JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

A Tool for Offline Debugging and Trace Visualization of SoftPLC Programs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Diplomstudium

Mechatronik

Eingereicht von:
Richard Berger, 0555165

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
o.Univ.-Prof. Dr. Dr. h.c. Hanspeter Mössenböck

Mitwirkung:
Dipl.-Ing. Dr. Herbert Prähofer

Leonding, Jänner, 2012

Abstract

This thesis describes the *Trace Visualization Tool* developed at the Christian Doppler Laboratory for Automated Software Engineering. It was part of a larger project, Capture & Replay, whose goal is the development of methods and tools for recording real-time PLC applications for deterministic replay, off-line debugging, and dynamic program analysis. The project is conducted in cooperation with and funded by KEBA AG.

This tool allows PLC developers to explore and analyse program recordings using a variety of different views. Each view shows a different level of detail, allowing the user to dive into the program from high-level views to a low-level views. Unlike a regular single-step debugger, it allows navigating the entire program run and examining the program's state at any point in time.

The thesis describes the different views provided by the tool, gives an introduction of how they are used for debugging, and outlines their implementation using the Windows Presentation Foundation. It also describes the main challenges of handling the huge amount of data and preparing it for display. Finally, a case study is presented which was developed to validate both the Capture & Replay approach and the visualization tool.

Kurzfassung

Diese Diplomarbeit beschreibt das *Trace Visualization Tool*, welches am Christian Doppler Labor für Automated Software Engineering entwickelt wurde. Es ist Teil eines größeren Projekts, Capture & Replay, welches die Erstellung von Methoden und Werkzeuge zur Unterstützung der SPS-Programmierung zum Ziel hat. Diese Werkzeuge umfassen das Aufzeichnen, deterministische Wiederabspielen und Off-Line-Debuggen, sowie eine dynamische Programmanalyse. Das Projekt wurde vom Unternehmen KEBA AG gefördert und in Kooperation mit diesem durchgeführt.

Dieses Werkzeug erlaubt es SPS-Programmierern Programmaufnahmen mit mehreren Ansichten zu analysieren. Jede dieser Ansichten zeigt einen anderen Detailgrad. Das Verhalten des Programms kann über High-Level-Ansichten sichtbar gemacht und über Low-Level-Ansichten genauer untersucht werden. Im Gegensatz zu Single-Step-Debuggern erlaubt es den gesamten Programmlauf nachzuvollziehen und den Zustand des Programms zu jedem Zeitpunkt zu untersuchen.

Diese Arbeit beschreibt die verschiedenen Ansichten des Werkzeugs, erklärt wie diese zum Auffinden eines Fehlers verwendet werden können und geht auf deren Implementierung mit der Windows Presentation Foundation ein. Sie beschreibt auch die Herausforderungen beim Arbeiten mit großen Datenmengen und deren Visualisierung. Abschließend beschreibt sie eine Fallstudie, die entwickelt wurde, um sowohl Capture & Replay als auch das Visualisierungswerkzeug zu validieren.

Acknowledgements

I want to thank everyone that supported me during my work on this thesis and the six years of study at the Johannes Kepler University.

First of all, I want to thank my adviser Prof. Hanspeter Mössenböck and the project lead Dr. Herbert Prähofer for offering me the opportunity to work on this project and their continued support during my work.

I would also like to thank my colleagues, Christian Wirth and Roland Schatz, with whom I had the pleasure working with. Without their excellent preliminary work on Capture & Replay, my thesis would not have been possible. Our in-depth and sometimes even heated debates have shaped the outcome of this work. Both were kind enough to read parts of my work and point out errors and mistakes. I would especially like to thank Christian Wirth for his never ending wishlist of features, discovered bugs and good ideas. I also have to thank Roland Schatz for continuously helping me improve my developer skills.

Special thanks go to the KEBA employees associated with the project "Testing and diagnosis of programmable logic controller programs", in particular Dr. Ernst Steller and DI Gottfried Schmidleitner, for supporting my work.

Last, but not least, I want to thank my family for their continuous support during my studies. I thank my parents for enabling me to study at university and allowing me to pursue my dreams. Finally I thank my sister for keeping an eye on her big brother, enduring my ups and downs, and helping me in so many different ways.

Richard Berger

This work has been conducted in the module "Testing and diagnosis of programmable logic controller programs" at the Christian Doppler Laboratory for Automated Software Engineering in cooperation with KEBA AG. It has been funded by KEBA AG and the Christian Doppler Forschungsgesellschaft.

Contents

Chapter 1

Introduction

This thesis presents results of work conducted at the Christian Doppler Laboratory for Automated Software Engineering (http://ase.jku.at) at the Johannes Kepler University in Linz, Austria. It is part of a larger project funded by and conducted in cooperation with our industrial partner KEBA AG (www.keba.com). KEBA provides a wide range of solutions for the automation industry, in particular, a hardware and software platform for developing automation solutions.

In this project (see [11], [9], [19] and http://ase.jku.at/modules/ for details) we are developing methods and tools for debugging and fault diagnosis of PLC (programmable logic controller) programs written in the IEC 61131-3 [7] language standard. Analysing the dynamic behaviour and finding bugs in PLC applications has been shown to be extremely complicated and costly and satisfactory solutions do not exist yet [11]. Debugging software systems depends on the ability to reproduce the execution run which led to a bug. This is difficult to achieve for PLC programs, the reasons being manifold [11]: (1) PLC programs are strongly determined by the physical environment on which they run, which is difficult or even impossible to reproduce in a test setting; (2) bugs often occur sporadically and only in field operation where debugging and analysis facilities are not available; (3) PLC programs are often subject to further sources of non-determinism, e.g., they may depend on the task scheduling of a real-time operating system. Moreover, PLC programs are long-running applications which exhibit complex behaviour and usually produce a huge amount of data, which makes it hard to locate a bug, even when debuggers are available. As of today, it is often necessary to send trained personal to investigate a software issue at a production site. There, they must perform detailed analysis, measure sensor values, and try to reconstruct a failure situation.

Our project's goal therefore is to improve this situation by allowing developers to look at software failures without having to reproduce initial and boundary conditions of the system. First, a deterministic replay debugging approach [11] has been developed which allows recording an application run in the field so that it can be deterministically replayed offline in a development system for debugging purposes. Then, based on the replay technology, a multi-level approach [9], [19] has been developed for extracting and abstracting the behaviour of control programs

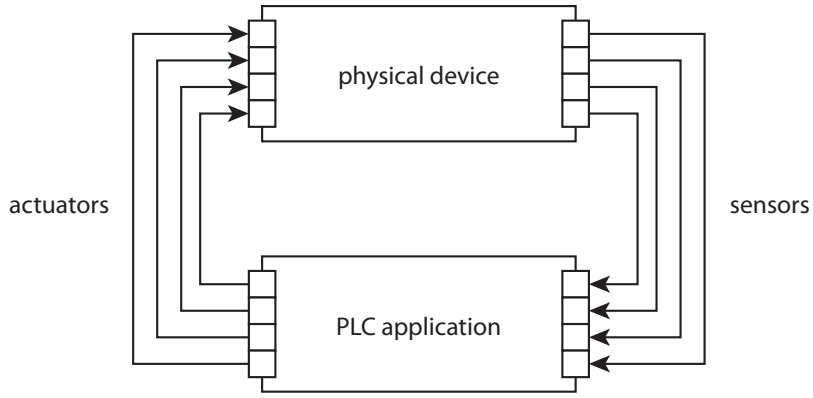


Figure 1.1: Control-Feedback Loop

from execution traces with the goal to support program comprehension and defect localization. In the specific work presented in this thesis a trace visualization tool, which allows visualizing control program runs at multiple levels of detail, has been developed.

1.1 PLC Applications

Before diving further into the trace visalization tool, its necessary to explain PLC applications and their programming model. PLC applications based on the IEC 61131-3 standard [7] implement control loops for physical devices. Such a device is controlled through multiple actuators and provides feedback about its physical state through various sensors (see Figure 1.1). A PLC program monitors this state and reacts accordingly by adjusting the control outputs to achieve its control objectives.

PLC programs are usually made up of multiple concurrent tasks. Each task runs a task program in a cyclic or event-based way. Event-based tasks are executed after a specific event, such as a variable changing its value. Cyclic tasks are executed at regular time intervals. This cycle time is usually specified in milliseconds.

Since such tasks are running in hard real-time, it is required that their programs must complete within their cycle time. Such programs therefore should not contain any blocking calls or long loops. Their primary structure is a set of branches which determine a task's control flow during each cycle.

Communication between tasks and hardware is done through global variables. The hardware addresses of actuator outputs and sensor inputs are mapped to global variables and can be used by any task.

1.2 Capture & Replay

As of today, the main tools used to debug control software are single-step debuggers and simulators, variable traces and stack trace reports combined with large scale code reviews. But the main problem remains: the code running on a machine can not be debugged. To tackle this problem, a new approach to debugging has been developed. A running prototype was created at the Christian Doppler Laboratory for Automated Software Engineering. Figure 1.2 illustrates how this new approach, called Capture & Replay, works. The following sections discuss the different phases.

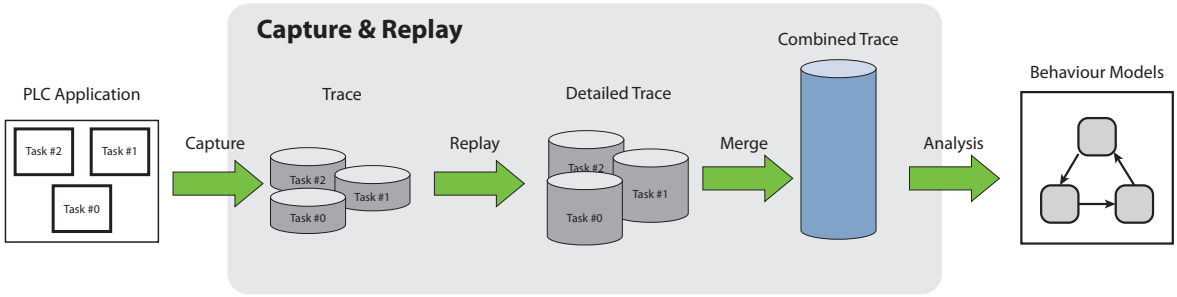


Figure 1.2: Data generation of Capture & Replay

1.2.1 Capture

The capture phase records a program run of the control application without violating real-time constraints. This is achieved by using an intelligent instrumentation of the source code, which is done transparently during compile time. The result of program recording is a trace file for each task which stores a list of trace operations.

A cyclic task running in trace mode stores timestamps at the beginning and end of each execution cycle. During such a cycle, important variable values are stored in the trace. Tasks are recorded independently and no scheduling information is saved. These operations are sufficient to deterministically replay the entire program.

1.2.2 Replay & Detailed Trace

The replay phase uses the recordings from the capture phase to reproduce the actual program run without needing access to the environment. In this phase, a different instrumentation is used to collect more information about the executions of task programs. For example, programs are instrumented to record all write operations to shared variables and to write a position mark at each branch. Using this trail of position marks, a deep understanding of the code execution is gained. In this phase, replay is still separate for each tasks and the result is a detailed recording of each task.

1.2.3 Merging & Combined Trace

Looking at single tasks gives us only half of the picture. What is really interesting is how different tasks interact with each other. In a third phase, all detailed trace files are merged into one big combined trace. In this phase the execution order of each trace operation is recreated. The new trace file includes task switch operations, which reflects where the task switches have happened in the real program run.

1.2.4 Trace Analysis

The merged trace now is used to analyse the program behavior. A main goal is to find high-level models and views of the program behaviour, which allows a user to get an overall understanding of a program run and to locate areas of interest. For example, in this phase the reactive behavior of a program is reengineered, which results in a transition model showing the different execution traces of a program run. This transition model is then further analysed to find recurring patterns of execution [9], [19].

1.3 Trace Visualization Tool

The Trace Visualization Tool is the topic of this thesis. It was created to process, transform and present data produced by Capture & Replay and trace analysis in an efficient and user friendly way. The primary challenge was coping with the great amount of data produced by control applications. Instead of focusing on low-level details of the trace data, high-level views highlighting different aspects of the reactive behaviour of the application were introduced. Using a top-down approach, users of this tool use high-level views to find the interesting regions of a recording. Lower-level views can then be used to examine details.

Figure 1.3 shows a screenshot of the tool and its most important views. Execution phases of the application recording are detected by pattern algorithms [9] and displayed along a timeline. In these phases, recurring patterns of the application's reactive behavior can be identified, as well as changes of the application state over time. At the lowest level, developers can instantly look at any line of the executed program at source code level, set bookmarks and jump between different locations. They can find out why variables have changed, plot their progression over time, and figure out how task interleaving is effecting program execution. Thus, this tool removes the necessity of recreating program runs through simulators, but instead allows instant and fluid navigation inside of large program recordings.

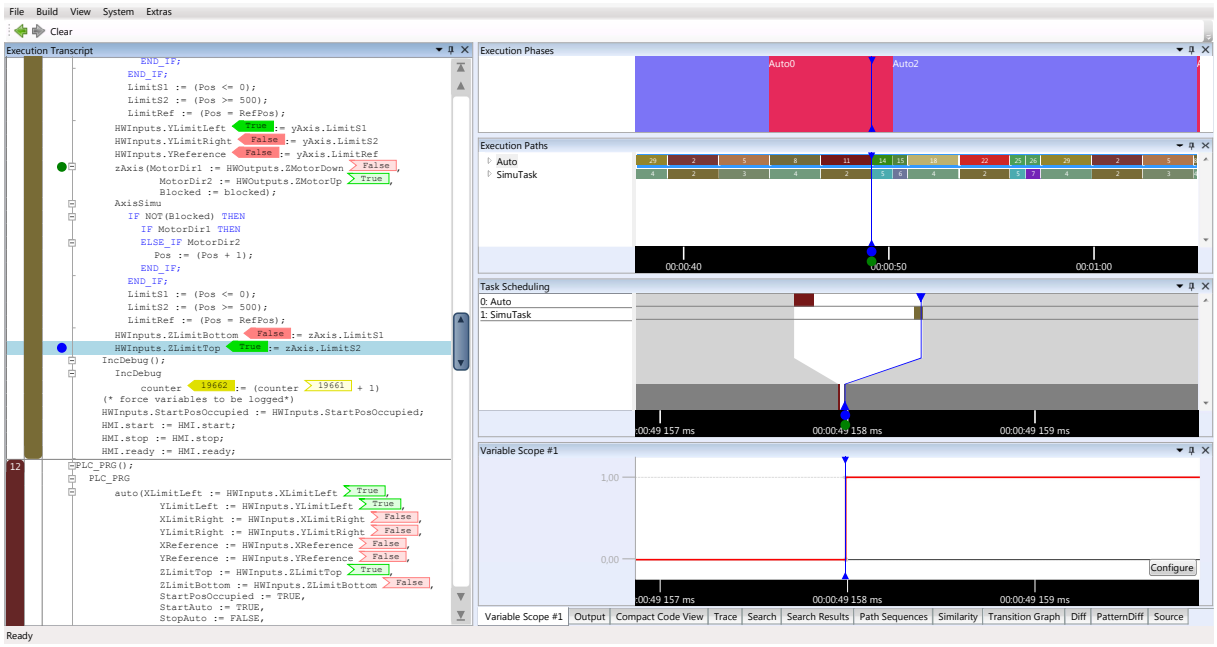


Figure 1.3: Screenshot of the Trace Visualization Tool

1.4 Outline of this thesis

This thesis is structured as follows:

Chapter 2 is a short introduction of SoftPLC applications and their overall structure.

Chapter 3 gives an overview of the trace visualization tool and its views and features. These are then used to find a defect in an application example. The chapter concludes with a description of the overall architecture of the tool.

Chapter 4 explains how data is processed and transformed. Further data virtualization is introduced and various implementation details explained.

Chapter 5 is a short introduction of how WPF programs are built and structured.

Chapter 6 shows the most important views of the Trace Visualization Tool and outlines how they are implemented.

Chapter 7 contains a bigger case study which has been developed in the course of this work for testing Capture & Replay and the Trace Visualization Tool.

Chapter 2

SoftPLC Applications

The entire project revolves around software written for programmable logic controllers (PLC). PLC Software is specifically designed to control machines through multiple digital and analog I/O ports. Furthermore, there is a distinction between traditional PLCs, which are simply microcontrollers wired up to I/O, and so called SoftPLCs. SoftPLCs run on industrial PC hardware and use a real-time operating system, such as Linux or Windows Embedded with Realtime Extensions. Our research focuses on SoftPLCs.

There are various ways to program such systems. However, to simplify the development, vendors have defined standards for PLC programming. One of these standards is IEC 61131-3 [7]. This standard defines programming languages for writing PLC programs. Vendors use this standard and provide software solutions around it. One of the well known implementations of the standard is Siemens S7 [10]. Another implementation comes from our industrial partner and is named KEBA Kemro IEC. This project uses yet another implementation of the standard, provided by a company named 3S (www.3s-software.com). Their software solution, called CoDeSys, provides an integrated development environment written in .NET. Besides Siemens S7, it is a very popular development platform for PLC applications based on the IEC 61131-3 standard.

2.1 Structure

Figure 2.1 shows the basic structure of PLC applications. They usually consist of one or more tasks which are executed in a cyclic or event-based way. Cyclic tasks are executed periodically using a specified cycle time. Event-based tasks react to variable changes, such as a Boolean flag which turns true. Each task's execution must complete within a given a time. This is a real-time constraint enforced by the runtime system. During each execution a task runs a list of task programs from beginning to end, updating internal variables and reading or writing global variables. This is done directly by the task program or indirectly by calling functions or function blocks.

Function blocks are used to create reusable components in an application. They are instantiated by creating a variable with a function block type. Each function block instance is allocated

when the application starts. Because PLC applications use a static memory model, no new objects are created at runtime.

Programs running in different tasks communicate through global shared variables. These can be of basic types (**`INT`**, **`REAL`**, **`BOOL`**, ...), complex structures or even function block instances. A primary challenge is keeping multiple tasks synchronized while accessing the same data.

Communication with hardware I/O is also done through global variables. They are mapped to specific memory locations in the hardware memory, which is written by sensors and read by actuators.

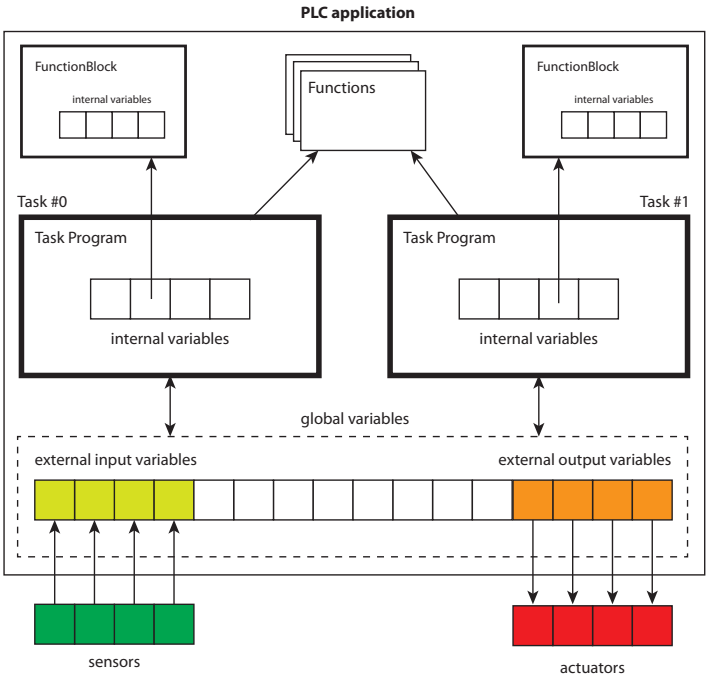


Figure 2.1: Structure of a PLC application

2.2 Program Object Units

Programs written using the IEC 61131-3 standard are structured into multiple program object units (POUs). A POU can be of one of the following types:

function A function is group of code which takes parameters and returns a result. It is stateless, which means its variable stack is purged once it finishes.

function block A function block can be used like a function, however, it does keep its state. Every execution can update this state. The output of a function block depends both on its parameters and its internal state.

program A program is a special type of function block which is only instantiated once. Usually a program calls other function blocks and functions to perform its duty. Programs are

executed by tasks. These are light weight processes run by the PLC application and form the entry point of any executed code.

2.2.1 Interface

POUs consists of two parts: an interface and implementation. The interface defines the variables of the POU. Listing 2.1 shows the interface of a simple function block named `CountUp`. Variables defined in a **`VAR_INPUT`** block are used as input parameters. **`VAR_OUTPUT`** variables are output parameters of the POU. Temporary variables, whose values should be purged after execution, are defined as **`VAR_TEMP`**. All other variables defined in a **`VAR`** block are part of the POU's state. As noted earlier, functions don't store any state. All their variables are temporary.

```
FUNCTION_BLOCK CountUp
VAR_INPUT
    maximum : INT;
    increment : INT;
END_VAR

VAR_OUTPUT
    result : INT;
END_VAR

VAR
    currentValue : INT;
END_VAR
```

Listing 2.1: Interface of a function block named `CountUp`

2.2.2 Implementation

The implementation of a POU is done using one of the textual or graphical languages defined by the IEC 61131-3 standard:

- Function Block Diagram (FBD)
- Instruction List (IL)
- Ladder Logic Diagram (LD)
- Sequential Function Chart (SFC)
- Structured Text (ST)

Our case studies, which are described in Section 2.3 and Chapter 7, only use ST, SFC and FBD implementations. Therefore, only these languages are described in the following sections.

Structured Text

Listing 2.2 shows the implementation of the simple counting function block. It is implemented as ST code, which is an imperative language similar to Pascal. In CoDeSys graphical languages are transformed into equivalent ST code at compile-time.

```
IF currentValue < maximum THEN
    currentValue := currentValue + increment;
    result := currentValue;
ELSE
    result := currentValue;
END_IF
```

Listing 2.2: Structured-Text implementation of a simple counter

Sequential Function Chart

SFCs are a graphical notation to define state machines. Rectangular shaped boxes represent the states of the function block. Each state can have multiple actions attached to it, which run depending on various conditions (N. . . every cycle, P . . . pulse, rising edge). Each action is a piece of code running in the scope of the function block. This means it has access to all function block variables. Figure 2.2. shows a simple SFC.

