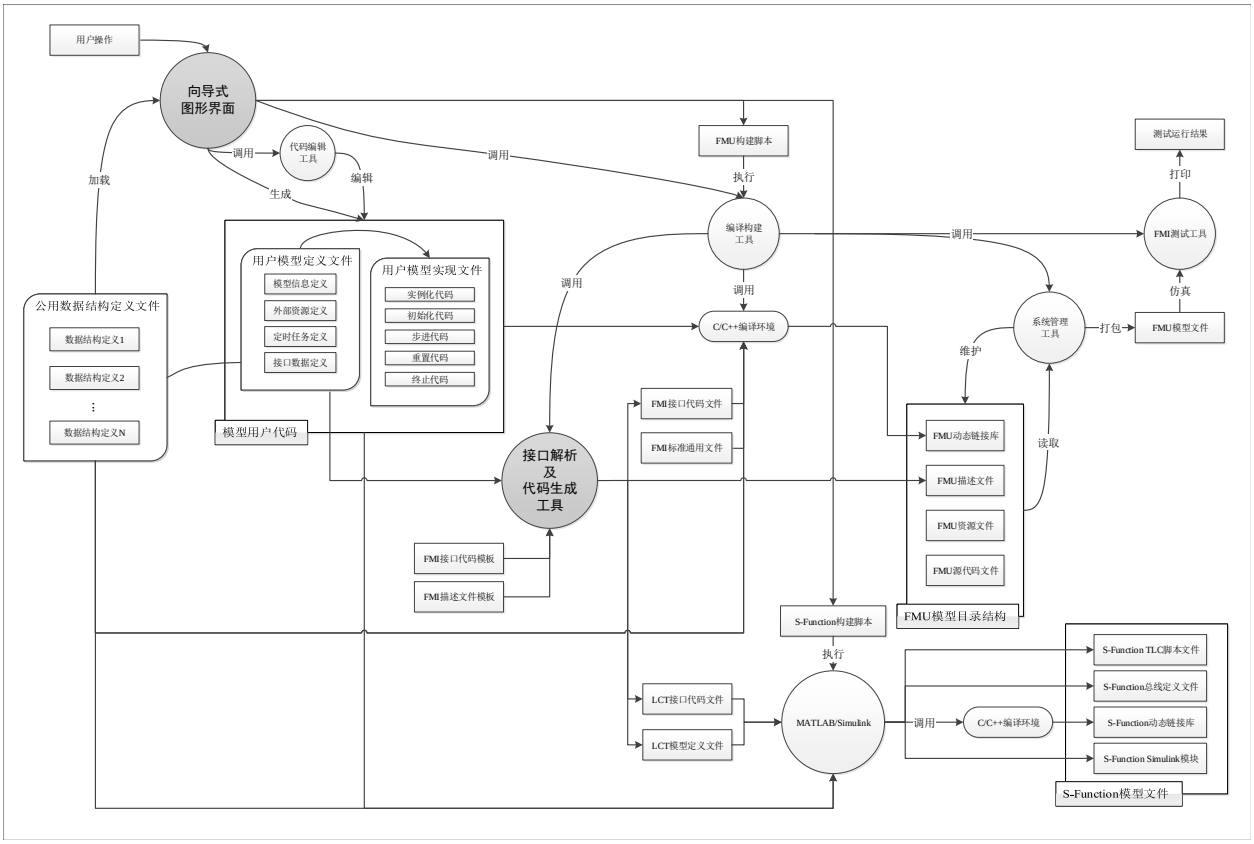


图 8.2: 系统架构及工作流程

## 8.3 详细目录结构

```
MASTERSIMULATOR WITH MILESTONE 1.0   <<-------- MasterSim 根目录
|  MasterSimulator.exe      <<-------- MasterSim 求解器
|  MasterSimulatorUI.exe    <<-------- MasterSim 主界面
|  ...
|
├─Milestone             <<-------- Milestone 根目录
|   |  CMakeLists.txt        <<-------- Milestone FMU 构建脚本
|   |  Readme.txt            <<-------- Milestone 快速使用指南
|   |  SFcnLists.m           <<-------- Milestone S 函数构建脚本
|   |
|   ├─bin
|   |      milestone         <<-------- Milestone 核心程序 (Linux)
|   |      milestone.exe     <<-------- Milestone 核心程序 (Windows)
|   |      ...
|   |
|   ├─export
|   |   |  fmu_controller.fmu <<-------- 导出的模型 FMU 文件
|   |   |  fmu_plant.fmu
|   |   |  fmu_plant_1.fmu
|   |   |
|   |   └─mastersim          <<-------- MasterSim 测试工程目录
|   |       |  sim.bm
|   |       |  sim.msim
|   |       |
|   |       └─sim
|   |           └─results
|   |                   values.csv    <<-------- MasterSim 测试数据
|   |
|   ├─include               <<-------- FMI 标准头文件、模板文件
|   |
|   ├─license
|   |      00-0C-29-19-A4-33.lic     <<-------- 单机授权文件
|   |      00-0C-29-7E-DB-12.lic
|   |      ...
|   |
|   └─model                 <<-------- 模型目录
|       |  interface.h              <<-------- 全局接口头文件
|       |
|       ├─controller                <<-------- 示例模型: 控制器
|       |  ├─resources              <<-------- 示例模型数据文件
|       |  |      init_config.txt
|       |  |      init_data.dat
|       |  |
|       |  └─sources
|       |         controller.cpp    <<-------- 示例模型代码文件
|       |         controller.h      <<-------- 示例模型头文件
|       |
|       ├─plant                     <<-------- 示例模型: 被控对象
|       |  └─sources
|       |         plant.cpp
|       |         plant.h
```

(下页继续)

(续上页)

```
│       │
│       └─plant_1                    <<-------- 示例模型: 被控对象 1
│           └─sources
│                   plant_1.cpp
│                   plant_1.h
│
├─...
```