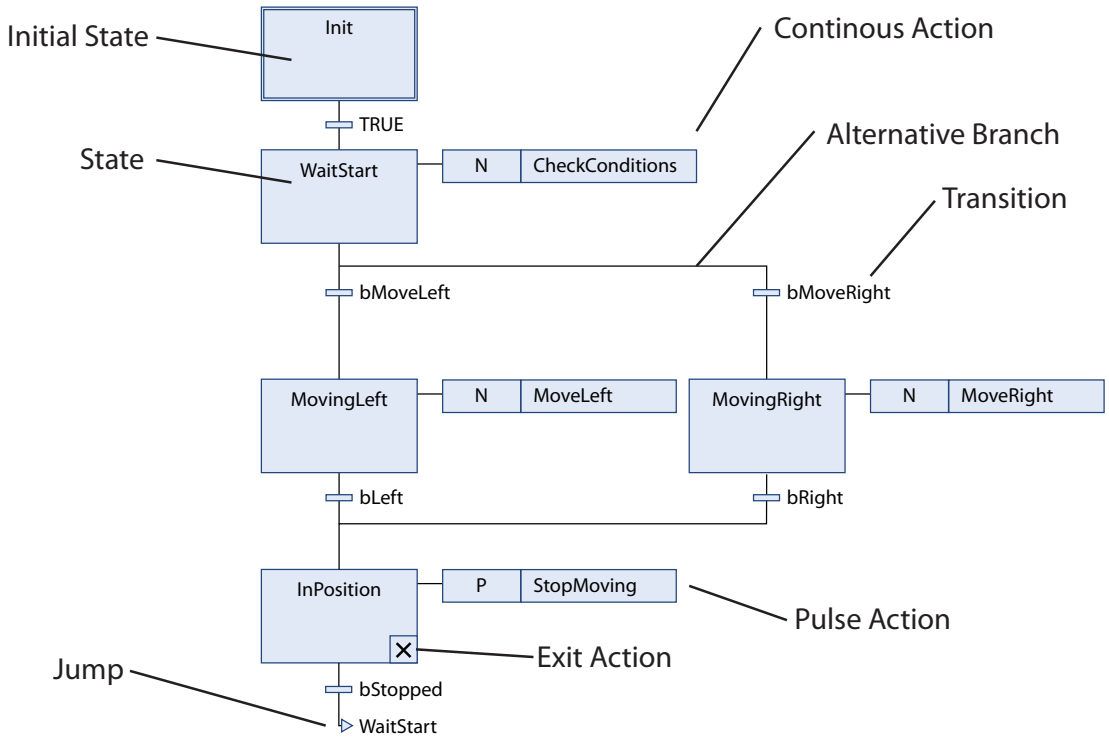


Figure 2.2: Simple SFC example

This SFC contains five different states. SFCs are executed from top to bottom. At the top is the initial state, in which the first call of this function block starts. During each execution, the active state of a SFC changes if a transition becomes true. Transitions are Boolean expressions which are evaluated during each execution. They are also used to choose between two alternative branches. Note that SFCs also support parallel branches, which causes multiple states to become active. An active state executes its actions while waiting for the next transition to become true. For example, while the state MovingLeft is active, the action MoveLeft is executed during each call. There are different modifiers used to change action behaviour. For instance, the pulse action StopMoving is only executed once. Each state may also use an entry action, executed when a state is entered and an exit action, which is executed while leaving the state. At the bottom of the SFC is an arrow pointing to the right at WaitStart. This is a jump back to the state WaitStart. Jumps are used to create loops or to skip states.

Function Block Diagram

Another graphical notation is the function block diagram. It consists of one or more networks. Each network uses an arbitrary number of input or state variables and connects them to function blocks. The outputs of the function block are processed by other blocks, or assigned to outputs. Each network therefore can be seen as chain of function blocks each modifying a given set of inputs and transforming them to outputs.

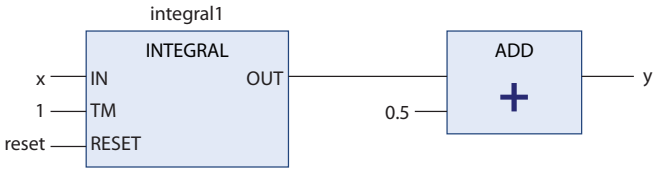


Figure 2.3: chain of blocks connected in a function block diagram

In Figure 2.3, a network implementing an integration is shown. It uses a standard function block `INTEGRAL` which integrates its given input `IN` during a given time `TM`. The value of the integrator can be reset using its `RESET` input. Inputs of a block are on the left side, while outputs are placed on the right side. The inputs are set by using variables `x` and `reset`, as well as a constant. The output of the integrator is used as input for an addition block, which adds 0.5 to the integrator result. The final value of the network is written into the variable `y`.

2.3 Example

One of our first case studies was the implementation of a portal robot example. It is based on a tutorial example which KEBA uses to introduce new employees to IEC programming. A robot with a gripper is moved inside a depot and can be controlled in all three dimensions. The movement is constrained by $xLimitLeft$ and $xLimitRight$ on the x-axis, by $yLimitLeft$ and $yLimitRight$ on the y-axis, and $zLimitBottom$ and $zLimitTop$ on the z-axis. The robot's task is to move elements from the origin to a reference position, where it is processed. The finished product must then be transported to its destination. Figure 2.4 shows an abstract view of the structure.

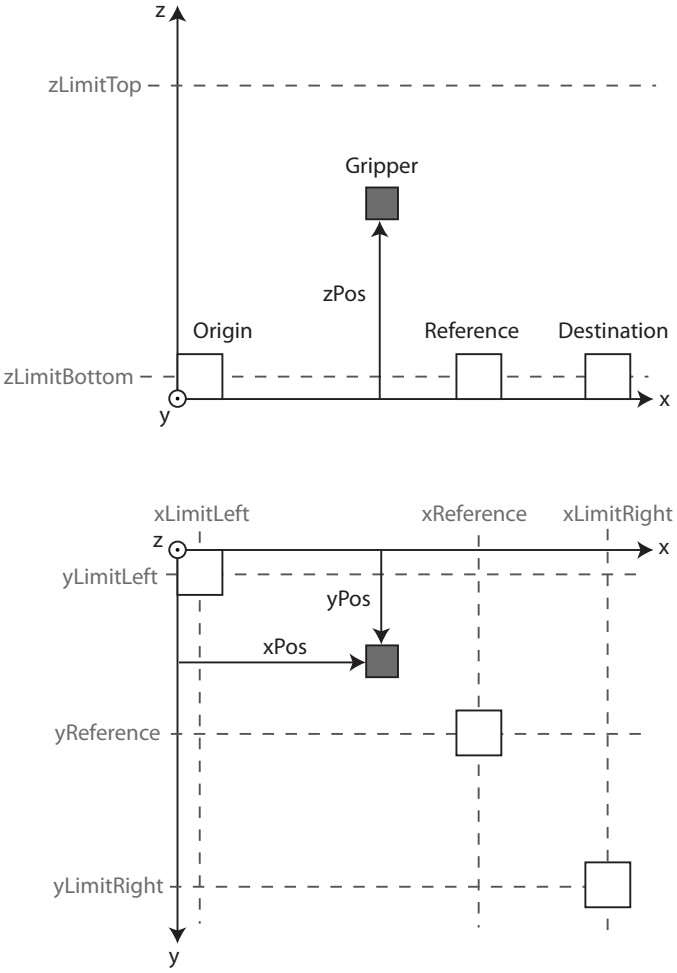


Figure 2.4: Schematic of the portal robot case study

The used hardware is quite simple. Every axis uses a sensor at each stop position and the reference position. These are mapped to global Boolean variables, which is TRUE if the gripper is at that position, and FALSE otherwise. The gripper itself is controlled by Boolean variables as well. Each axis has two Boolean inputs, one for each movement direction. Setting one of

these moves the gripper in that direction. If both inputs are FALSE, the gripper stops moving.

2.3.1 Implementation

The implementation of this example can be split up into two parts:

- A controller application, which controls the hardware using global shared variables. It repeatedly moves the gripper to the origin, picks up a new item, moves to the reference position and lets go, picks up the final product, and moves it to its destination.
- A simulator running in a separate task, which emulates the original hardware

Simulator

Each axis is simulated separately. A function block, named `AxisSimu` is used to simulate an axis position over time, based on the input variables `MotorDir1` and `MotorDir2` and its state. Each axis is considered to be of equal length, stretching from 0 to 500. Both the x and y axis additionally define a reference position, which is configured by setting the `RefPos` input. To simulate a hardware failure, each axis can also be blocked, which halts its movement. Depending on the current position of the gripper along the axis, output variables indicate its location. The Boolean variable `LimitS1` becomes true if the gripper passes $pos = 0$ (end position 1). `LimitS2` is true when the gripper passes $pos = 500$. `LimitRef` becomes true if the reference position is reached. The interface of the `AxisSimu` function block can be seen in Listing 2.3.

```
FUNCTION_BLOCK AxisSimu
VAR_INPUT
    MotorDir1 : BOOL;
    MotorDir2 : BOOL;
    RefPos : INT;
    Blocked : BOOL;
END_VAR
VAR_OUTPUT
    LimitS1 : BOOL;
    LimitS2 : BOOL;
    LimitRef : BOOL;
END_VAR
VAR
    Pos : INT := 250;
END_VAR
```

Listing 2.3: Interface of the AxisSimu Function Block

The machine simulator uses three of these function blocks, one for each axis (see Listing 2.4). During each program cycle, the axis objects are called, which updates their state and reports back. The input and outputs of these axis objects are then connected to global shared variables (see Listing 2.5). Since real machine inputs and outputs are mapped the same way, a separate running task which reads and writes to shared variables can be used to emulate the behavior of a real machine.

```
PROGRAM Simu
VAR
    xAxis : AxisSimu;
    yAxis : AxisSimu;
    zAxis : AxisSimu;
    blocked : BOOL;
END_VAR
```

Listing 2.4: Interface of the simulator program

```
xAxis(MotorDir1 := HWOutputs.XMotorLeft,
    MotorDir2 := HWOutputs.XMotorRight,
    RefPos := 350,
    Blocked := blocked,
    LimitS1 => HWInputs.XLimitLeft,
    LimitS2 => HWInputs.XLimitRight,
    LimitRef => HWInputs.XReference
);
```

Listing 2.5: Calling of function block and connecting it to global shared variables

Process controller

The controller of the portal robot example is implemented using a sequential function chart (SFC), seen in Figure 2.5. It is a chain of states, which trigger specific movements of the robot.

After the application is started, the robot moves up the z-axis until it reaches $ZLimitTop$. It then moves to the origin along the x- and y-axis. The origin is reached once the $XLimitLeft$ and $YLimitLeft$ sensors get activated. It then waits until something is placed at the origin position. This is indicated by the `StartPosOccupied` flag. The robot then moves down and grabs the new item once it reaches $ZLimitBottom$. Now loaded, it moves back up to $ZLimitTop$ and moves to the reference position along the x- and y-axis. At the reference position, indicated by the $XReference$ and $YReference$ sensors, the gripper moves down to $ZLimitBottom$ and releases the item. A worker then processes the delivered item and notifies the robot when it is done by setting the Ready flag. The gripper grabs the finished product and moves back up to $ZLimitTop$. The final product is then moved to its destination, which is located at $XLimitRight$ and $YLimitRight$. The robot goes down and releases the product at $ZLimitBottom$. It then repeats this working cycle. Figure 2.5 shows the implementation of this work cycle in the SFC graphical notation.

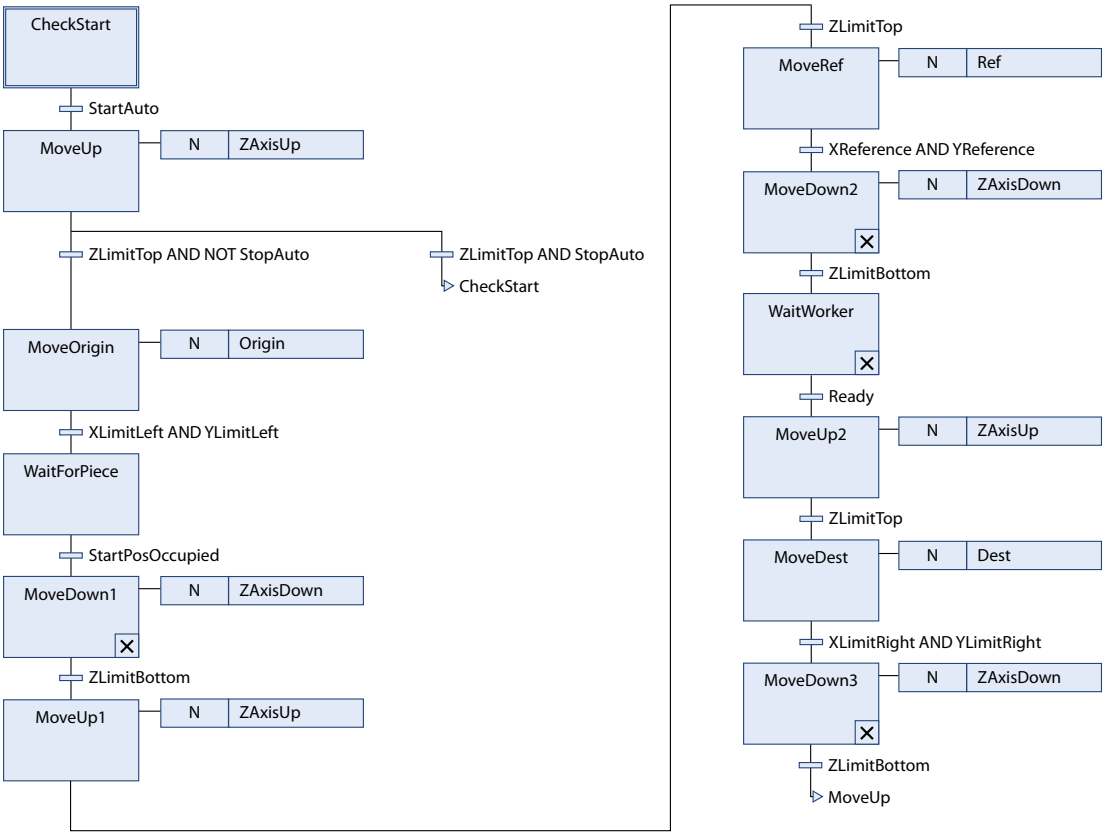


Figure 2.5: Function Block MoveAutomatic, which controls the robot

Axis controllers

The individual movements triggered in the actions of `MoveAutomatic` are realized by three `AxisMove` function blocks. Each controls one axis of the robot by setting its `MotorDir1` or `MotorDir2` inputs. The controller itself is controlled by setting the `StartDir1` or `StartDir2` input variables. Additionally, a timer is used to implement a timeout if a movement takes too long. Note that Listing 2.6 shows the code for moving in one direction. The variables `MotorDir1` and `MotorDir2` are outputs which are forwarded to `MoveAutomatic` and further connected to the global variables of each axis (e.g. `HWOutputs.XMotorLeft`).

```
IF (StartDir1 OR MotorDir1) AND NOT StartDir2 THEN
    MotorDir2 := FALSE;
    TimerDir2(IN := FALSE);
    IF TimerDir1.Q THEN
        State := Error;
        MotorDir1 := FALSE;
        TimerDir1(IN := FALSE);
    ELSIF LimitS1 THEN
        State := Limit1;
        MotorDir1 := FALSE;
        TimerDir1(IN := FALSE);
    ELSE
        State := Mid;
        MotorDir1 := TRUE;
        TimerDir1(IN := TRUE, PT := T#5S);
    END_IF
END_IF
```

Listing 2.6: Part of the controller which causes the robot to move left along an axis

Chapter 3

Trace Visualization Tool

The offline debugging and trace visualization tool utilizes the data obtained by Capture & Replay as well as analysis methods of fellow colleagues. It initially started out as a simple plug-in for the CoDeSys Automation Platform, allowing to replay traces and navigating in them. During this time it was nicknamed *Time-Travelling Debugger*, since it not only supported single step debugging, but also stepping backwards in time. However, simply navigating and replaying traces was only the beginning. New views emerged and eventually made it necessary to turn this plug-in into a standalone application. It gradually grew more complex and became a complete debugging environment, challenging usual single-step debuggers.

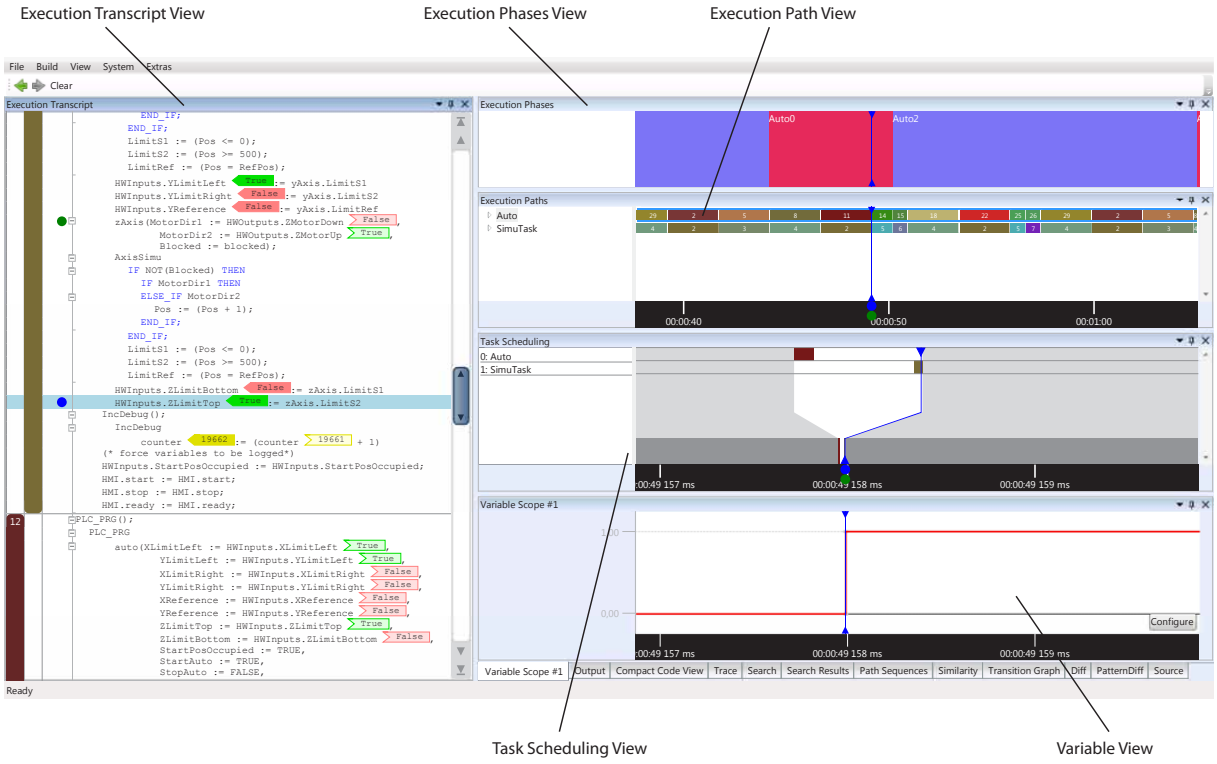


Figure 3.1: Screenshot of the trace visualization tool

This tool allows developers of PLC programs to explore and analyse trace recordings using a variety of different views. Data is presented at multiple levels of abstraction. Each view shows a different level of detail, allowing the user to dive into the program from a high-level view to a low-level view. The main views can be seen in Figure 3.1 and are [19]:

Execution phase view: Highlights phases of a program run, detected by a phase detection algorithm as described in [9].

Execution path view: A high-level view which shows the recorded execution paths over time.

Task scheduling view: A low-level view displaying active task regions over time and illustrating task interleaving.

Execution transcript view: Similar to the task scheduling view, however, with far more detail. It shows the source code of executed statements for the different tasks, also including task switches.

Variable view: Shows variable plots over time.

Additionally, there are many smaller views, which are used as aid or to highlight specific aspects of other views. For example, the diff view allows comparing the executed code of different program executions.

3.1 Overview

The following sections introduce what kind of data is processed and how the tool displays it. The different views and their core features are described.

3.1.1 Trace Data

The combined trace data produced by Capture & Replay is the main source of data for the tool. It is a binary file containing millions of trace entries, which have been recorded at the production facility and enriched with additional information by replaying the recording. A human readable portion of this raw data is shown in Listing 3.1. Each line represents a trace entry. While replaying an application, these entries are consumed from beginning to end. A trace always starts with a task switch. This sets the active task. All successive entries are then consumed by this task until another task becomes active through a task switch. Each task's execution cycle begins with a *BeginCycle* and ends with an *EndCycle* entry. Both carry a timestamp. In between these two, various *Position* statements can be found. These are used as position markers for each block of code. They are added in the second phase of Capture & Replay, allowing the reconstruction of executed source code. Besides these position marks, important variable values are stored in *Read* and *Write* entries.

```
TASK SWITCH 1
    BEGIN CYCLE at 16529553540728
TASK SWITCH 0
    BEGIN CYCLE at 16529553542178
    POSITION at 5508
    READ  at 5494 HWINPUTS.XLIMITLEFT (bool), ptr: 172763997 = False
    READ  at 5495 HWINPUTS.YLIMITLEFT (bool), ptr: 172764263 = False
    READ  at 5496 HWINPUTS.XLIMITRIGHT (bool), ptr: 172763998 = False
    READ  at 5497 HWINPUTS.YLIMITRIGHT (bool), ptr: 172764342 = False
    ...
    POSITION at 489997
    READ  at 48872 TIME() (uint32), ptr: 0 = 16529238
    POSITION at 48978
    WRITE  at 5502 HWOUTPUTS.XMOTORRIGHT (bool), ptr: 172764727 = False
    WRITE  at 5503 HWOUTPUTS.XMOTORLEFT (bool), ptr: 172764728 = False
    WRITE  at 5504 HWOUTPUTS.YMOTORRIGHT (bool), ptr: 172764729 = False
    WRITE  at 5505 HWOUTPUTS.YMOTORLEFT (bool), ptr: 172764730 = False
    WRITE  at 5506 HWOUTPUTS.ZMOTORUP (bool), ptr: 172764731 = False
    WRITE  at 5507 HWOUTPUTS.ZMOTORDOWN (bool), ptr: 172764732 = False
    POSITION at 20768
TASK SWITCH 1
    POSITION at 55819
    READ  at 55814 HWOUTPUTS.XMOTORLEFT (bool), ptr: 172764728 = False
    READ  at 55815 HWOUTPUTS.XMOTORRIGHT (bool), ptr: 172764727 = False
    ...
```

Listing 3.1: Example of a combined trace buffer (human readable text version)

Execution Paths

A further source of information is generated by looking at each task independently and without considering task switches, i.e., trace entries from other tasks are ignored. A trace recording can then be seen as a list of this task's execution cycles. The contents of each cycle are the trace operations between *BeginCycle* and *EndCycle*. These operations create a sequence of position marks. Each of these sequences is like a trail which shows which path was taken through the code of the task program. Figure 3.2 illustrates how a sequence of trace operations can lead to different paths. It shows an instrumented task program containing two IF statements. Both have a THEN and ELSE branch. Depending on which *Position* statements are executed, a different control flow can be reconstructed. Because of the two possible choices at the IFs, there are four paths through this piece of code. Every path is represented by a unique sequence of trace operations. Each of these sequences is given a path id (see Figure 3.3). In a program run, each task follows one path during each cycle. Therefore a list can be generated, which contains the path id for each cycle. Such a list may look like Listing 3.2.

```
1111111111111112348888888888888888888888888897777777777777777777234555555...
```

Listing 3.2: A typical list of execution paths

In this list certain paths (1, 5, 7, 8) are executed for more than one cycle. These paths can be seen as longer lasting activities of the task. Other paths (2, 3, 4) however are executed only

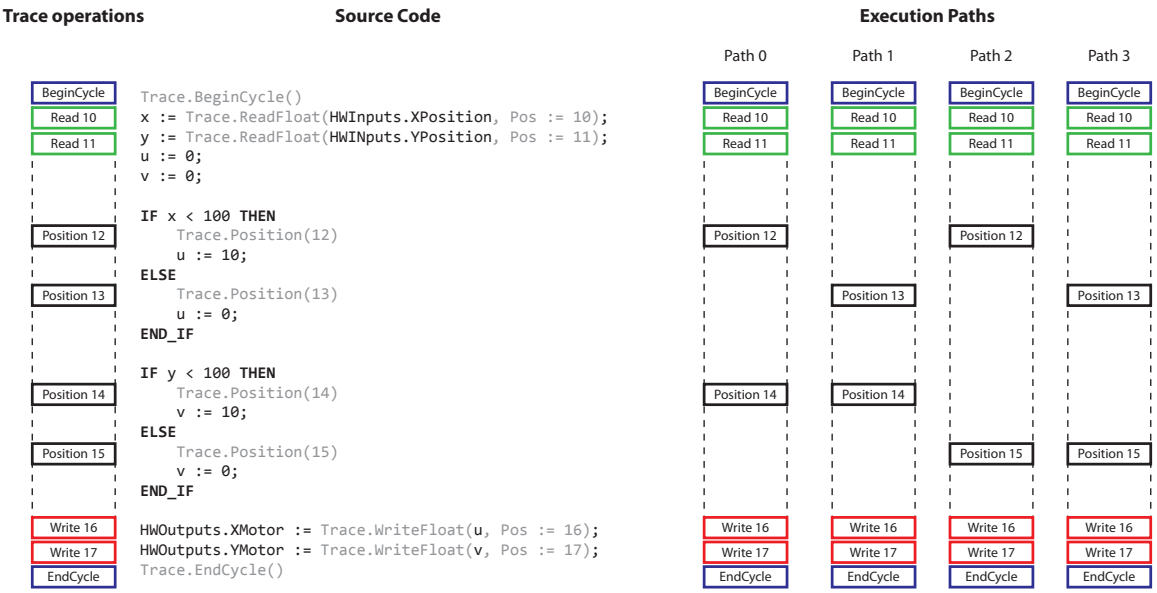


Figure 3.2: Execution paths are unique sequences of trace operations

once consecutively. These paths are considered to be transitions between program states. This progression of activities and transitions can be interpreted as the reactive behavior of a task. The execution path lists of all tasks can therefore be seen as a recording of the reactive behavior of the entire application.

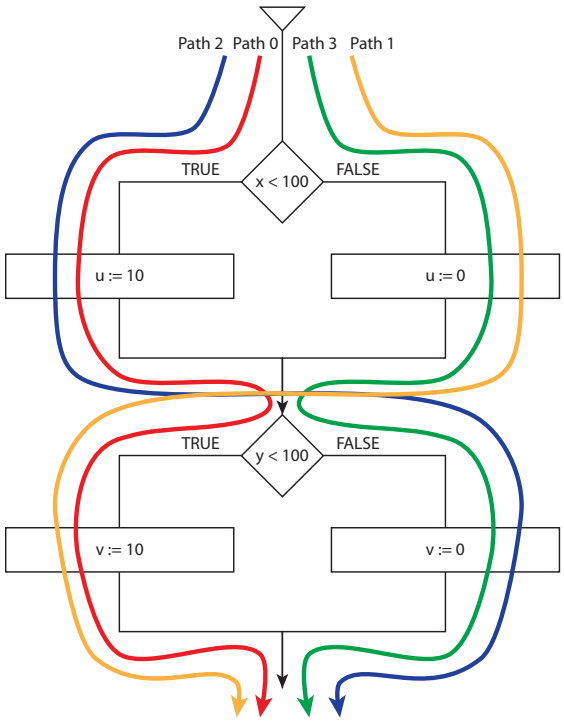


Figure 3.3: Execution paths

3.1.2 Task Scheduling View

The task scheduling view uses combined trace data to display the task interleaving in a Gantt chart. It is shown in Figure 3.4. Time progresses along the horizontal axis while the individual rows show different tasks. This view displays when the execution of each task cycle begins and ends. These cycles are defined by the timestamps stored in the *BeginCycle* and *EndCycle* entries and the *TaskSwitch* entries. Each task is colored in a unique way.

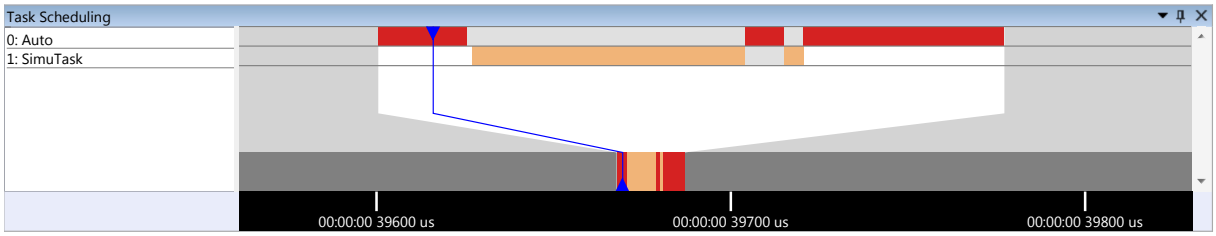


Figure 3.4: Screenshot of the task scheduling view

A cycle may be interrupted by other tasks. Such interruptions are visualized by drawing a gray region between active parts of a cycle. In Figure 3.4 the Auto task is interrupted by the SimuTask two times. Finding problematic interleavings of tasks is the main purpose of this view. Because of the dynamic scheduler used by the CoDeSys runtime environment, data races could be introduced when using global variables for inter-task communication. Finding these problematic areas helps narrowing down the problem.

Note that since execution times of task programs are very short (a few nanoseconds) compared to cycle times (milliseconds), showing these cycles using a linear time scale is not practical. All interesting parts would be too small compared to the idle time. Therefore, this view expands the active portions of a program run and compresses the idle time. To illustrate this expansion of time, a regular linear timeline is shown underneath the magnified area.

This view displays data at a low level, only focusing on a few dozen cycles at once. Displaying more than a few cycles by increasing the visible time range is limited, since at some point, individual cycles can no longer be identified.

3.1.3 Execution Path View

The execution path view visualizes the reactive behavior of an application as a Gantt chart (see Figure 3.5). Tasks are drawn along the vertical axis, while time progresses along the horizontal axis. Unlike the task scheduling view, this view can display the entire trace recording at once. The time range shown in Figure 3.5 is about 2.5 minutes long. Displaying a task with a cycle time of 1 ms therefore shows 150,000 task cycles of that task at once.

This is done by assigning a unique color to each execution path of a task and drawing each cycle as a rectangle along the time axis. As seen in Figure 3.5 this reveals common patterns caused by regular application operation. The colors are chosen in a way so that similar execution paths have similar colors.

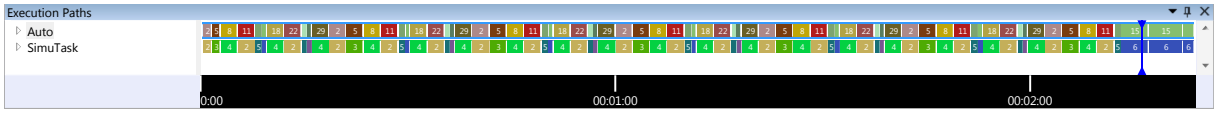


Figure 3.5: Screenshot of the execution path view

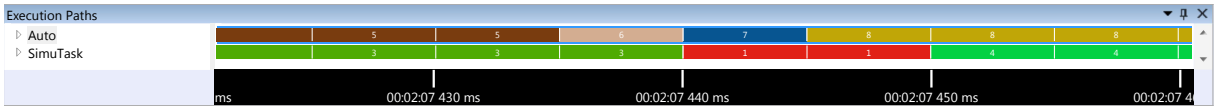


Figure 3.6: Zooming into the execution path view reveals individual cycles

Depending on the zoom level, long sequences of equal paths are grouped to one element. The execution path id is drawn in the center of this group. Zooming into this view reveals individual cycles (see Figure 3.6), each containing its own execution path id.

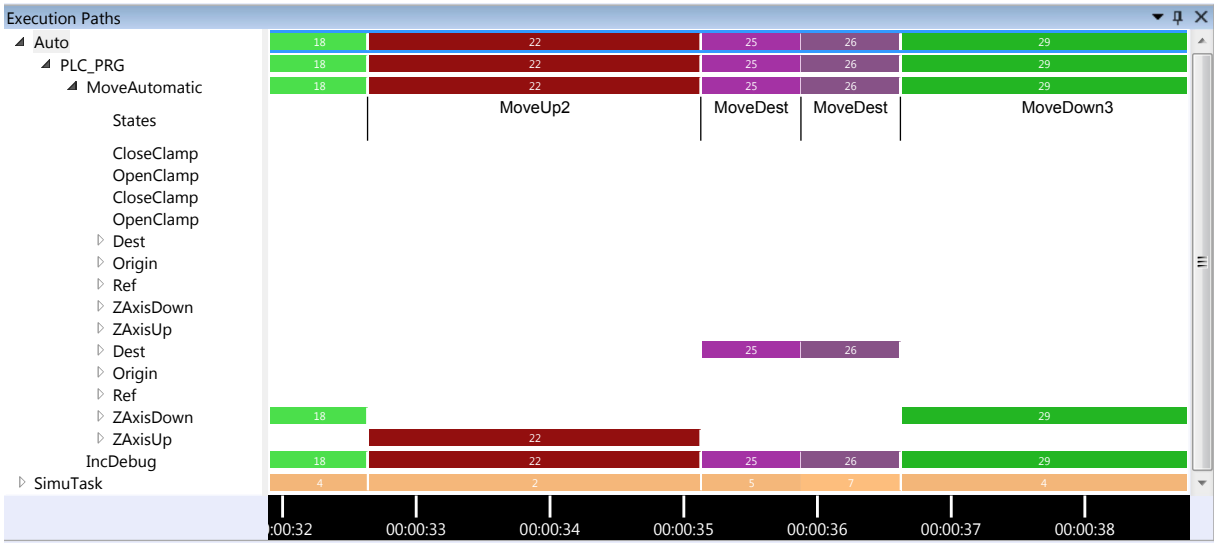


Figure 3.7: Screenshot of the expanded execution path view

Each task can also be expanded (see Figure 3.7) which shows its static call tree. Each element of this tree only displays execution paths which contain the corresponding call. This can be used to identify positions where a specific function block is executed. Calls of POUs which are implemented in SFCs also show an additional row which contains the progression of active states of the SFC graph. Every execution path is selectable. Right-clicking on any execution path opens a context menu. This menu is used to display specific execution paths in the diff view, which is introduced in Section 3.1.6.

3.1.4 Execution Phase View

Another view, which is usually placed above the execution path view, is the execution phase view. It provides a high-level view of the program run by showing execution phases of a task which have been detected by a phase detection algorithm [9]. An execution phase is given by a

unique sequence of execution paths. Less common phases are unusual execution behavior and may be interesting starting points for debugging.

Figure 3.8: Screenshot of the execution phase view

The current version of this view assigns an automatic name, built by the task name and an increasing count (e.g. Auto0), and a unique color to each phase. Both name and color of each phase can be changed by the user.

3.1.5 Execution Transcript View

This view displays the executed statements of the entire program run at the source code level. Figure 3.9 shows a screenshot of the view.

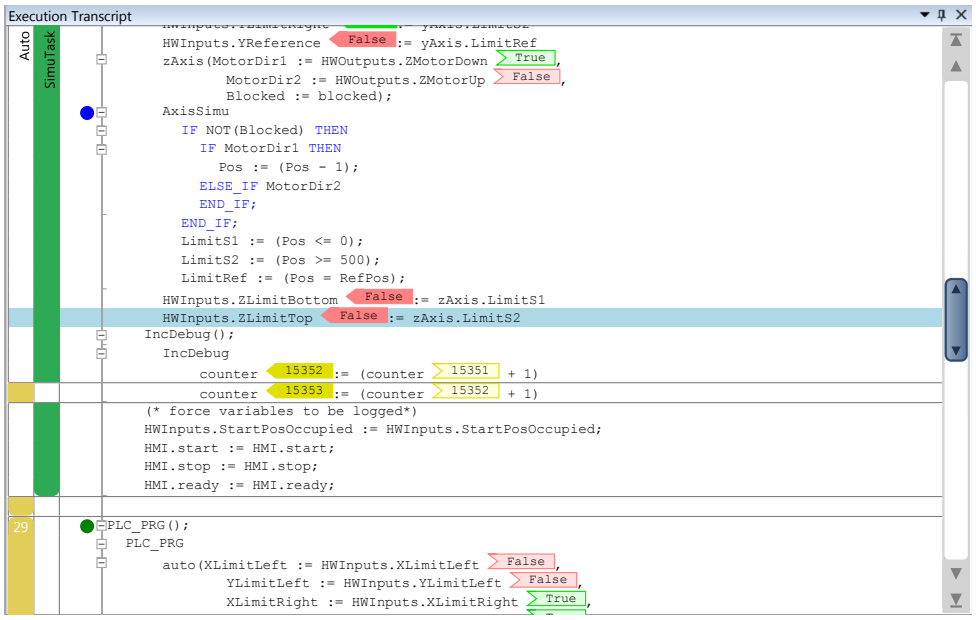


Figure 3.9: Screenshot of execution transcript view

As statements progress along the vertical axis, task switches might occur. To illustrate the task affiliation of each statement, vertical bars are drawn next to the statements. Each column represents a specific task. Each sequence of statements affiliated with one task is delimited by a horizontal gray line. Note that the colors used to fill these columns are the same as in the task scheduling view. In some use cases, developers might only want to focus on smaller groups of tasks. Each task can be hidden by double clicking the task column. This hides all statements of that task. Double clicking again makes them reappear. Every statement is further enhanced by displaying variable values of read or written variables.

This view is designed to replace a single step debugger. It is no longer required to run to a specific location to reach a specific state. A developer can jump to any location and look at the

variable values. By looking at the progression of values, the execution of the program can be examined.

Each program run contains millions of lines of code. To simplify navigation, this view highlights a single line with a light blue background. This is the current line of the view, which represents its current position in the trace. Navigation moves this highlight to other lines. The highlighted line itself however is always near the vertical center of the view.

Diff Mode

The execution transcript view has a special diff mode, seen in Figure 3.10. In this mode, the view presents a single task only. For each consecutive sequence of identical execution paths, only one cycle is displayed. This reduces the amount of data displayed by the view, since many execution paths might be executed a thousand times, and highlights the reactive behavior of the program.

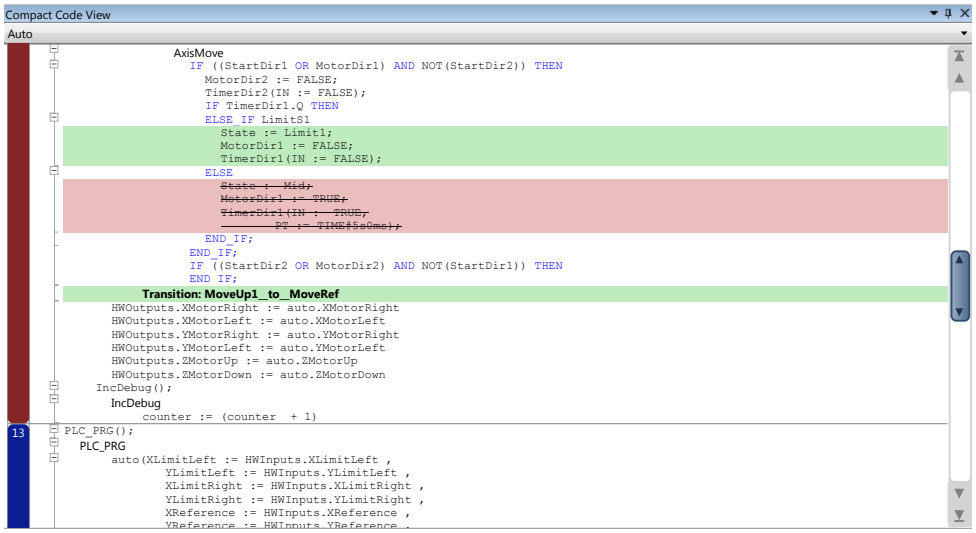


Figure 3.10: Diff mode of the execution transcript view

In essence, this view shows how the execution of a task changes over time. To illustrate these changes, the output is enhanced by diff information. Every cycle highlights statements which have been removed and added compared to the previous execution cycle. Statements which have been executed in the previous cycle but not in the current cycle have a red background and are striked through. Statements which are executed in the current cycle but not in the previous cycle are highlighted with a green background.

3.1.6 Diff View

The source code of individual execution paths can be compared using a diff view. Figure 3.11 shows the comparison of two very similar execution paths. The difference between the left and right execution path is highlighted with two colors. A green background highlights new code compared to the other execution path. This new code also introduces a gap in the other path

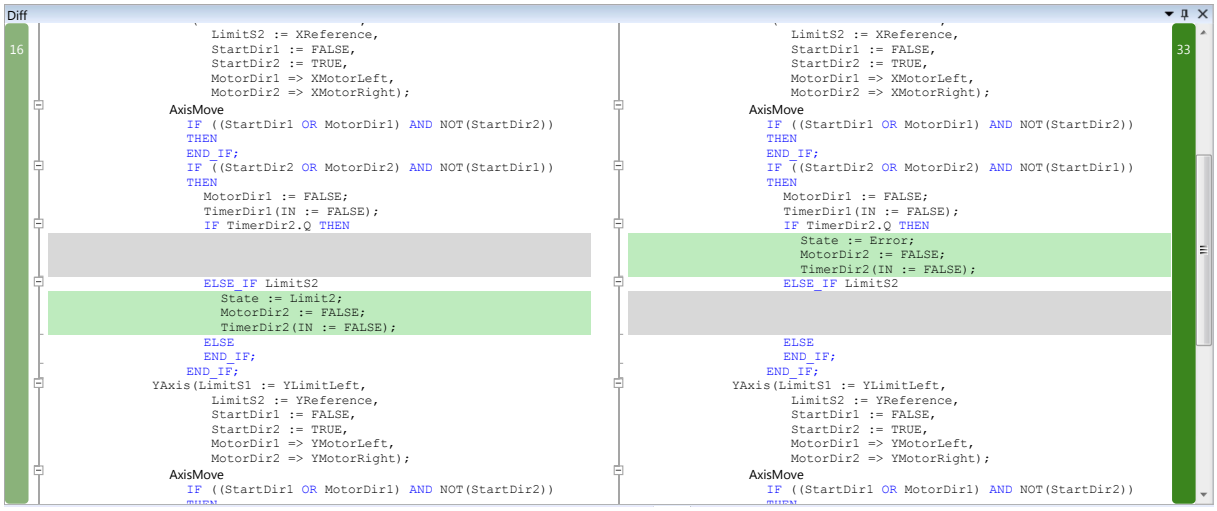


Figure 3.11: Difference between text-diff and diff view

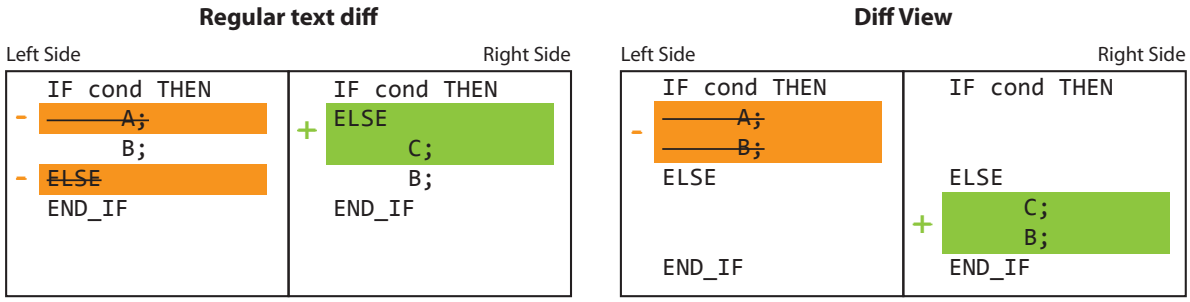


Figure 3.12: Difference between text-diff and diff view

that is highlighted in gray. The difference seen in Figure 3.11 shows that the left execution path executes an ELSIF branch, while the right path executes the THEN branch.

Unlike regular diff tools, which compare individual text lines, this diff view respects the hierarchy of the source code. The difference is illustrated in Figure 3.12.

While a text diff deletes individual lines which differ, it also keeps similar parts if they are in the correct order. However, this diff view uses the block hierarchy of the source code to decide if parts are removed or added. It adds gaps to compensate the removal or addition of blocks of code.

3.1.7 Variable View

The variable view is used to display one or more variables along a timeline. Each variable can be given a unique color, which distinguishes it from others. All variable plots are step functions, even those of REAL variables. This is because the value of a variable is always updated at discrete times. However, these steps are only visible at the highest zoom level. This view is optimized for performance, reducing the level of detail whenever possible. Figure 3.13 shows a variable view containing a plot of a Boolean variable.

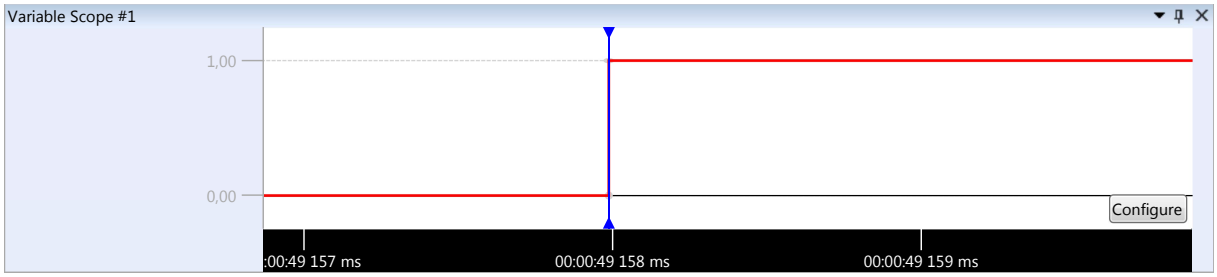


Figure 3.13: Variable view showing a `BOOL` variable

3.1.8 Navigation and Positioning

Navigation is done in a uniform way across all views. Visible time ranges can be moved by left-clicking and dragging the mouse. The mouse wheel is used to zoom into a specific location. Dragging with the left mouse button and holding the shift key highlights a time range into which the view zooms once the mouse button is released.

Many of the above views can not be used efficiently on their own and thus can be coupled through various mechanisms. One way of connecting views is by synchronizing the visible time range. For example, a variable view can be connected to the execution path view, which allows seeing how the reactive behavior influences the variable over time. Instead of synchronization, the visible time range of views may also be displayed as overlay on other views. In Figure 3.14 the visible time region of the task scheduling view is displayed as a blue rectangle in the execution path view.

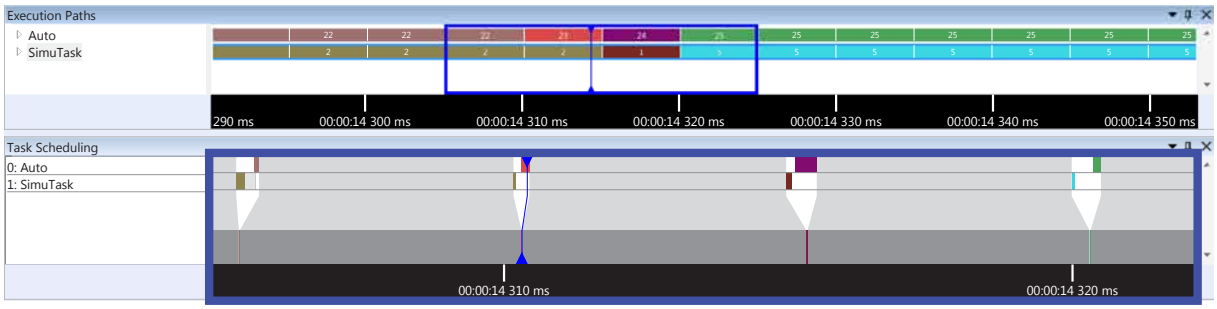


Figure 3.14: Visible time regions of other views are shown as colored rectangles

Another mechanism used to connect views are cursors. Each view with a timeline has an active cursor. It is set by double clicking on any position along the timeline. Cursors mark the current position of views which focus on a single location of the recording. E.g. the execution transcript is positioned at a specific line, highlighted by a light blue background. Scrolling inside the execution transcript view moves the current line, which is drawn as a blue cursor in other views. Figure 3.15 shows the blue cursor which marks the current position of the execution transcript. Setting this blue cursor as the active cursor allows high-level views to position it by double clicking on them.

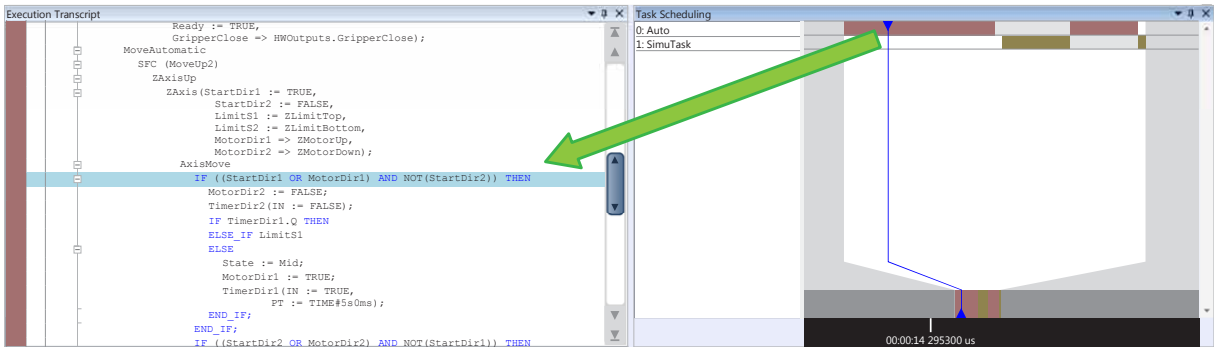


Figure 3.15: The blue cursor shows the current location of the execution transcript

Positioning of cursors through high-level views lets other views jump to different locations. Because developers may want to return to previous locations, two possibilities are offered: while jumping, a jump history is recorded. Forward and backward buttons, as well as special mouse buttons, allow jumping to the position before a jump or returning to the last jump position. A more permanent way of remembering certain locations is offered by setting bookmarks inside the execution transcript view. This is done by clicking on the left side of a statement. A colored circle appears (see Figure 3.16), which can also be seen on the timeline of other views. Clicking on such a circle jumps to that location.

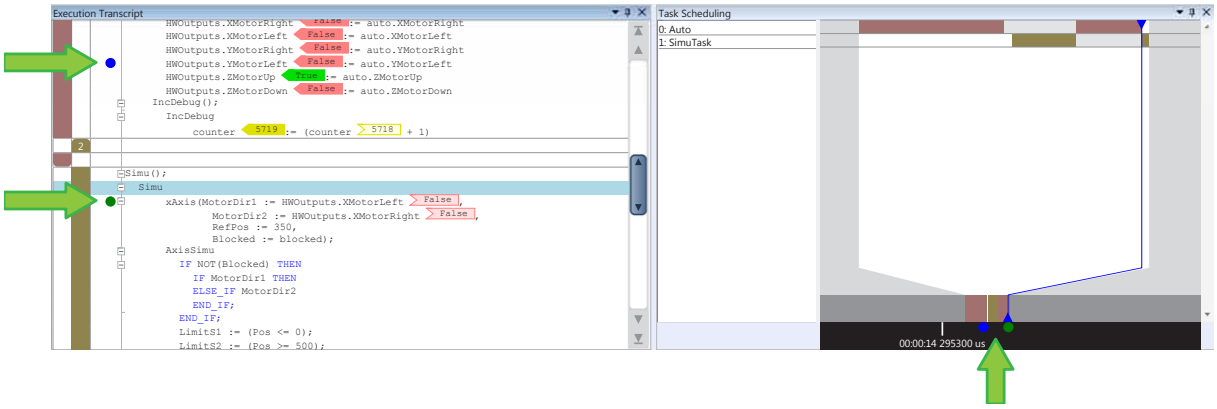


Figure 3.16: Line bookmarks mark specific locations and are displayed in various views

3.2 Example

The views introduced above are now used to show how a bug in the portal robot example can be found. The error report of the customer states that the robot stops moving. Looking at the execution phases of the Auto task, seen in Figure 3.17, shows a repeating pattern of the phases Auto0 and Auto2 (light green, dark green). However, at the end of the trace a different phase is found, named Auto3 and colored in red.

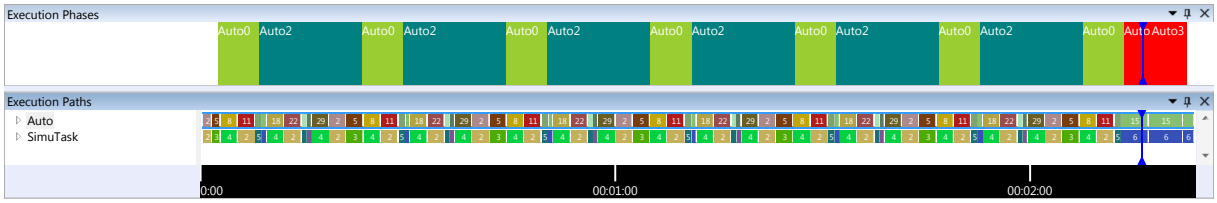


Figure 3.17: Execution phases of the examined trace

Positioning the mouse above the red Auto3 region and using the mouse wheel zooms in and out relative to that position. Zooming into these regions also zooms the connected execution path view to the same location. This reveals that the Auto task is in execution path 15, which eventually changes to execution path 33 (see Figure 3.18).

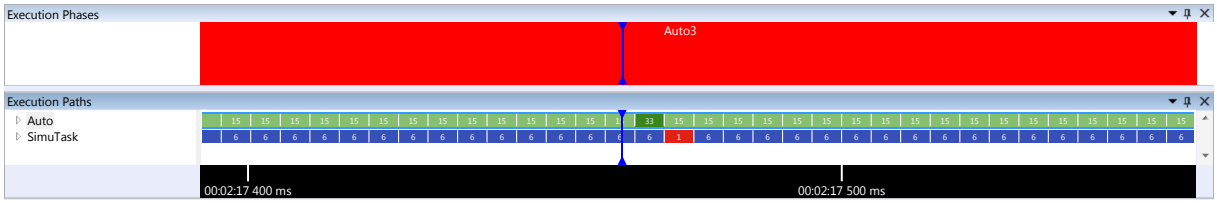


Figure 3.18: Execution paths of the unusual execution phase

In contrast to this incorrect behaviour, one could ask the question what should normally happen after an Auto0 phase. Zooming into the beginning of a Auto2 phase shows that the regular execution also contains a long sequence of path 15, however at the end it is followed by path 16 (see Figure 3.19). This means that the reason for the errornous behaviour must be the difference between the path sequence 15-16 and 15-33.

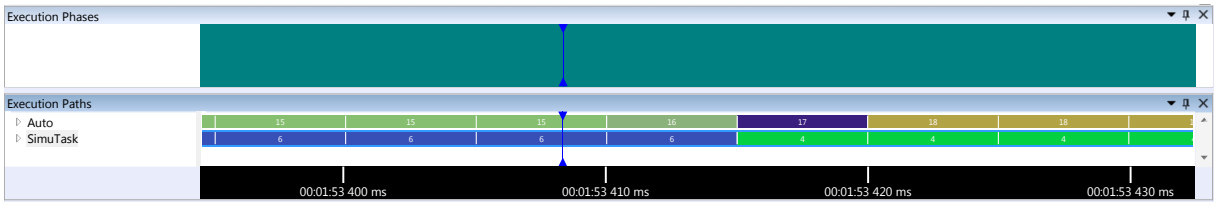


Figure 3.19: Execution paths of a regular execution phase

To examine these differences, these paths can be compared using the diff view. Paths are selected by right clicking on them in the execution path view and using the context menu item “Set Left to Path x” or “Set Right to Path x”. Another way is dragging execution paths from

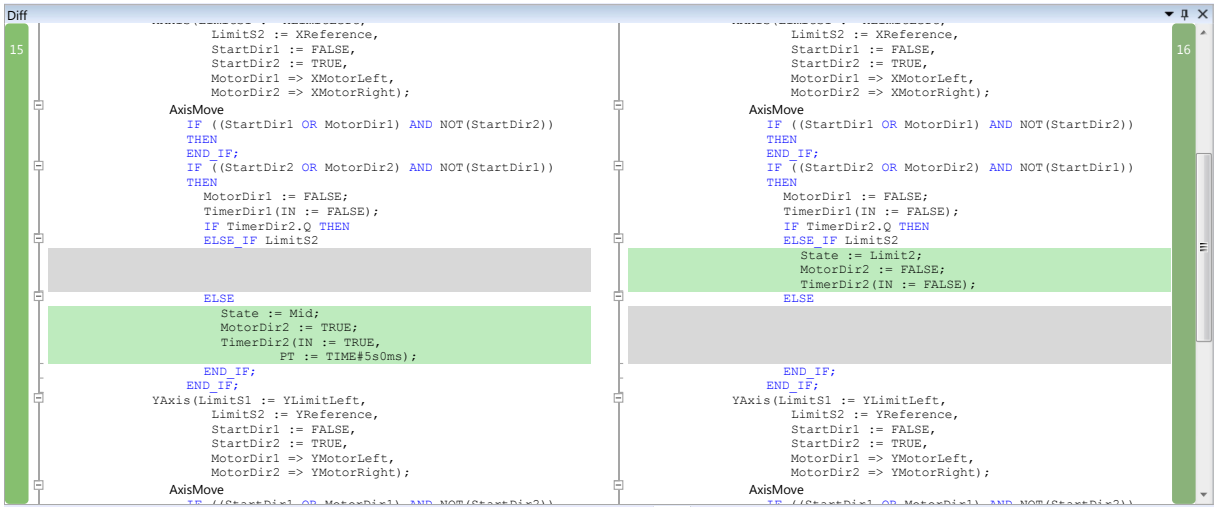


Figure 3.20: Difference between path 15 and 16

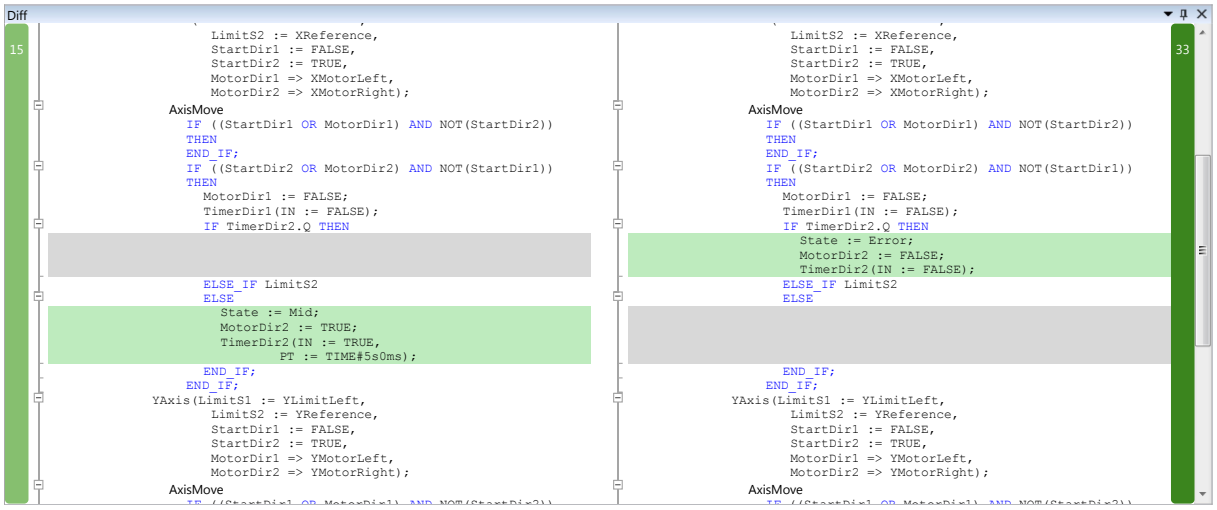


Figure 3.21: Difference between path 15 and 33

the view into the diff view. Figure 3.20 shows the difference between path 15 and 16. The application programmer can see that the code difference is inside a call of the function block `AxisMove`.

This POU consists of multiple cascading IF statements. Path 15 visits an ELSE branch of such a IF statement, while Path 16 enters an ELSIF branch. It is reached because the variable `LimitS2` is TRUE. `LimitS2` is an input variable assigned to the global variable `XReference`. The difference between path 15 and path 33 can be seen in Figure 3.21. Path 33 enters the THEN branch of the IF statement because of `TimerDir2.Q`. It is a boolean value which becomes true if a timeout occurs.

This means the robot stops because of a timeout. The next question is how and why does this timeout happen. During regular execution, path 15 is succeeded by path 16. This means `XReference` is TRUE before the timeout is fired. To see what happens with this variable, a variable scope containing the `XReference` variable is added and connected to the execution path

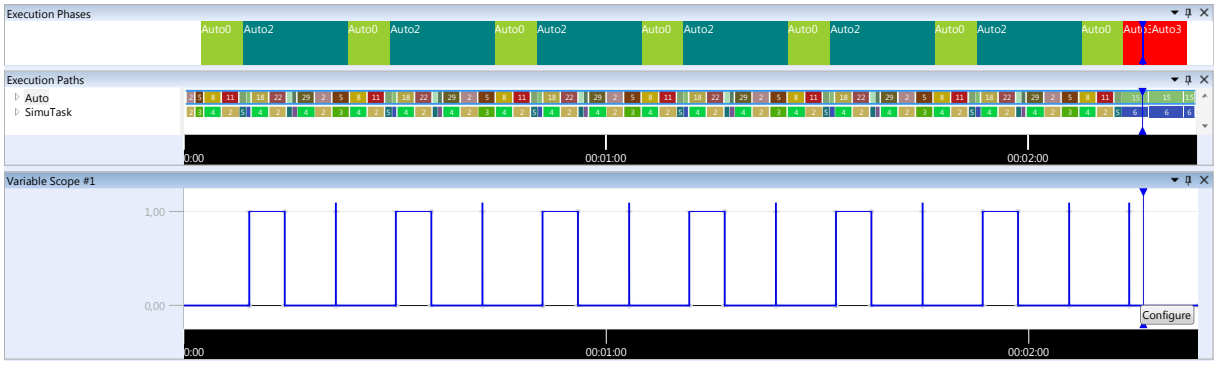


Figure 3.22: Variable Scope displaying the time behaviour of the `XReference` variable

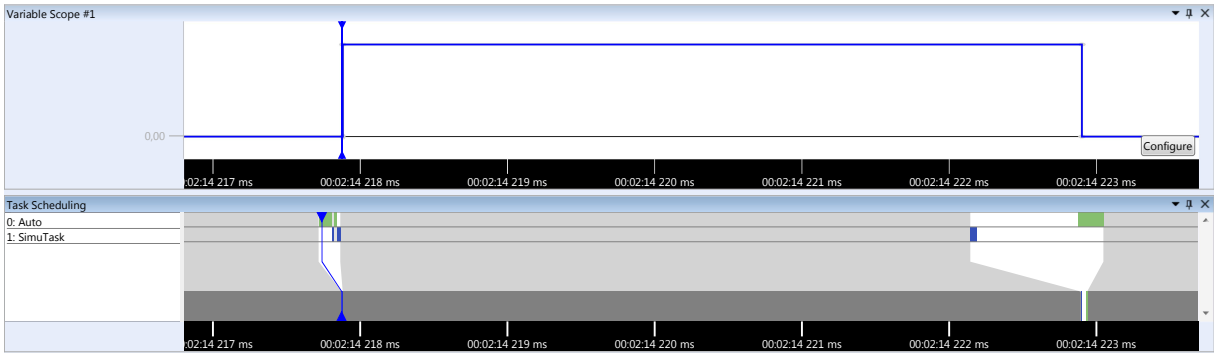


Figure 3.23: Last TRUE spike of the `XReference` variable

view (see Figure 3.22).

This view shows how the value of the variable changes over time, relative to execution paths and phases. During regular execution, the `XReference` variable seems to be following a regular pattern. It stays TRUE for a long time, becomes FALSE and is briefly set to TRUE before repeating the pattern. However at the end of the trace, in the erroneous Auto3 phase, the short TRUE spike is repeated. Something must have gone wrong here, because normally a longer TRUE phase should follow. Zooming into the last TRUE spike and looking at the task scheduling view reveals that it only lasts for one execution cycle. See Figure 3.23.

A developer will now want to find out where the variable is set to TRUE and who is responsible for resetting it to FALSE. To do this, the execution transcript view can be used to look at the executed source code. It can be positioned by double clicking on the task scheduling view. To simplify positioning, the variable view allows clicking on the variable curve as well. The cursor of the execution transcript is then positioned to the nearest write of that value. This has been done in Figure 3.24. The blue highlighted line in the execution transcript view is exactly positioned at the assignment statement, which sets `XReference` TRUE. To remember this location, a bookmark can be placed by clicking on the left side of the source code. A colored circle appears next to the line and along the timelines of all other views. Clicking on the circle moves the execution transcript back to this position.

Double clicking on the right edge of the TRUE spike moves the current line to another write

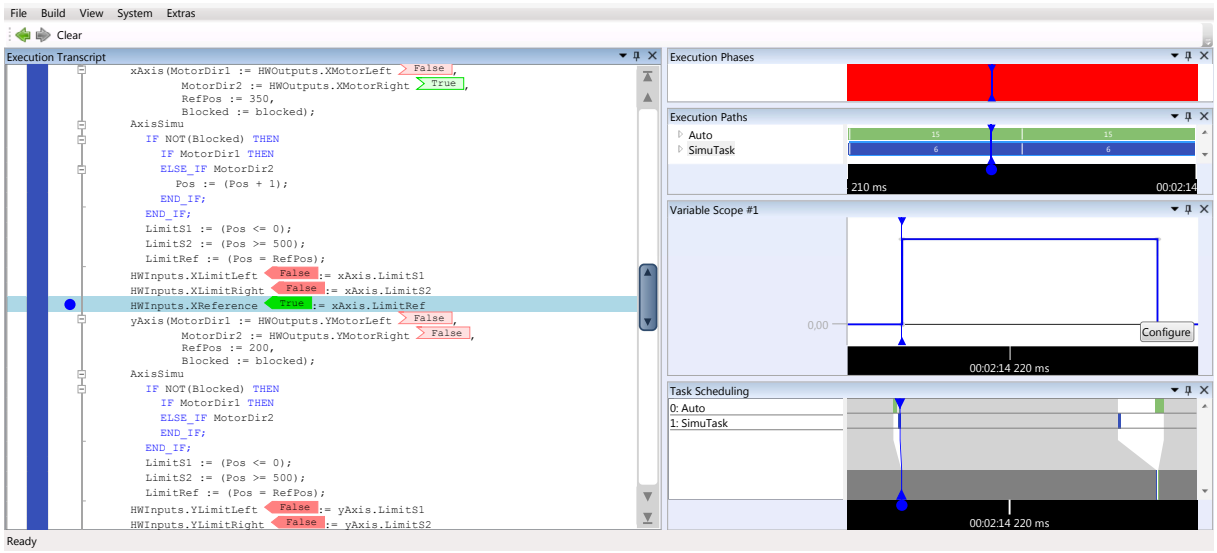


Figure 3.24: Line writing `XReference := TRUE`

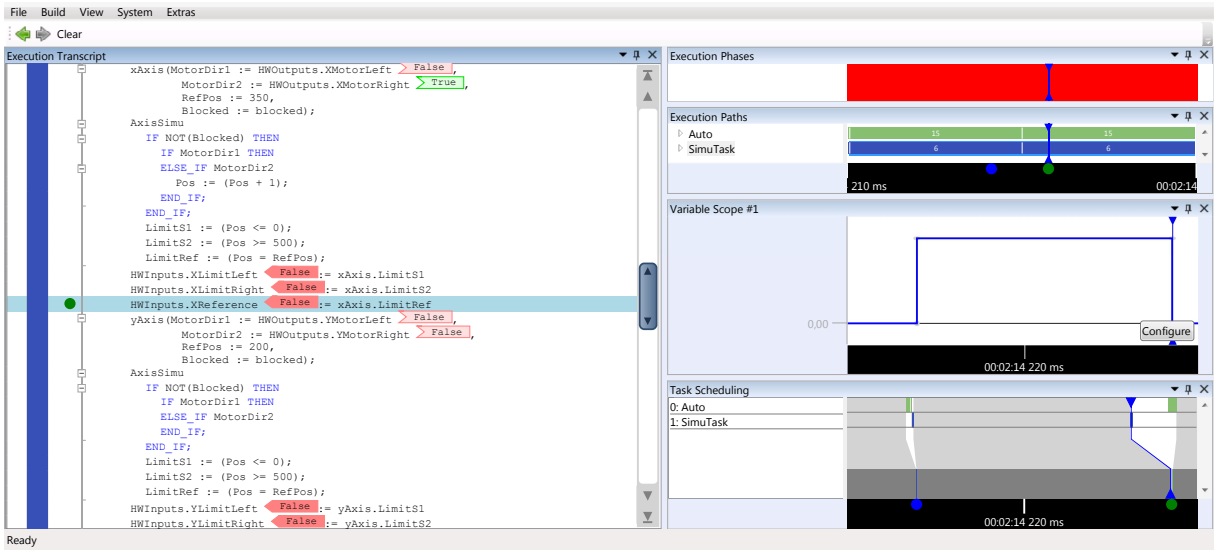


Figure 3.25: Line writing `XReference := FALSE`

operation. This one is responsible for setting the variable FALSE and can be seen in Figure 3.25. By adding a bookmark here as well, it is now very easy to switch between these two code locations by clicking the blue or green circle in the timelines. Reviewing this code in conjunction with task scheduling information reveals the problem. The simulator task is responsible for writing the `XReference` variable. The controller task only reads it. The simulator sets `XReference` TRUE if the robot reaches the reference position along the x-Axis. It is simply assumed that other tasks read this value before the simulator task is executed again. This works most of the time, as long the execution order of tasks remains always the same. However, for some reason at some point tasks are scheduled differently, which leads to a different execution order. The simulation task is executed two times in a sequence. In the second execution the simulator sets `XReference` to FALSE. The controller task never reads the TRUE value because it was scheduled after the

second execution of the simulator task. Hence a data race between simulator and controller task was responsible for the defect.

3.3 Tool Architecture

The following chapters focus on different aspects of the implementation. Before diving into the details, the overall architecture of this tool is discussed. Three main building blocks can be identified: Data, Core and View. Figure 3.26 illustrates these building blocks. They are explained in the following sections.

3.3.1 Data

Before any data can be visualized graphically, it must first be generated, loaded, processed, transformed and analysed. These responsibilities have been put into the `TraceTools` namespace and form a class library. The main purpose of the classes can be grouped into four topics:

Data Structures: Over the course of this project, many different data structures have been created to hold different data. Some of them are saved by serialization, others are generated on demand.

Caching: All data that is persisted on the hard drive is eventually loaded into memory. However, because of the large file sizes, not all of the data can be loaded at once. Intelligent caching mechanisms and strategies are needed to reduce the memory footprint.

Data Transformation: Once raw data has been loaded, usually some transformations are necessary in order to prepare it for display. For example, out of a raw binary trace the executed source code is recreated.

3.3.2 Core

The core of the application is a set of classes which define its state and behavior. It is made up of several subsystems which have different responsibilities and provide a framework for building data driven views.

Project management

The project management subsystem plays a major role in the application life cycle. Its main task is loading and switching between different trace projects. A trace project is a logical grouping of files associated to the same trace recording. In the current implementation all of these files are put into the same folder. The folder name defines the project name. The tool can only work on one trace project at a time.

Many of the files are generated using external tools, which process the trace files or other generated files. Since various views can not be displayed without first generating this data,

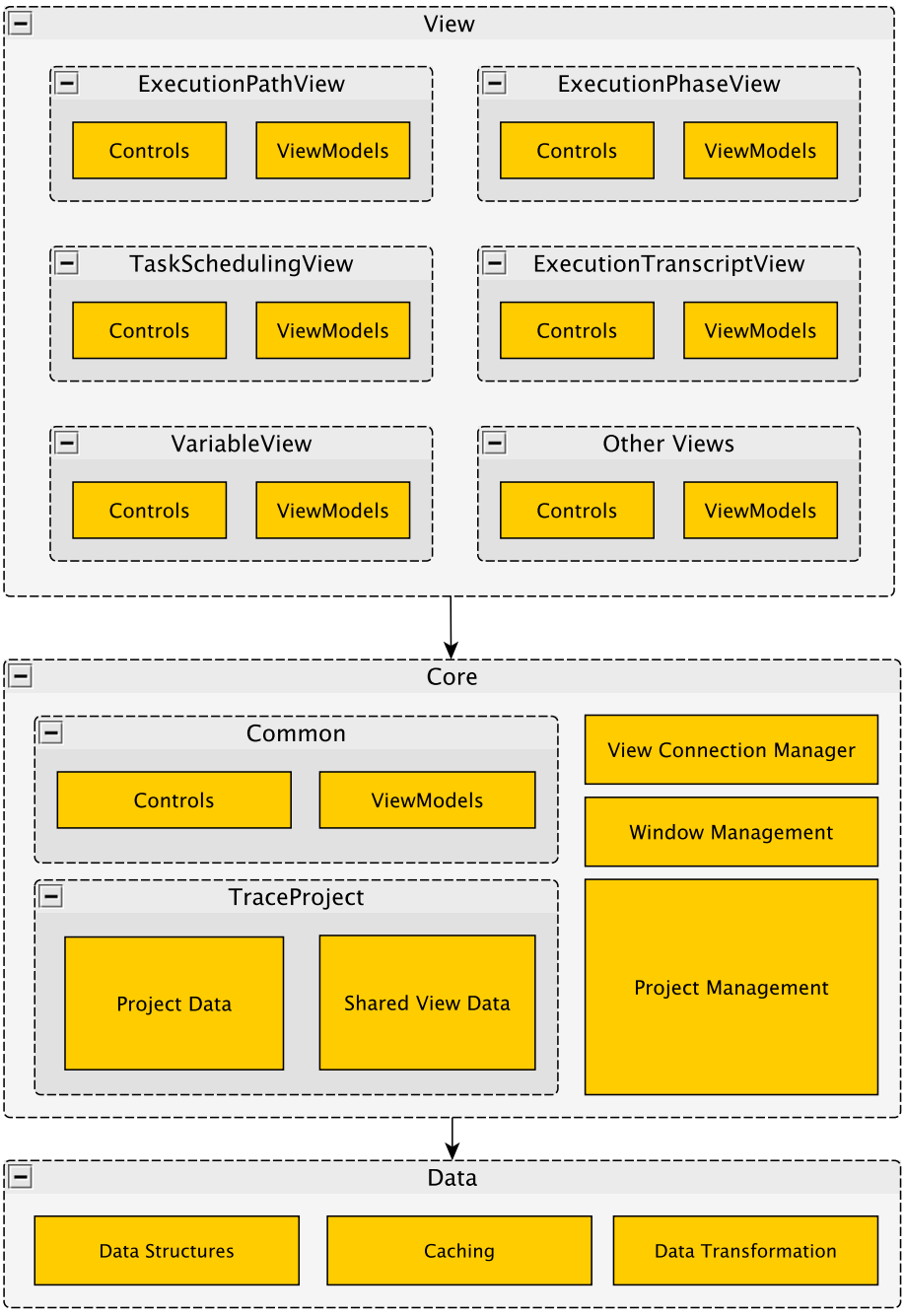


Figure 3.26: Architecture of the trace visualization tool

calling these external utilities has been integrated into the tool. The project management allows building or rebuilding a project, which means running the required utility programs. It also supports “clean all” which removes any generated files.

All views of the application use the active trace project to get their data. The DataSource of a project allows accessing data stored in project files by using the API exposed through the data layer. Additionally, a project manages information which is not view specific. For example, all views use the same colors to show the same elements. Therefore, a project provides a global color table which is used by all views.

Window & View Connection management

This project started out with a single view of the trace data. However as the number of views increased the introduction of a window management system became necessary. In order to focus on the actual task of building views, the open source docking library AvalonDock [1] was used as base of the implementation. By using dockable windows, views can be arranged and grouped freely.

View Framework

Although many different views have been implemented for this project, there are some parts which they all have in common. For example, mouse navigation is done in a unique, consistent way throughout all views. To simplify the process of creating new views and not reinventing the wheel all the time, much of the common functionality has been put into reusable components. These components can be roughly grouped into Controls and ViewModels. The latter term is introduced in Chapter 5 while discussing the Model-View-ViewModel pattern.

3.3.3 View

Views are built on top of core classes. They have access to the trace project and to the trace data sources. As illustrated in Figure 3.26, each view can be split up into controls and ViewModels. This depends on the way it is implemented.

Outlook

In the following chapters the details of the tool implementation are presented.

- Chapter 4 describes important aspects of the Data subsystem.
- Chapter 5 is a short introduction to various aspects of programming views in WPF.
- Chapter 6 then discusses the implementation of the most important views.

Chapter 4

Data Virtualization

Displaying large amounts of data is a challenging task. Although computers now have an increasing amount of memory, sometimes up to 8 GB and more, designing software which is capable of traversing millions of records and displaying them is still a great engineering effort.

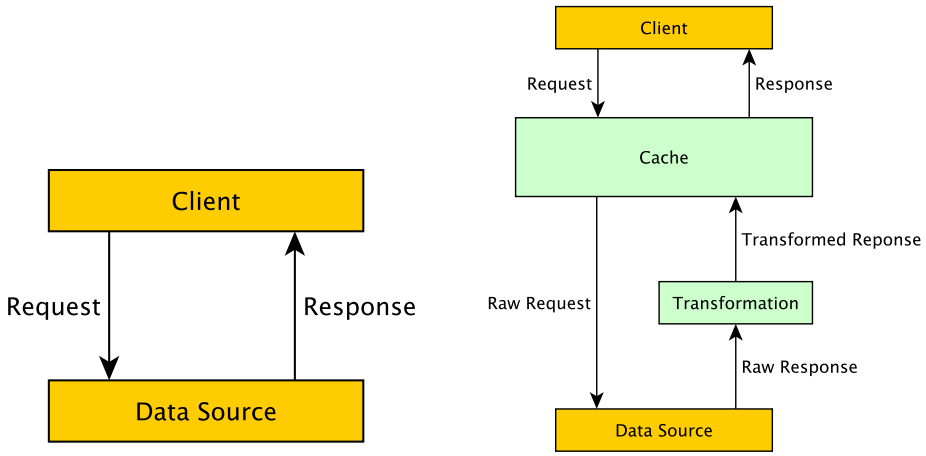
The reason why trace files are getting so big comes from the level of detail they expose. Each PLC application is split up into tasks, which runs every few milliseconds. During this time, depending on the size of the task program, a few dozen up to a few hundred trace operations are recorded. As an example, let us assume the average byte count per trace instruction is 16 bytes. A task program running every 1 ms producing approximately 50 nodes per cycle generates 1000*50*16 = 781 KB per second, 46 MB per minute, and 2,68 GB per hour. Multiplying this by the number of tasks and acknowledging the fact that 50 nodes per cycle is a rather small program shows the scale of this problem.

On top of all this, when data is displayed on a screen in a graphical way, the loaded raw data from a disk is only a small portion of the data necessary to display it. Usually, transformations are performed to generate primitive shapes or new data structures.

In order to navigate through big data sources, the memory used has to be managed in an efficient way. This is done by using an intelligent caching mechanism, which only keeps parts of the data in memory and loads new parts as required. Besides loading raw data from disk, data is transformed into other formats or data structures. Combining these two basic tasks, caching and transforming data, into a high level API is known as data virtualization [8] (see Figure 4.1).

In the tool at hand data virtualization forms a layer which is built on top of low-level data access APIs. Instead of accessing data sources directly, it hides implementation details and offers access to all the data at any time and at any position. The user of such an API only knows how much data there is, but does not care about how it is loaded, managed or generated.

The following sections show how multiple layers of data virtualization were implemented in order to access and generate data from traces. Section 4.1 starts with a description of how raw trace data is represented in memory. From this representation various definitions of time are introduced in Section 4.2. Section 4.3 then describes how executed code of the program run is reconstructed.



(a) Direct access to a data source (b) Virtualized access to a data source

Figure 4.1: Difference between direct access and virtualized access to data

4.1 Combined Trace

As described in Section 1.2, Capture & Replay produces a combined trace file of a program run which contains the trace entries of all tasks, including task switches between them. It is saved in a custom binary format storing a list of entries. An entry begins with a byte marker, indicating its type. The size of the remaining entry data depends on various factors. Not all entry types have a fixed size. Entries storing variable values do not store any type information but just raw data. This means a trace file is not readable on its own. Variable information, such as type and name are stored externally in a data structure named `VariableInfoStore`. This structure is generated during instrumentation. It must be queried every time a value of a variable is read in order to determine its type and therefore its byte size.

4.1.1 Nodes

Each entry is loaded into memory as `Node` object which is the basic unit of information while processing the trace. Each entry type has an equivalent `Node` type.

Table 4.1 lists all types of `Nodes` and summarizes their function. It also introduces an abbreviation which is used in the following sections to illustrate different trace nodes. A *TaskSwitch* is represented by a `NodeTaskSwitch` object which stores the next active task id. *BeginCycle* and *EndCycle* entries are represented as `NodeBeginCycle` and `NodeEndCycle` objects. Both contain a timestamp. *Read* and *Write* entries are represented by `NodeRead` and `NodeWrite` objects, each containing a reference to variable information, a position id and the value. `NodePosition` and `NodeMarker` contain the position id of *Position* and *Marker* entries. The only difference between the two is that markers can be given a descriptive name by the developer of the PLC application.

Each node is associated with its node index, which is a global zero-based counter. During a

Node type	Short form	Description
NodeTaskSwitch	TSx	Represents a task switch. Stores the id x of the next active task
NodeBeginCycle	BC	Beginning of a new task cycle, including a time-stamp
NodeEndCycle	EC	End of the current task cycle, including a time-stamp
NodeRead	Rx	Stores the value of a variable, specified by its variable id, which is read at a given position id x
NodeWrite	Wx	Stores the value of a variable, specified by its variable id, which is written at a given position id x
NodePosition	Px	Stores a position id indicating which code block has been visited
NodeMarker	Mx	Similar to a NodePosition, however, its position id x can be given a descriptive name

Table 4.1: Node types stored in a trace file

replay of a program, nodes are consumed one after another by instrumented statements. Each of them is owned by the task which was specified in the previous task switch. Figure 4.2 shows two tasks and nodes at the beginning of a sample trace file.

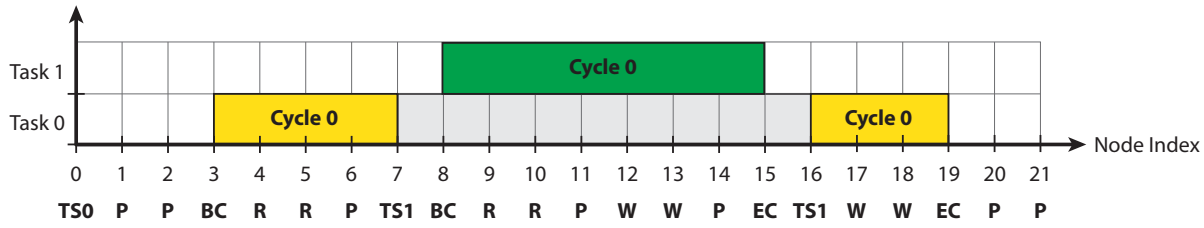


Figure 4.2: Beginning of a combined buffer containing traces from two tasks

Note that the first node, node index 0, must always be a task switch node. This is necessary, in order to know which task is executed first. In this example, Task 0 was executed at the beginning of the program run. Its first task cycle starts at node index 3 and ends at node index 19. During this time, it is interrupted by task 1, whose cycle starts at node index 8 and ends at index 15.

All nodes after a *BeginCycle* and before the following *EndCycle* of the same task are part of a task cycle. In the following this time span is called the *active region* of a task cycle for that task. All other nodes, like the *Position* nodes at index 1, 2, 20, and 21 can be ignored. They are the result from instrumentation of statements which are not part of the original PLC application's source code. That means, they have no meaning to developers and do not affect the execution of the program. In Figure 4.2, the active region of the first cycle of task 0 is marked with a yellow background, while the active region of cycle 0 of task 1 is colored green.

4.1.2 Chunks

Looking at Figure 4.2 and considering the active regions of each task, it can be seen that task switches can interrupt active regions. This interruption is illustrated by a light gray background between nodes 7 and 16. That means, the active regions are split into parts where task switches occur. These parts are called *chunks* in the following and are basic units which are further analysed. Thus each task cycle's active region is made up of one or more chunks. All chunks are associated with a chunk index, which is a zero-based global counter. Note that global in this context means, independent of the task owning it.

Figures 4.3-4.6 give examples how chunks emerge. The number of chunks largely depends on the location of the task switches. In Figure 4.3 both task cycles are executed after one another, without interruption. Each cycle's active region therefore has one chunk.

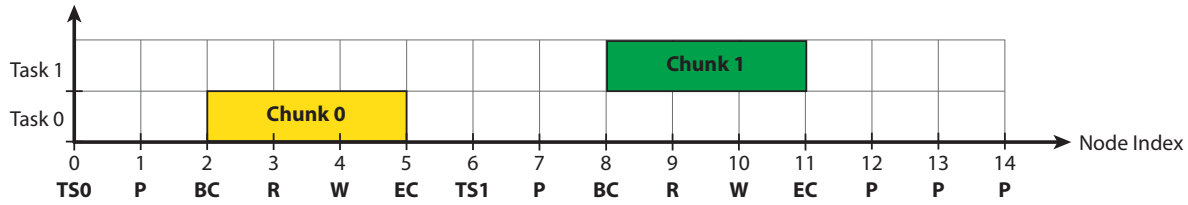


Figure 4.3: Two uninterrupted task cycles

Figure 4.4 illustrates how task 0 is interrupted by a complete cycle of task 1. This interruption takes place at node index 4 and lasts until node index 10. The first chunk starts at the *BeginCycle* of task 0 at node index 2 and lasts until the task switch at node index 4. Task 1 is not interrupted, which means its active region is the second chunk. The third chunk is the second part of the active region of task 0. It starts from the task switch at node index 10 and lasts until the *EndCycle* at node index 12.

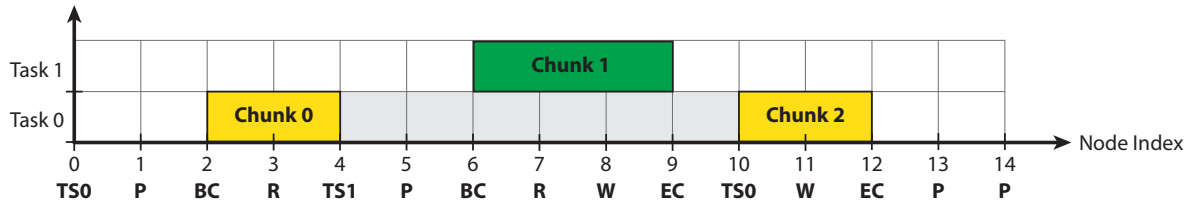


Figure 4.4: One task interrupting another in the middle of execution

Figure 4.5 shows a special case which occurs, when a *BeginCycle* is directly followed by a *TaskSwitch*. Because of the definition of chunks, the region between *BeginCycle* and *TaskSwitch* is seen as part of the active region and therefore becomes a chunk. This happens despite the fact, that between these two nodes there are no other trace operations. The reason why this definition was chosen is because it gives these short regions a node index range of length 1, which is later used by the visualization. Other definitions would introduce chunks with zero length, which produces far less compelling visualizations.

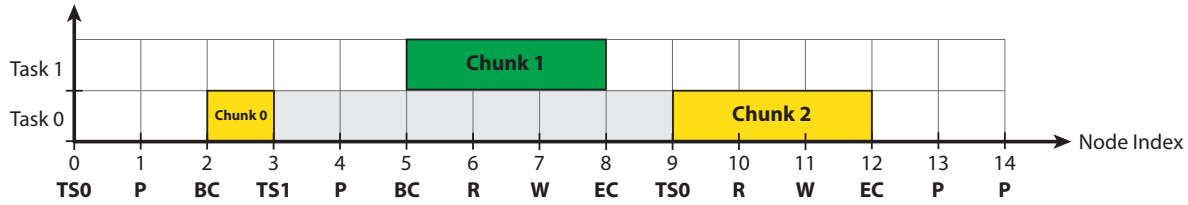


Figure 4.5: One task interrupting another at the beginning of execution

A similar case can be seen in Figure 4.6. In this case, task 0 is interrupted before the end of its cycle. The *EndCycle* at node index 12 is delayed and preceeded by a *TaskSwitch*. This is also seen as chunk.

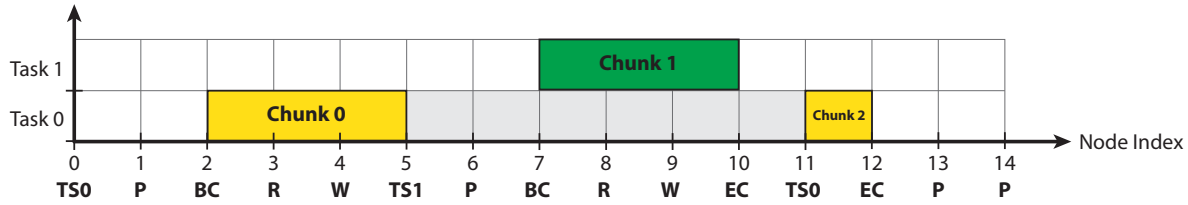


Figure 4.6: One task interrupting another at the end of execution

Finally, chunks do not have to contain *BeginCycle* or *EndCycle* nodes at all. In figure 4.7, chunk 2 is only confined by two task switches at node indices 9 and 11.

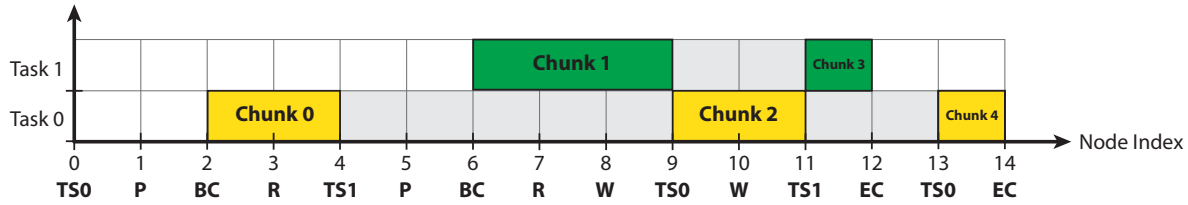


Figure 4.7: Multiple interruptions during one cycle

4.1.3 Accessing the combined buffer

As one can see, nodes and chunks are two different views of the trace data.

- One view sees the entire trace as a list of nodes, each independent of each other, but owned by a specific task. A node index specifies the position inside this list.
- Another view is grouping nodes to chunks. Each chunk is part of a task cycle and consists of one or more nodes.

In both cases, accessing a specific node at a given node index or a chunk at a given chunk index requires appropriate information. Nodes, for instance, have a variable byte size. Thus, the byte offset of a node can not be calculated by multiplication of the node index and some

fixed size. Its position depends on the sizes of all previous nodes. Therefore getting the 1000th node requires information about the size of the other 999 nodes.

This is even a greater problem when trying to access consecutive chunks. Chunks do not always follow each other. Sometimes nodes are between them, which are not part of any active region.

Accessing nodes and chunks randomly in such a file format therefore becomes very costly. What's necessary is a way to limit the amount of data being traversed in order to access a specific node or chunk. To do this, several steps have to be taken.

`TraceState` and `NodeEnumerator`

It is first recognized that if we traverse a trace node by node, from beginning to end, each position that is visited has a unique state. This state consists of several counters and flags, which define the location inside the trace and the application state. Such a state is represented by a `TraceState` object. Figure 4.8 shows the properties of the `TraceState` class.

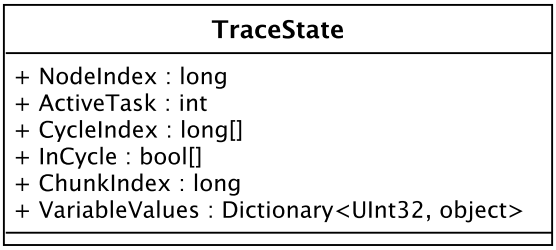


Figure 4.8: UML diagram of the `TraceState` class

These properties are:

NodeIndex While processing, this counter is incremented after each node.

ActiveTask This state variable is used to keep track of the active task index, which is changed by *TaskSwitch* nodes and defines which task owns the current node.

CycleIndex Each task executed performs several cycles, starting with a *BeginCycle* and ending with an *EndCycle* node. The `CycleIndex` array stores the increasing cycle index of each task. A task's cycle index is increased after its *EndCycle* node is processed. This array gives us an idea in which cycle each task is at the current processing position.

InCycle Besides counting cycles, it is also necessary to keep track of which tasks are currently active. This is done by setting `InCycle[ActiveTask] = true` at a *BeginCycle*, and `InCycle[ActiveTask] = false` at the *EndCycle*. By checking the `InCycle` flags, it becomes easy to determine which tasks are interrupted.

ChunkIndex Using the state variables introduced above, one can determine when a new chunk is started. The `ChunkIndex` counter stores the index of the current chunk. It has to be

increased if a *EndCycle* node is reached or during a *TaskSwitch*, if the currently active task is still in a cycle. Note that this chunk index only becomes valid once the active task enters an active region.

VariableValues The current value of each traced variable is stored in a dictionary. It maps the variable's `UInt32` pointer to its value. Every *Read* and *Write* node accessing such a pointer updates this value.

Node iterator

Iterating over nodes is a common task. In order to simplify the process, a special iterator was introduced. `NodeEnumerator` implements the `IEnumerator<Node>` interface and enhances it by adding access to the current state. Listing 4.1 outlines its usage. It is constructed with a starting state and an `IEnumerator<Node>` which is used as node source. While iterating, the iterator updates the trace state. Both `State` and `Current` properties of the iterator are updated by the `MoveNext` call.

```
var nodeEnumerator = GetNodeEnumerator();

while (nodeEnumerator.MoveNext()) {
    var node = nodeEnumerator.Current;
    var state = nodeEnumerator.State;

    // process node and its state ...
}
```

Listing 4.1: Usage of `NodeEnumerator`

Index structure for the combined trace

The introduction of a `TraceState` at each node index allows generating an index structure which accelerates accesses to the combined trace file. This data structure is implemented by the `TraceIndexList` class and is illustrated in Figure 4.9. It is a list of `IndexNodes`, each storing the byte offset and the `TraceState` of specific nodes in the trace file. Moving to the byte offset of the trace file and using the `TraceState` to initialize a `NodeEnumerator` allows resuming node enumeration at the specific locations without processing all other previous nodes.

The data between two succeeding `IndexNode` locations can be seen as a block of the trace file. These blocks have been named *data blocks*. Figure 4.9 shows how `IndexNodes` partition the trace into multiple data blocks. Data block 0 spans from node index 0 to p, data block 1 from index $p+1$ to q and data block 2 from index $q+1$ to r. If a trace is split up into N data blocks, `TraceIndexList` contains $N+1$ `IndexNodes`. The last `IndexNode` contains the `TraceState` at the end of the trace. This can be used to determine the total number of nodes, chunks, and cycles.

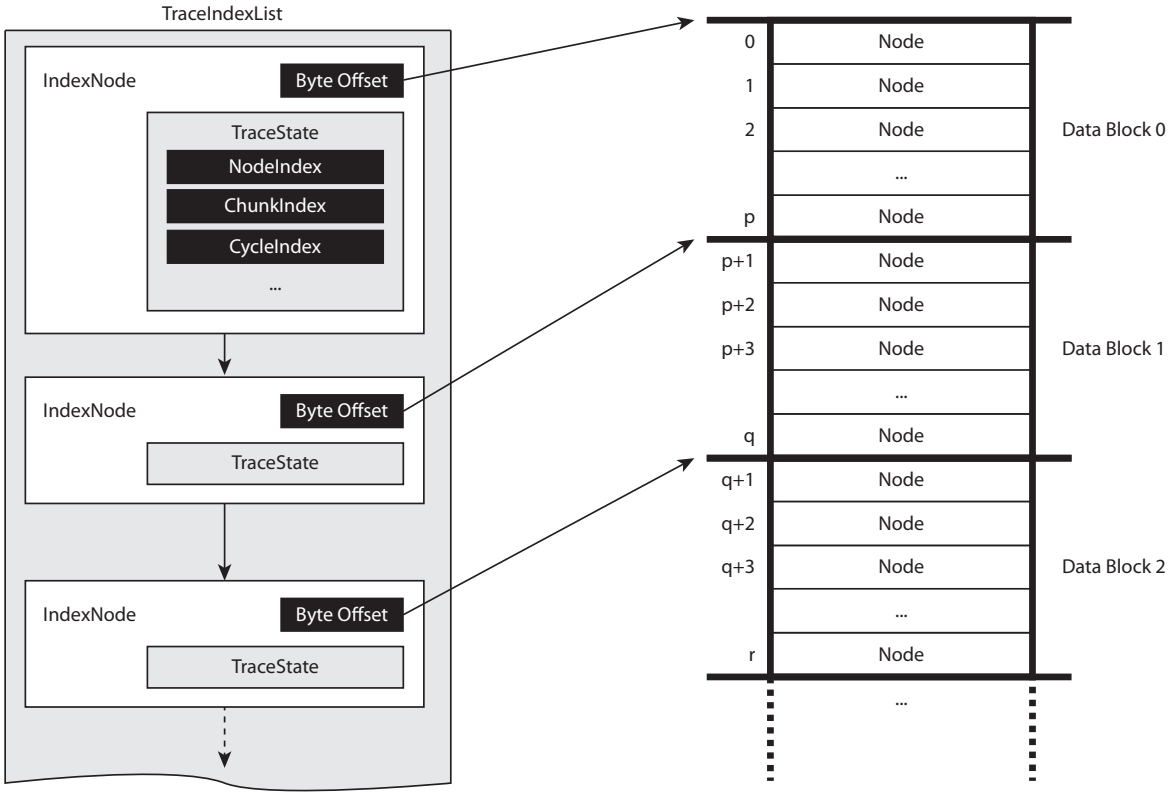


Figure 4.9: Partitioning of the combined trace using `IndexNodes`

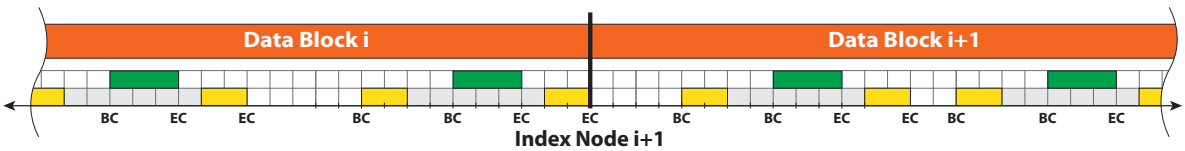


Figure 4.10: A valid data block boundary

The placement of each `IndexNode` is important. In later processing stages each data block is treated as an independent unit of information, starting with a given `TraceState` and containing a certain number of nodes. There should be no data dependencies between data blocks. `IndexNodes` are therefore created approximately every 10,000 nodes in such a way that all contained task cycles are complete. This means, that all chunks of a task cycle are in the same data block (see Figure 4.10).

Accessing a node with a specific `NodeIndex` is reduced to finding the data block which contains that node and iterating to it. The same applies to accessing a specific chunk with a `ChunkIndex` and a task cycle with a `CycleIndex`.

4.1.4 Virtualized access to nodes and chunks

By using data blocks introduced in the previous section, data virtualization for nodes and chunks can be implemented. Figure 4.11 shows the architecture used to access nodes and chunks. In the following, each class of this architecture is described:

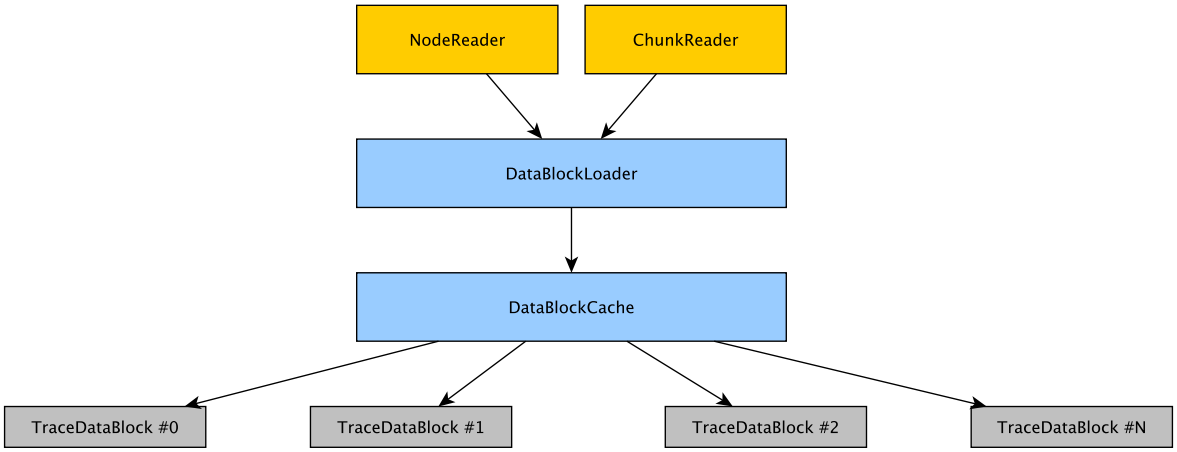


Figure 4.11: Architecture for node and chunk access

TraceDataBlock Loading a data block creates a new object of the `TraceDataBlock` class. Figure 4.12 shows the most important properties of this class in an UML diagram. Each data block is referenced by its data block index which is stored by the `Index` property. The `StartState` property stores the `TraceState` of the first `IndexNode` which defines the data block's starting point. `Node` objects are created by loading the trace entries from the binary trace file and stored in the `Nodes` collection. `Chunk` objects are generated on demand during the first access of the `Chunks` collection. The purpose of the `TimeData` property is explained in Section 4.2.

DataBlockCache Data block requests are processed by a class named `DataBlockCache` seen in Figure 4.13. It implements a simple cache of data blocks. The current implementation of `DataBlockCache` allows loading and keeping a fixed amount of data blocks in memory.

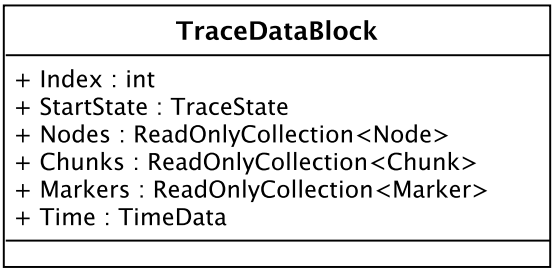


Figure 4.12: Interface of the `TraceDataBlock` class

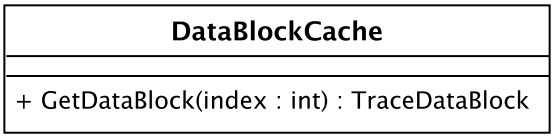


Figure 4.13: Interface of the `DataBlockCache` class

Loading more data blocks unloads others. Which blocks are unloaded is based on a last access strategy. Frequently accessed data blocks are kept in memory, less often used blocks are discarded.

DataBlockLoader This class is responsible for translating access requests to data block requests. Its interface, shown in Figure 4.14, contains methods for accessing data blocks which contain a specific `TraceState` property value (`NodeIndex`, `ChunkIndex`, ...) or a timestamp. Note that *compressed time* is described in Section 4.2.2. Both `NodeReader` and `ChunkReader` use this class to access their data.

NodeReader is an API class which allows accessing single nodes and ranges of nodes. Its UML class diagram is shown in Figure 4.15. The total amount of nodes in a trace file is accessible through the `Count` property. Individual nodes are accessed through their `NodeIndex`. The `TraceState` of such a `NodeIndex` can be retrieved by the `GetState` method. All other methods return an `ITraceNodeSource` object which is used to retrieve a `NodeEnumerator` for a given `NodeIndex` or time range.

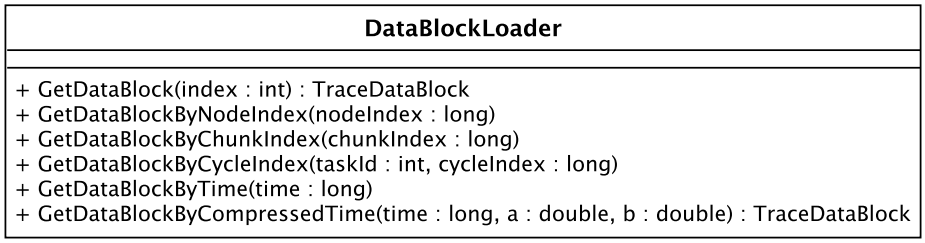


Figure 4.14: Interface of the `DataBlockLoader` class

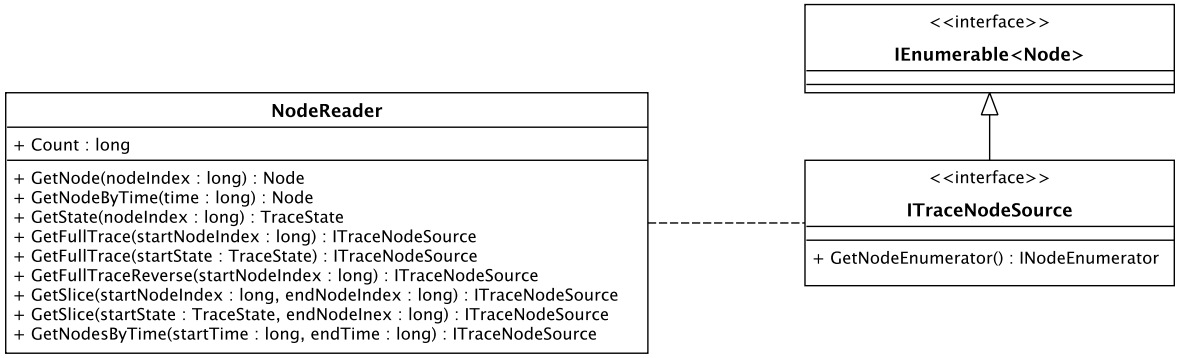


Figure 4.15: Interface of the `NodeReader` class

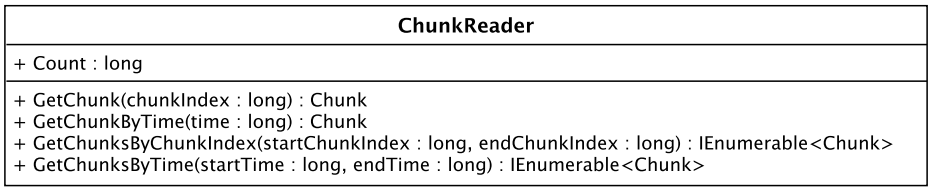


Figure 4.16: Interface of the `ChunkReader` class

ChunkReader is an API class used to access individual chunks and ranges of chunks. Figure 4.16 shows the UML diagram of this class. A chunk is accessed by its `ChunkIndex`. Ranges of chunks are requested by specifying a `ChunkIndex` range or a time range. The result of such a request is an `IEnumerable<Chunk>` object.

4.2 Time

Although nodes and chunks provide enough information to create visual representations of the trace data, displaying ranges of node indices and chunks is not very intuitive. What is missing is a natural mapping of node indices and chunk indices to time. This, however, generates a completely new problem: what's the time in a node-based trace?

4.2.1 Real time

Luckily, some nodes like *BeginCycle* and *EndCycle* actually store timestamps during tracing. A combined trace's time range is defined by its first and last node containing a timestamp. These timestamps are considered to be in *real time*, which means they are wall clock times during recording. Therefore, it is possible to relate times to specific node indices. To expand this to all other node indices, a linear interpolation is used. Nodes which store a timestamp are used as control nodes of the interpolation (see Figure 4.17). Since the combined trace is already split up into data blocks, interpolation of time is also done on a block basis. To do this, each data block's time range must first be defined. Each block contains complete cycles, which means the first

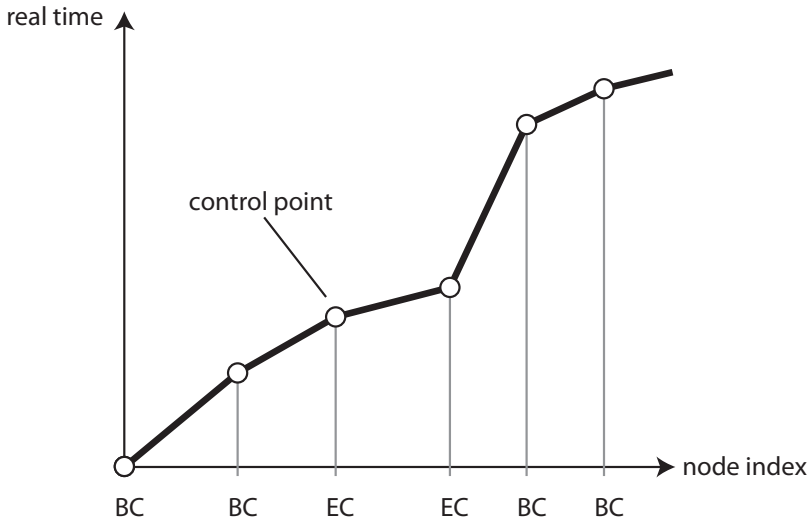


Figure 4.17: Linear interpolation of node index to time

BeginCycle and last *EndCycle* node of a data block can be used as boundaries. It is sufficient to store the first timestamp of a data block in its `IndexNode` of the `TraceIndexList`. This gives us the ability to search for blocks with a given time range.

Once a data block is loaded as an object of `TraceDataBlock`, it provides a data structure named `TimeData` which interpolates timestamps to node indices and backwards.

4.2.2 Compressed time

Displaying objects like nodes and chunks using realtime might have its purposes, however it quickly became clear that using such a fine resolution of time is not very useful for display. Tasks usually run every few milliseconds, yet only need a few nanoseconds to perform their calculations. This means that most of the time, an application is waiting for its next execution. The amount of time which is actually part of an active region of a task is often much smaller than the amount of time it is waiting. Thus, since we are more interested in what happens during the active times, a linear time scale does not fit our requirements.

What is therefore done is the introduction of a time measure θ with two components, the *incompressible time* u and the *compressible time* c. The incompressible time u measures the progression of realtime during active states. The compressible time c elapses outside of active region. The sum of both is equal to the realtime difference Δt between two timestamps t_1 and t_2.

$$\theta = (u, c)$$
$$\Delta t = t_2 - t_1 = u + c$$

As both names imply, using these components of time allows compressing parts of it during

display. This can easily be accomplished by using a scaled time offset Δp which results from applying different scaling factors a, b to the two time measures.

$$\Delta p = a \cdot u + b \cdot c$$

Partitioning of time into compressible and incompressible time is done during generation of a data block's `TimeData` structure. While iterating through all timestamps of a data block, it is checked if the difference between the current timetamp and the last timestamp is greater than a fixed threshold. If so, time between these two timestamps is considered to be compressible time. Of course, since each data block has a compressible time and incompressible time offset, this information also needs to be saved in its `IndexNode`.

4.2.3 Accessing the trace through time

In Section 4.1.4, Figure 4.15 and Figure 4.16, we have seen that both the `NodeReader` and the `ChunkReader` allow accessing the trace by timestamps. Both support accessing single elements through timestamps, as well as returning regions of elements in a given time range. Accessing an arbitrary number of nodes and chunks of any time interval in the trace is very powerful, however, it must be used cautiously. Limiting the maximum amount of nodes and chunks is crucial, because even though access is optimized and cached, processing a great number of elements slows down display time.

4.3 Code reconstruction

As described in Section 1.2, instrumentation actually gives us enough information about a program run, so that the executed source code can be reconstructed. This is done on a chunk basis. Since a trace contains many thousands of cycles, each containing one or more chunks, this creates millions of lines of code, which have been executed during program run.

In order to navigate through these lines of code, another layer of data virtualization, built on top of the first one is necessary. The following sections dive into the reconstruction of executed source code and the steps necessary to allow virtualization of code lines.

4.3.1 `CodeTree` data structure

Generation of the executed code requires access to the the entire instrumented source code of the application. The most obvious way of accessing this data is through CoDeSys's own object data structures. However, instrumentation is done transparently during compilation, which means it is not persistent in any form other than the compiled machine code. Moreover, CoDeSys's own object structure is saved in a proprietary project format which is not easily accessible to external applications. Therefore, an independent object infrastructure was needed.

The `CodeTree` data structure is generated after instrumentation for detailed trace. It consists of a list of tasks and an object dictionary, called `ObjectStore`, which stores all POU objects of an application and maps them to a GUIDs (Genuine Unique Identifier). CoDeSys itself uses GUIDs to differentiate between objects. Using the same GUIDs provides some compatibility between the two object representations.

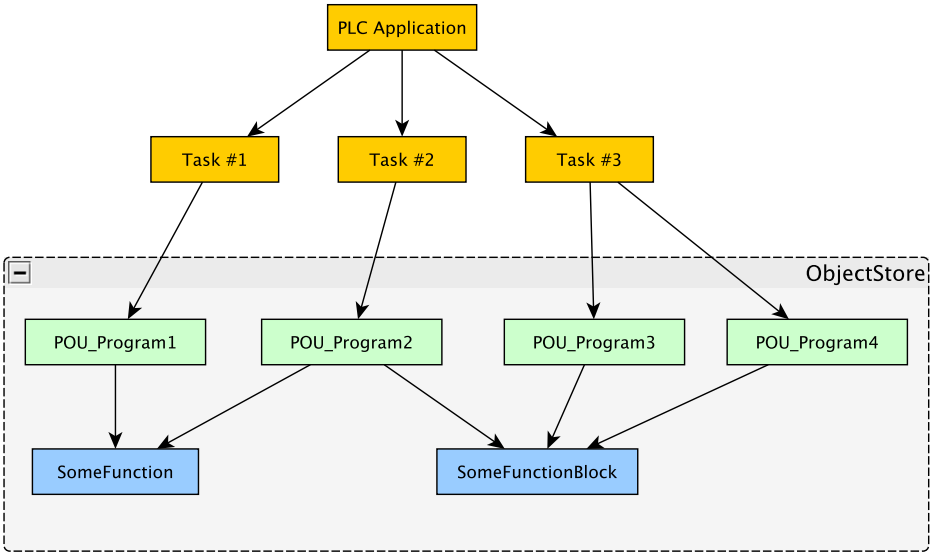


Figure 4.18: Program structure of a PLC application

Figure 4.18 illustrates how an application is made up of multiple tasks, which call one or more programs. Each program can call various functions or instantiate function blocks.

The abstract syntax tree of the source code is used to generate POU objects which are then stored in the object store. Additionally, the `CodeTree` contains a list of all tasks, which call one or more program POUs.

4.3.2 `CodeTree` representation of source code

Listing 4.2 shows a simple program written in structured text which is used to illustrate the `CodeTree` representation of source code. It is a small counting application, which sets its `finished` flag once its counter hits 100.

```
PROGRAM PRG_Counter
    VAR
        counter : INT;
        finished : BOOL;
    END_VAR

    IF counter < 100 THEN
        finished := FALSE;
        counter := counter + 1;
    ELSE
        finished := TRUE;
    END_IF
END_PROGRAM
```

Listing 4.2: Counter program example

The program is instrumented for detailed trace as shown in Listing 4.3. A *Position* operation is always added at the beginning of a POU. It marks the beginning of its code block. The beginning of the THEN and ELSE branch of the IF statement are also marked with a *Position* operation. Finally, task programs must be delimited by a *BeginCycle* and a *EndCycle* operation.

```
Trace.Position(1);
Trace.BeginCycle();
IF counter < 100 THEN
    Trace.Position(2);
    finished := FALSE;
    counter := counter + 1;
ELSE
    Trace.Position(3);
    finished := TRUE;
END_IF
Trace.EndCycle();
```

Listing 4.3: Instrumented code of the counter example

Figure 4.19 shows how the sample program is transformed into a tree structure. Each statement is transformed into a `CodeElementNode` object. Yellow nodes are `CodeElementNodes` produced by regular statements, while green nodes are `CodeElementNodes` of instrumented statements. Statements containing trace operations, such as *Position*, *Read*, and *Write*, store a

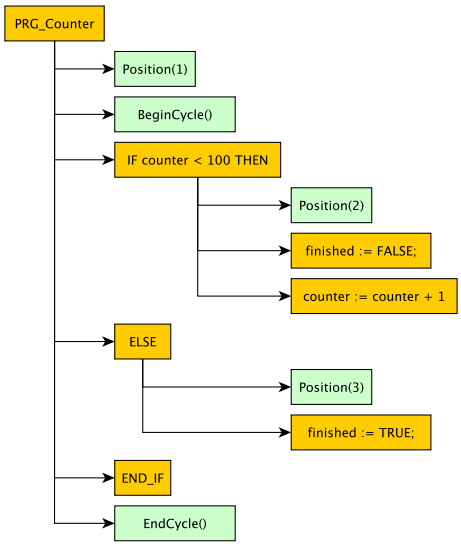


Figure 4.19: Static Code Tree structure of the Counter program

list of position ids identifying trace operations contained in that statement. This information is later used to parse the tree structure and detect which parts are executed.

Control structures, such as a IF statements, contain multiple branches. IF, ELSE and END_IF are added as individual `CodeElementNode` objects. The statements of the THEN branch are added as children of the IF node, while the statements of the ELSE branch are added as children of the ELSE node.

Calling other POUs

In the next example it is shown how calls to other POUs are saved in the `CodeTree` structure. Listing 4.4 shows a modified version of the previous example code. It now contains a function call to the `calculateNextCounter` function. The implementation of this function is shown in Listing 4.5. This function simply takes one argument and adds 1 to it.

```
Trace.Position(1);
Trace.BeginCycle();
IF counter < 100 THEN
    Trace.Position(2);
    finished := FALSE;
    counter := calculateNextCounter(current := counter);
ELSE
    Trace.Position(3);
    finished := TRUE;
END_IF
Trace.EndCycle();
```

Listing 4.4: Instrumented code of the counter example

```
FUNCTION calculateNextCounter : INT
    VAR_INPUT
        current : INT;
    END_VAR

    Trace.Position(10);
    calculateNextCounter := current + 1;
END_FUNCTION
```

Listing 4.5: Definition of the `calculateNextCounter` function

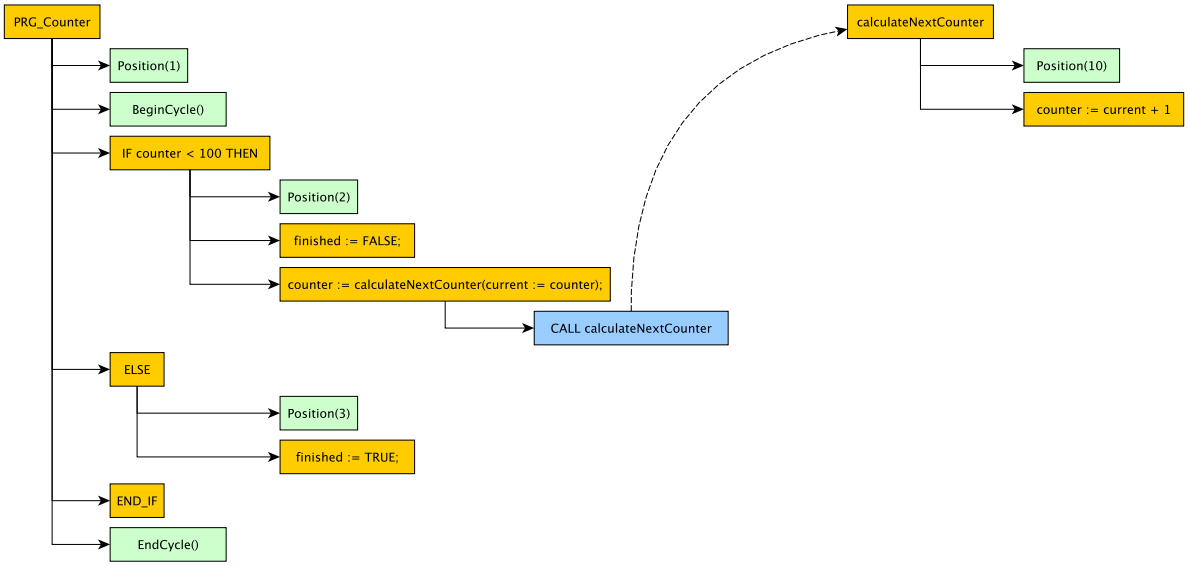


Figure 4.20: Sample program with function call

Figure 4.20 illustrates how the function call is represented in the tree structure. The assignment which contains the function call has one child. This child is a `CallTreeNode` object, referencing the code tree of the `calculateNextCounter` function POU.

4.3.3 Reconstruction of executed code

During a program run, not all lines of code are evaluated. Control flow statements activate or deactivate portions of code. This can be monitored by looking at the recorded trace operations between *BeginCycle* and *EndCycle*.

The Counter example program has two possible execution paths which can be taken. Either the THEN branch of the IF statement is executed or the ELSE branch. Each path leaves its own unique trail of instrumented position ids. The THEN branch is visited if *Position* 2 is reached. The ELSE branch is visited if *Position* 3 is reached. This leads to two possible node sequences, which are illustrated in Figure 4.21.

By using the information saved in the `CodeTree` structure, a filtered code tree can be generated based on the visited path. Each filtered code tree consists of `FilteredCodeElement` objects and only contains statements which have been visited. In the Counter example this

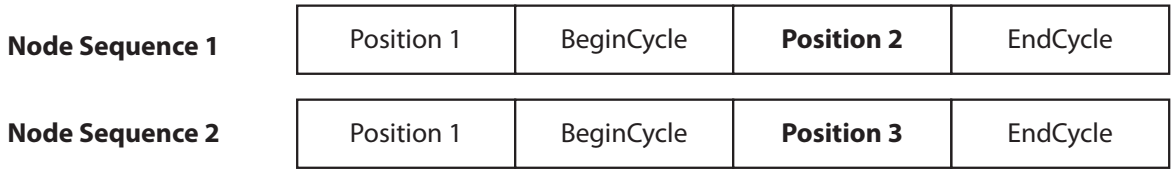
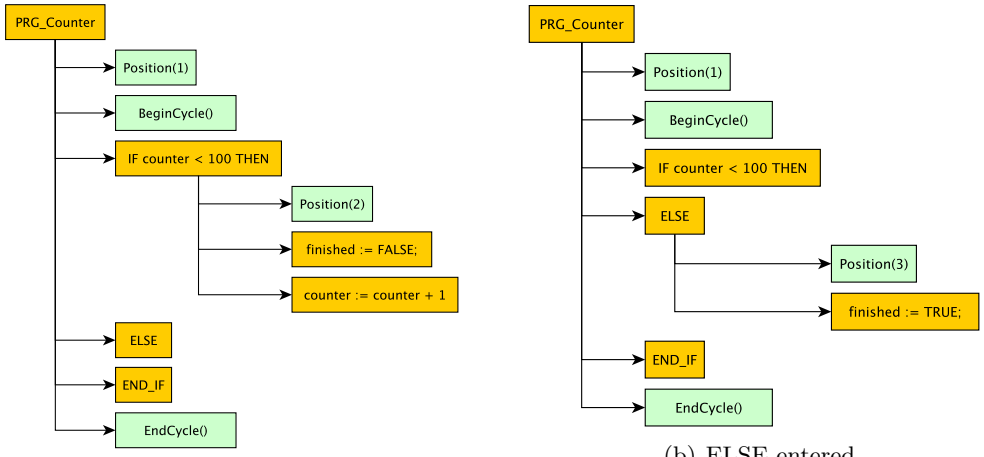


Figure 4.21: Two possible node sequences of the Counter example



(a) IF entered

(b) ELSE entered

Figure 4.22: Filtered code trees of the counter example

leads to two possible filtered trees, which are shown in Figure 4.22. Such filtered trees form the basis for displaying executed code in the execution transcript view described in Section 6.4.

In order to show the executed code, many `FilteredCodeElement` objects have to be created. These trees have a significant impact on memory usage. Since most information is verbose, a filtered code tree does not store any code tree information redundantly. Instead, it references the original source `CodeElementNode`.

4.3.4 Sequential Function Charts

The most frequently used language in the case studies, besides ST code, were sequential function charts (SFC). This graphical language consists of states, actions and transitions. At compile time, CoDeSys transforms this graph into ST code.

Although this generated ST code can be processed as described above, this adds huge amounts of clutter around very simple code. It was therefore decided, that the generated code of SFCs should remain invisible to developers. Thus, SFCs are treated differently during code reconstruction.

This required saving more information about SFCs. POUs implemented in SFC are stored as `SFCObjects` in the `ObjectStore`. These objects contain the instrumented generated source code as code tree, but additionally keep structural information of the SFC graph. This includes

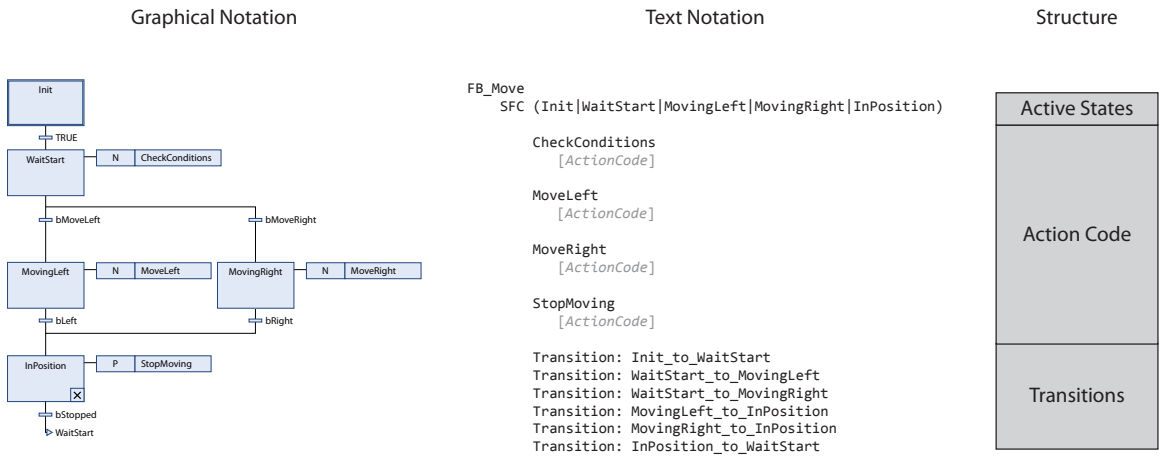


Figure 4.23: Graphical SFC notation mapped to a text notation

information about which states exist and which transitions and actions are used.

The goal was to present the executed source code of a SFC in a compact text-based format, which would naturally fit into the remaining filtered code tree. Figure 4.23 shows how a simple SFC is transformed into a textual tree representation. On the left side of the figure is the graphical notation of a SFC. It consists of five states: `Init`, `WaitStart`, `MovingLeft`, `MovingRight` and `InPosition`. In the middle is the text notation which is built by creating an alternate filtered code tree. The right side of the figure illustrates the structure of the text notation, which is described in the following:

Active states: The top part of the text representation of a SFC execution begins with the name of the SFC function block. It is followed by a line which starts with *SFC* and contains a list of state names in parentheses, separated by vertical lines. During execution, only one or more of the SFC states are active. In the text notation, this is illustrated by only showing the active states' names in the parentheses.

Action code: Active states of a SFC execute actions. For example, the `MovingLeft` state executes a continuous action named `MoveLeft` while active. In the generated ST code, this means the SFC POU calls the action POU. Each executed action POU produces an entire filtered code subtree of its own. This executed action code is added in the middle part of the text notation. Each action's name is appended and its subtree of executed code is added as child. Action calls have a unique execution order in the generated ST code. This order is preserved by the text notation, ensuring that code at the top was executed before code at the bottom.

Transitions Finally, the bottom part of the text notation lists the names of transitions, which were active during execution.

What is now necessary is a mapping between the executed ST code to this compact text format.

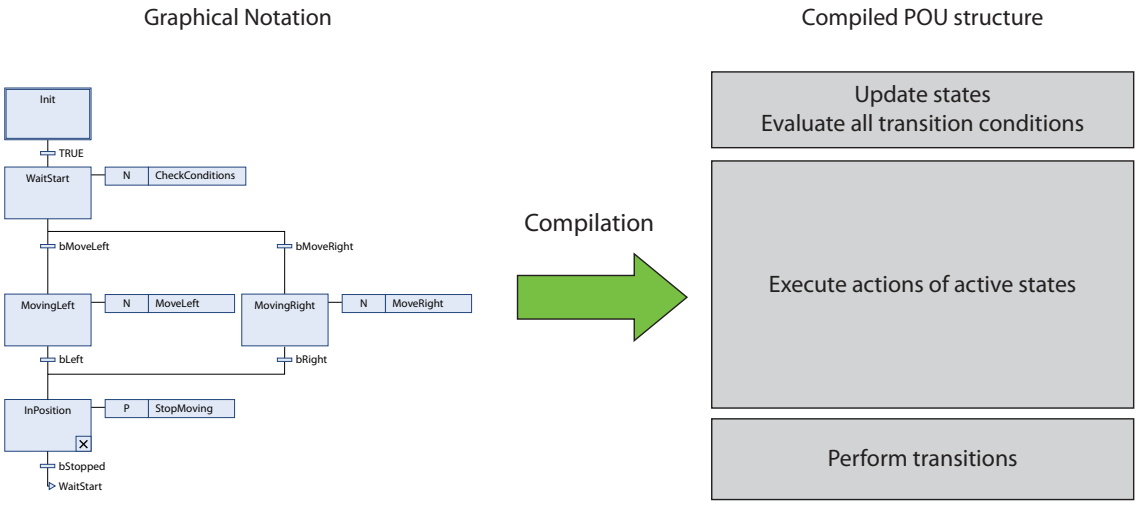


Figure 4.24: Structure of the compiled POU generated from the graphical notation

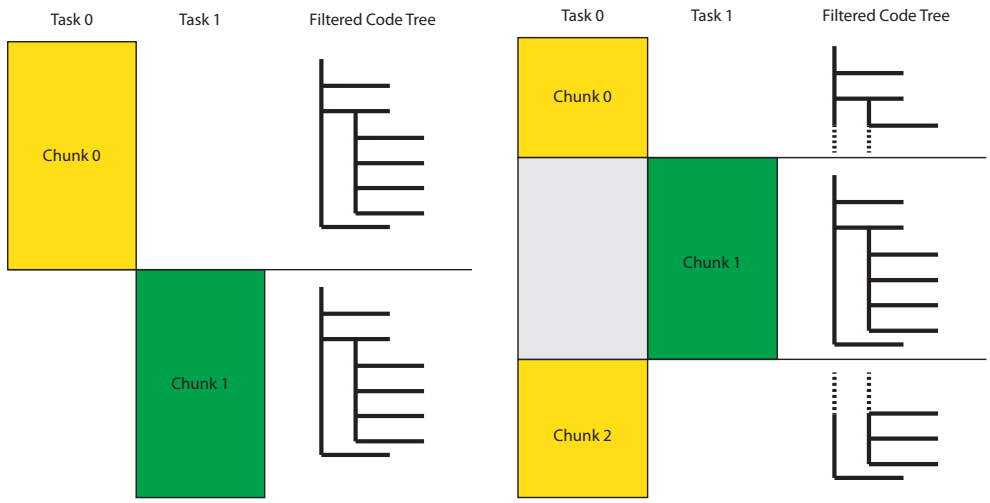
In order to accomplish this, the original mapping of SFCs to ST code was reverse-engineered. By understanding the structure of the generated code, specific control flows could be mapped to specific states, actions and transitions. Figure 4.24 shows how the generated code is organized. Three main parts of the ST code could be identified. The top part of the code is a series of IF statements, responsible for updating states and evaluating transition conditions. It is followed by cascading IF statements, representing the different branches of the SFC and containing the action calls. The final part of the code is responsible for moving from one state to another, based on transitions.

Detecting if a state was active or not turned out be as simple as checking whether a certain THEN branch of an IF statement was visited. Because this code is instrumented, this means checking whether a specific *Position* statement was reached. Similar mappings can be found for transitions and actions.

4.3.5 Processing multiple tasks and task interleaving

Up until now we have only considered single task programs during code reconstruction. Processing these can be done using a very simple parser, which reads `Nodes` and outputs the filtered code tree of each task cycle. However, as described earlier, the combined trace is a mix of many tasks, which sometimes even interrupt each other. This requires generating only parts of executed code, not complete cycles. These parts are chunks which have been introduced in Section 4.1.2. Code reconstruction of multiple tasks can therefore be seen as generating filtered code trees out of chunks.

Figure 4.25a illustrates the result of code reconstruction if two tasks do not interrupt each other. Each task cycle runs to completion. Therefore each cycle is exactly one chunk. These chunks each create a filtered code tree, which contains the executed statements of the entire task



(a) Uninterrupted execution (b) Interrupted execution

Figure 4.25: Interruptions affect filtered code tree

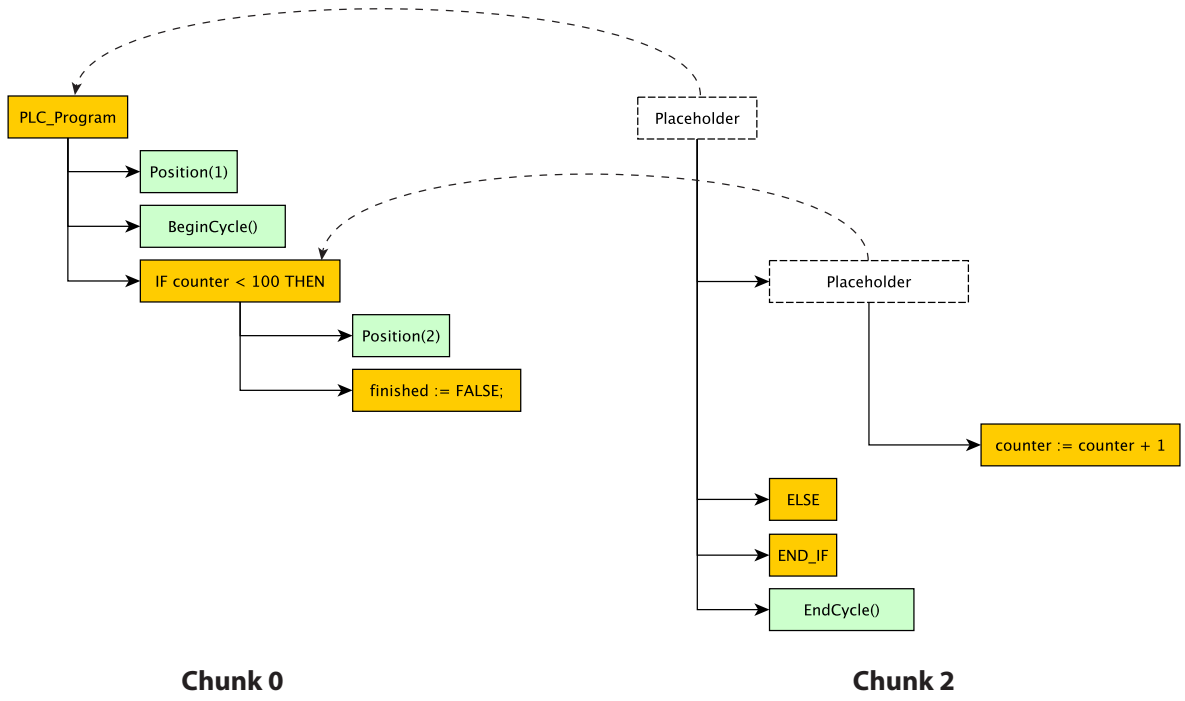


Figure 4.26: Placeholders ensure that children of other chunks stay connected ot the original tree

program.

In Figure 4.25b the cycle of task 0 is interrupted by task 1. It is therefore split up into two chunks. The filtered code tree generated by chunk 0 only contains statements which have been executed before the task switch. Chunk 1 is a complete cycle of task 1 and therefore produces a filtered code tree containing all executed statements of that task. Finally, chunk 2 is used to generate a filtered code tree, which contains the remaining executed statements of the task 0 cycle.

Note that the filtered code trees of chunk 0 and 2 are connected in a special way. To illustrate this connection, Figure 4.26 shows how the counter program could have been interrupted. It shows the filtered code trees of both chunk 0 and 2. The task switch occurred inside the THEN block of the IF statement, after the `finished` flag is set FALSE. Interruptions can happen at any level of the call hierarchy. The code tree of the chunk 2 must resume at the same level where the first tree ended. What is therefore done is duplicating the parent tree of chunk 0 and replacing it with placeholder nodes. These placeholders are invisible, yet connected to the original parent nodes. This way the call hierarchy is retained in the second tree.

`ChunkData`

Filtered code trees generated by code reconstruction are saved in objects of type `ChunkData`, which is shown in Figure 4.27. It has the following properties:

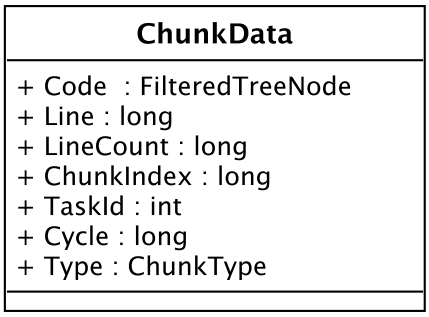


Figure 4.27: UML Diagram of the `ChunkData` class

Code: The `Code` property stores the root node of the generated filtered code tree.

Line: The global line number of the first line of this chunk. Unlike the local line number, which is a index in a POU's source code, the *global line number* counts all lines of code executed during a program run up to this point.

LineCount: The number of lines this chunk contains. This number might differ from the amount of tree nodes, because some tree nodes are marked as invisible.

ChunkIndex: The chunk index of this chunk.

TaskId: The task id which this chunks belongs to.

Cycle: This property stores the cycle index to which this chunk belongs.

Type: The type property is used to differentiate between chunks which are delimited by a task switch, a begin cycle, an end cycle, or any combination of these.

4.3.6 Virtualized access to lines of code

As mentioned earlier, reconstruction of executed code has an significant memory footprint. It is therefore necessary to apply the same caching strategy used by nodes and chunks. Executed code is reconstructed blockwise. Since data blocks contain complete cycles and their chunks, they can easily be processed independently. Figure 4.28 shows the architecture used to access individual lines of code and code chunks. Note that `DataBlockLoader` has been described in Section 4.1.4. In the following, the remaining two classes are described:

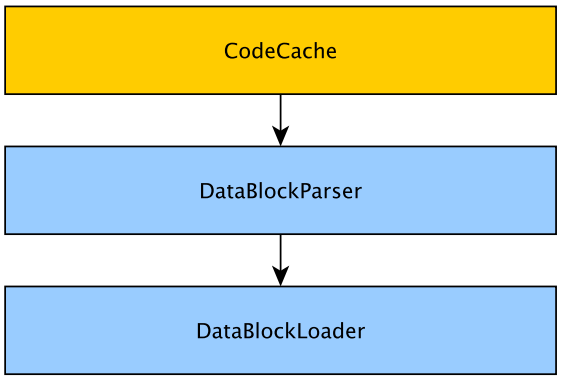


Figure 4.28: Architecture for code access

DataBlockParser This class is responsible for the code reconstruction of a single data block. This loading process of has been optimized. Figure 4.29 shows an overview of the different processing phases. While loading of nodes and chunks using the data layer is done on a single thread, code reconstruction is done in a multi-threaded way. To improve performance, code reconstruction spawns multiple worker threads. Each worker thread only processes chunks of a single PLC task and generates the filtered code trees of these chunks. Once all workers have finished, these trees are collected and sorted by their chunk index.

CodeCache `ChunkData` objects remain in memory as long as its data block is in memory. This task of loading and unloading the generated code of blocks is performed by the `CodeCache` class. Similar to `DataBlockCache`, it implements a last access strategy. The `ChunkData` objects of fixed amount of blocks can be kept in memory, objects of less frequently accessed blocks are unloaded. Internally it uses `DataBlockLoader` to load the nodes required to run code reconstruction. Figure 4.30 shows the public interface of this class. It allows

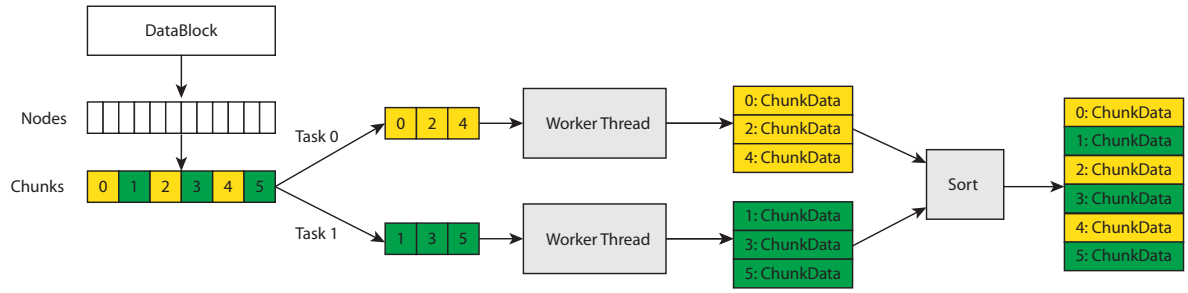


Figure 4.29: Code Reconstruction Overview

accessing the `ChunkData` generated by a chunk. Individual lines of code can be retrieved as `ChunkData` objects as well. In this case, the `Code` property of the `ChunkData` only contains the `FilteredTreeNode` which represents a single line of code.

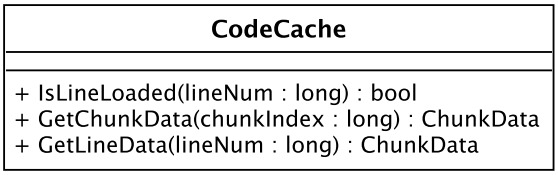


Figure 4.30: UML Diagram of the `CodeCache` class

Note that accessing individual lines of code requires knowing which data block contains these lines. This is done by precomputing the total amount of lines in each data block and saving the line offset of each data block as list. This list is then queried when looking for the data block which contains a specific line.

Outlook

In the previous sections, it was shown how various forms of data are generated, transformed and cached. Having virtualized sources of data is required for any data intensive application. Building UI applications which access this data pose yet another challenge. In the following chapters, the creation of user interfaces and data driven views using the Windows Presentation Foundation (WPF) is described.

Chapter 5

WPF

The CoDeSys Automation Platform is an IDE running on the .NET platform. Since this thesis started out by creating a plugin for this platform, there were two choices of how to develop new user interfaces. The two graphics toolkits provided by .NET are WinForms and Windows Presentation Foundation (WPF). Both allow the creation of graphical user interfaces (GUI). WinForms is a managed wrapper over the Win32 C API. For years this wrapper and its Win32 base have dominated Windows UI development. However since the introduction of .NET 3.5 and Windows Vista, a new way to write Windows applications has emerged. WPF is a graphical toolkit built entirely from scratch and built on a hardware accelerated DirectX 9 layer [21]. In WPF, user interfaces can be specified in a XML based markup language named XAML and are populated with data using data binding. Although CoDeSys is built on top of WinForms, it was decided to use the new and more powerful approach introduced by WPF. It seemed much more suitable for rapid prototyping of GUIs and creating graphic intensive visualizations, which was the main task of this thesis.

5.1 Defining user interfaces in XAML

In WPF user interfaces are built by creating and manipulating a visual tree. This tree can be defined programmatically and through the XML markup named XAML. To illustrate this, the definition of the window shown in Figure 5.1 are explained in detail.

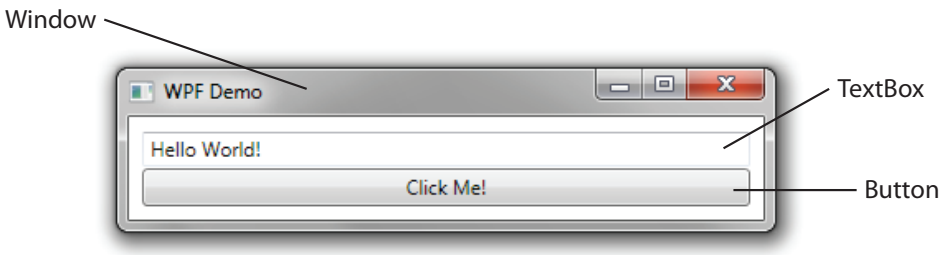


Figure 5.1: A simple WPF Window

```
<Window x:Class="WPFDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WPF Demo" Width="400" Height="100">
    <Grid Margin="8">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>

        <TextBox x:Name="txtOutput" Grid.Row="0"
                   HorizontalAlignment="Center" VerticalAlignment="Center">
            <TextBox.Text>Hello World!</TextBox.Text>
        </TextBox>
        <Button x:Name="btnClickMe"
                Click="OnClick" Grid.Row="1">Click Me!</Button>
    </Grid>
</Window>
```

Listing 5.1: XAML code of a Window, containing a button and a text box

One way of defining UIs is shown in Listing 5.1. It is the XAML definition of a window, contained in a file named `MainWindow.xaml`. Each XML tag usually stands for a class name and creates an object of that class. The properties of the class can be set by using attributes. In addition to inline attributes, they can also be set by adding a child node using the `<ClassName.PropertyName>` syntax (e.g. `<TextBlock.Text>`). In the example, the root element of the Window's visual tree is a `Grid` panel, which contains two rows, defined by two `RowDefinitions`. The first row contains a `TextBox` control, named `txtOutput`. The second row contains a `Button` control named `btnClickMe`.

This definition creates a new class, named `MainWindow`, which extends the `Window` class. While defining visual elements within the tree, each named object is added as instance variable of the `MainWindow` class. Since C# supports partial types [5], the class introduced in XAML can be extended using a partial class. This class is put into a file, which is called the *code-behind file*. Usually it has the same name as the XAML file but adds a language specific extension at the end. In this case, the code behind is defined in `MainWindow.xaml.cs`. Thus the XAML and C# file together form a new class.

XAML is used to layout the visual tree. The code-behind should be used to define the behaviour of the UI. In this simple example, the code behind shown in Listing 5.2 reacts to the `Click` event of `btnClickMe` and sets a new text on the `TextBox` control.

```
using System.Windows;

namespace WPFDemo {
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitComponents();
        }

        private void OnClick(object sender, RoutedEventArgs args)
        {
            txtOutput.Text = "Goodbye!";
        }
    }
}
```

Listing 5.2: Code-behind file of `MainWindow`

5.2 Creating controls

Windows are the root containers of all user interfaces elements. WPF provides many standard controls which can be used in combination with various layout containers to create complex interfaces. In object-oriented programming, there two main extension mechanisms in existing object structures: Composition and inheritance. The previous section showed how to create the visual tree of a window using composition of multiple standard controls. The window itself introduces a new class `MainWindow` as a subclass of the `Window` class.

The following sections outline how new controls are created in WPF. Section 5.2.1 and 5.2.2 describe how existing controls can be customized by setting their properties through *styles* and how to use *resources*. Section 5.2.3 briefly explains how new controls are defined.

5.2.1 Styles

Controls can be manipulated by setting their properties. However, a even simpler way to customize a standard control and group many property changes at once is by using styles. They allow adapting the look and feel of controls or even an entire UI application. A style is usually applied to a `TargetType`, such as a `CheckBox`. Figure 5.2 illustrates how a style may change the appearance of `CheckBox` control. On the left side, the text of the unchecked `CheckBox` control is colored in red. On the right side, the control is checked and its text is colored in green.

Listing 5.3 shows how this behaviour can be achieved by applying a style to a `CheckBox`. By

Figure 5.2: A `CheckBox` changing its text color when selected

```
<CheckBox>
    <CheckBox.Style>
        <Style TargetType="CheckBox">
            <Setter Property="Foreground" Value="Red" />

            <Style.Triggers>
                <Trigger Property="IsChecked" Value="True">
                    <Setter Property="Foreground" Value="Green" />
                </Trigger>
            <Style.Triggers>
        </Style>
    <CheckBox.Style>
</CheckBox>
```

Listing 5.3: Setting a property using a trigger in a style

default, the `Foreground` color of the `CheckBox` is set to `Red` using a `Setter`. A trigger is used to react to changes of the `IsChecked` property. Once it becomes `True` the trigger uses another setter to change the `Foreground` color to `Green`.

5.2.2 Resources

The style defined above was used by directly setting the `Style` property of the `CheckBox` control. Usually, styles are defined in separate resource files. That means, they are considered to be resources of the application. Each element of the visual tree has a resource collection. Any type of object can be put into this collection and is accessed by a key. For example, the style object could have been put into the `Resources` collection of the `Window` as shown in Listing 5.4.

```
<Window.Resources>
    <Style x:Key="MyStyle" TargetType="CheckBox">
    ...
    </Style>
</Window.Resources>
...
```

Listing 5.4: Defining a style as resource

It could then be used by any child object in the visual tree by accessing it through the `StaticResource` syntax seen in Listing 5.5.

```
<CheckBox Style="{StaticResource MyStyle}">
```

Listing 5.5: Applying a style resource

This works because resource collections inherit the resources of parent UI elements. If the resource collection of `CheckBox` is empty, `StaticResource` looks at the resource collection of its parent. The same is true for the other direction as well. If a style is defined for a specific `TargetType`, but without any key, it is applied to all children objects of that type. This is a very powerful mechanism to enforce a specific look and feel of specific objects in a subtree of the visual tree.

5.2.3 User Controls and Custom Controls

One way of reducing the complexity of a composition is by grouping controls and putting them into a reusable container. Thus defining a control of controls. This is called a *user control*. It is edited the same way like a window, but instead of defining a subclass of `Window`, it is a subclass of `UserControl` and can be used like any other control.

Strictly speaking, WPF always creates new controls by subclassing. `UserControl` is a special class, which allows editing the visual tree directly through XAML. But there are many more ways for creating custom controls. Possible base classes are for example:

FrameworkElement The base class of all user interface controls defining support for data-binding, styles, animation, layouts. Custom drawing is done by overriding the `OnRender` method of this class.

Control Adds various common properties like font, foreground, background to the UI element. Also defines a `Template` property of type `ControlTemplate`, which can be set by using a global style. This `ControlTemplate` can contain an arbitrary visual tree. Creating controls this ways is the standard way of creating new, fully customizable controls. Their behaviour is defined in a class file, while the control template defines their exchangeable look & feel.

Of course any existing control can also be the base class of a new control. This is useful if many features are already implemented in a standard control but should be extended.

5.3 Data Binding

Data binding is a mechanism built deeply into the framework. It is a mechanism allowing objects to synchronize their property values. If a property is updated, the property of another object *bound* to that property updates as well. These connections can be one way or two way. They can be created programmatically, but they are mostly used in XAML to connect the UI with the business logic. Their usage is briefly illustrated in the following example seen in Figure 5.3.

The window shown in this figure contains a `Slider` and a `TextBlock` control. A `Slider` is used to set a value by moving it in a certain range. This value should now be displayed in the `TextBlock` control. Listing 5.6 shows how this is done by defining a `Binding` to the `Slider` property `Value`.

This is a binding between two controls. The same mechanism can used to connect to an arbitrary data source. If a binding only defines a path to a property, it uses the element's `DataContext` as property source. Each `UIElement` has a `DataContext`, which is a inherited property propagated recursively to all children of the visual tree. If the data context of a window is set, all of its children have that data context. Children can of course set their own `DataContext`, which then propagates to their children. This mechanism is used in the MVVM pattern which is described next.

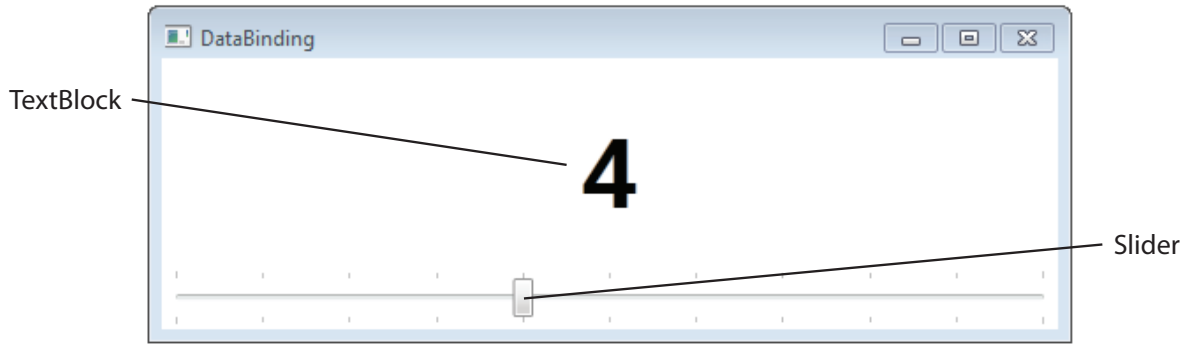


Figure 5.3: Databinding example

```
<Slider x:Name="mySlider" />
<TextBlock Text="{Binding Path=Value, ElementName=mySlider}" />
```

Listing 5.6: Binding a slider's value to a text block

5.4 Model-View-ViewModel

The architecture of the tool is based on the so-called Model-View-ViewModel pattern (MVVM). This phrase was coined in 2005 by John Gossman [4] [14], a WPF architect at Microsoft. In this pattern, an application is split up into three groups of objects.

Model: The model of an application describes its data. All objects which form an abstraction of the problem and do not have anything to do with the user interface can be put into the model. Usually, the same model can be used with different UI toolkits, e.g. WPF or WinForms.

View: The user interface of an application is built of multiple view objects. In WPF these objects are mostly defined using XAML. Views are created by arranging existing controls and customizing them or creating completely new ones. They are populated with data from the *ViewModel* using data binding.

ViewModel: A ViewModel is an abstraction of a view. It contains its state and its behaviour. Java developers would call this a controller and an adapter. It prepares data for display and exposes it through properties. Actions which can be performed on the ViewModel are exported as commands.

In the following, the components of this pattern are explained using a simple example. Figure 5.4 shows a screenshot of the application. This example displays the value of a counter at the top of the window and allows changing this value by clicking on buttons. Depending which button is clicked, -5, -1, +1 or +5 is added to the current value. Moreover, the counter's value is limited to the range -100 to 100.

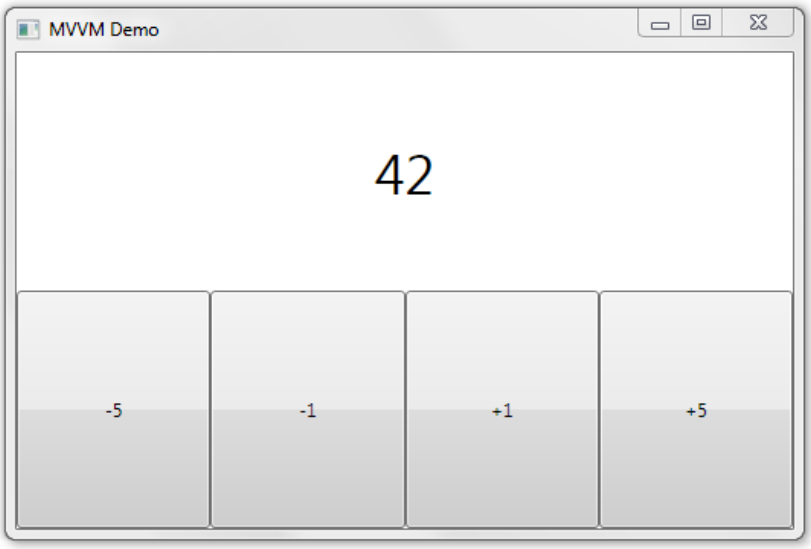


Figure 5.4: Screenshot of the MVVM example application

5.4.1 Model

The model of this example is implemented by the `LimitedCounter` class. The main responsibility of the model classes is to save and validate data. In this example, the data of the application is just a integer value. However, the model must ensures this value stays in the range $[-100, 100]$. Listing 5.7 shows the implementation of the `LimitedCounter` class. It provides the integer property `Value` with getter and setter. In the setter, new values are limited to the valid range.

```
public class LimitedCounter  {
    private int _value;

    public int Value {
        get { return _value; }
        set {
            _value = value;

            if(_value > 100)
                _value = 100;

            if(_value < -100)
                _value = -100;
        }
    }
}
```

Listing 5.7: Implementation of the `LimitedCounter` model class

5.4.2 Defining a ViewModel

The ViewModel of the example application is implemented by the `LimitedCounterViewModel` class. ViewModels expose data from the model to a view. Unlike controllers, ViewModels usually

```
public abstract class BasicViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void RaisePropertyChanged(string propertyName)
    {
        if(PropertyChanged != null){
            PropertyChanged(this,
                            new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Listing 5.8: Implementation of the INotifyPropertyChanged interface

do not know anything about specific view objects. Instead they define properties and commands which are consumed by views through data binding. In the following, these two mechanisms are described.

Properties

There are several ways to create bindable properties which can be read and observed by WPF controls. One of the main building blocks of WPF binding mechanism is the replacement of regular C# properties by dependency properties. These properties enable special features such as bindings, animation and inheritance of values. Every object derived from `DependencyObject` uses these properties to define its external interface. Since `FrameworkElement` is a subclass of `DependencyObject`, every class in WPF inherits this mechanism.

ViewModel classes can be implemented by creating a subclass of `DependencyObject` and using dependency properties. However, in many cases it is sufficient to use regular objects with so called *POCO (plain old CLR object) properties*. WPF's property system allow objects, which are not dependency objects, to participate in data binding. All they have to do is implement the `INotifyPropertyChanged` interface. Listing 5.8 shows a simple implementation, which was put into a abstract class called `BasicViewModel`.

Each property of a ViewModel must raise the `PropertyChanged` event if one of its property values is modified. Listing 5.9 shows how the `LimitedCounterViewModel` class defines a POCO property named `Counter`. It exposes the value of the `LimitedCounter` model and fires `PropertyChanged` events when set. Because of its simplicity, many WPF developers favor this implementation. Dependency properties should only be used to extend existing `DependencyObjects`, such as controls. There is no need to pollute a new class hierarchy with features of `DependencyObject` if they are not necessary.

```
public class LimitedCounterViewModel : BasicViewModel
{
    private LimitedCounter _counter = new LimitedCounter(); // model

    public int Counter
    {
        get { return _counter.Value; }
        set
        {
            if(_counter.Value != value)
            {
                _counter.Value = value;
                RaisePropertyChanged("Counter");
            }
        }
    }
    ...
}
```

Listing 5.9: Definition of a POCO property in `LimitedCounterViewModel`

Commands

Property changes can trigger background operations and have side effects. Sometimes these actions should be directly accessible through UI controls such as menu items or buttons. Such controls can be bound to command objects, which implement the `ICommand` interface shown in Listing 5.10.

```
interface ICommand
{
    bool CanExecuted(Object parameter);
    void Execute(Object parameter);
}
```

Listing 5.10: ICommand interface

Every command reports if it is executable by returning TRUE through its `CanExecute` method. Controls binding to this command are disabled if this method returns FALSE. The `Execute` method performs the command action. Both methods have an optional command parameter. This parameter can be used to supply additional data to the command.

Since writing a new class for each command is cumbersome, the `RelayCommand<T>` utility class [14] is used as a generic container of two delegate methods. `T` is the type of the command parameter. Listing 5.11 shows how the implementation of the `IncrCommand` in `LimitedCounterViewModel`. A new instance of `RelayCommand<string>` is created, passing a lambda expressions for the `Execute` and `CanExecute` methods as parameters. The command parameter of type string is used to encode an integer number. If invoked, this command increments the `Counter` property using the integer value of the command parameter. Since the `CanExecute` delegate always returns true, this command is always enabled.

```
public class LmitedCounterViewModel : BasicViewModel
{
    ...
    private RelayCommand<string> _incrCommand;

    public ICommand IncrCommand
    {
        get
        {
            if (_incrCommand == null)
            {
                _incrCommand = new RelayCommand<string>(
                    (param) =>
                    {
                        // Execute
                        int val = int.Parse(param);
                        Counter += val;
                    },
                    (param) => true // CanExecute
                );
            }
            return _incrCommand;
        }
    }
}
```

Listing 5.11: Definition of a command in `LimitedCounterViewModel` by using `RelayCommand`

5.4.3 Using a ViewModel in a View

The final step now is to connect the ViewModel with a view. ViewModels are usually made available to controls by setting them as the `DataContext` of the visual tree. Every binding using a relative path then uses this context as binding source. Listing 5.12 shows how a ViewModel is defined as a resource and then set as `DataContext` of the `Grid` container. The `TextBlock` control binds to the `Counter` property of the `DataContext` and is placed in the middle of the first row. In the second row of the grid, four buttons are placed. Each button is bound to the `IncrCommand` and specifies a different `CommandParameter`.

Clicking on one of the buttons invokes the bound command. The effect is that the `Counter` property is incremented and fires a `PropertyChanged` event. This event makes the `TextBlock` binding reevaluate and display the new value.

```
<Window x:Class="MVVMDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:MVVMDemo="clr-namespace:MVVMDemo"
        Title="MVVM Demo" Height="350" Width="525">

    <Window.Resources>
        <MVVMDemo:LimitedCounterViewModel x:Key="viewModel" />
    </Window.Resources>

    <Grid DataContext="{Binding Source={StaticResource viewModel}}">
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>

        <TextBlock Grid.Row="0" Grid.ColumnSpan="4" FontSize="36"
            Text="{Binding Counter}"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"/>
        <Button Grid.Row="1" Grid.Column="0"
            Command="{Binding IncrCommand}"
            CommandParameter="-5">-5</Button>
        <Button Grid.Row="1" Grid.Column="1"
            Command="{Binding IncrCommand}"
            CommandParameter="-1">-1</Button>
        <Button Grid.Row="1" Grid.Column="2"
            Command="{Binding IncrCommand}"
            CommandParameter="1">+1</Button>
        <Button Grid.Row="1" Grid.Column="3"
            Command="{Binding IncrCommand}"
            CommandParameter="5">+5</Button>
    </Grid>
</Window>
```

Listing 5.12: Defining a ViewModel as resource and using it

Chapter 6

Views

The following sections outline the implementation of the most important views of the trace visualization tool.

Section 6.1 describes how the task scheduling view uses multiple layers to display the scheduling of tasks by drawing chunks along a timeline, and how it interacts with other views.

Section 6.2 explains why the execution path view requires a different time measure and shows how it is mapped to real time. It further describes how the colors of execution paths are chosen. It concludes with the description of how multiple `TreeViews` are connected to the same ViewModel and how they display the static call tree and execution cycles.

Section 6.3 introduces a utility view named `RegionView`, which is used by many other views, for example the execution phase view, to highlight regions along a timeline. Furthermore, it describes how a custom layout panel must be implemented in order to fit into the WPF layout system.

Section 6.4 describes the functionality of the execution transcript view. It shows how a custom scrollable and virtualized panel is created which is used to display large collections of objects.

6.1 Task Scheduling View

The primary focus of the task scheduling view is to display active regions of task programs and their interruptions along a timeline. Active regions of code contains trace nodes which are executed between the *BeginCycle* and *EndCycle* nodes of a task. As described in Chapter 4, these active regions are split up into chunks, which are loaded and made available by a `ChunkReader` (compare Section 4.1.4).

6.1.1 Displaying chunks

Figure 6.1 shows a screenshot of the view and names the visible elements. On the left side of the view, a list of task names is shown. Each task is displayed as a new row. In this row, the right side of the view shows the chunks of that task along a timeline. Some chunks might be complete execution cycles, while others are parts of an interrupted execution cycle. To illustrate that an execution cycle has been interrupted, the space between interrupted chunks is colored in gray.

Note that in this view chunks are not displayed on a linear timeline. As argued in Section 4.2.2, the duration of chunks can be very small compared to inactive time regions. Active regions are therefore stretched in time, while inactive regions are compressed.

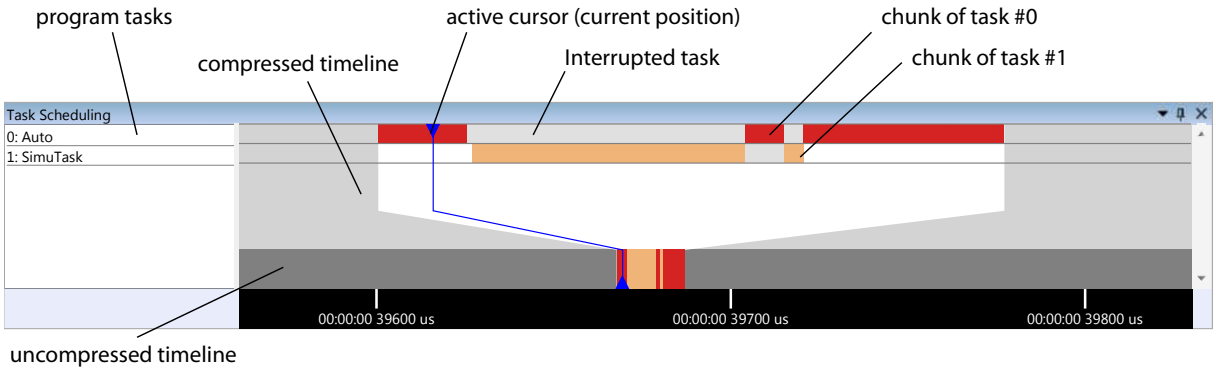


Figure 6.1: Screenshot of the Task Scheduling View

By taking the visible real time range and transforming it into a compressed and uncompressed time range, a list of visible chunks can be obtained from the `ChunkReader`. Each chunk must then be drawn using its start and end node index. These indices can be easily converted to compressed and uncompressed time, using the linearly interpolated time introduced in Section 4.2. What remains is transforming the time into graphical (x, y)-coordinates. The y-coordinate is set by the task index, the x-coordinates is calculated from the time t using the following formula

$$p_x = (t - t_{start}) * \frac{ViewWidth}{t_{end} - t_{start}}$$

with t_{start} and t_{end} marking the visible time range and $ViewWidth$ being the total width of the view in screen units (pixels).

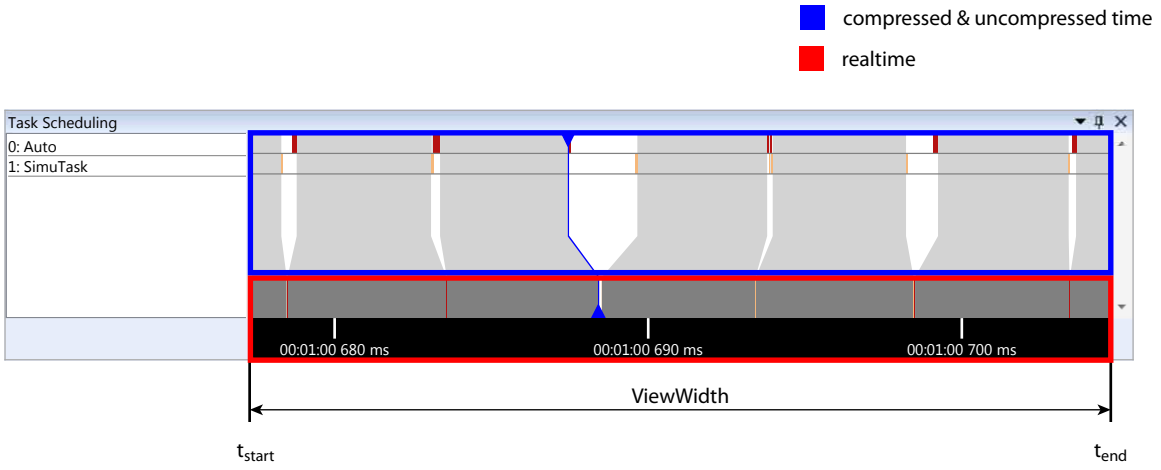


Figure 6.2: Separation between realtime and compressed & uncompressed time

As described above and as can also be seen in Figure 6.2, the main body of the view is made up of the compressed timeline of chunks. To emphasize that time has been stretched, a linear, unaltered timeline is placed underneath. Active regions are additionally colored with a white background, compressed time ranges remain gray. This creates the effect that parts of the linear timeline seem to be expanded above.

A chunk's color is defined by the execution path that was taken in its cycle. This also helps identifying chunks of the same cycle, since they have the same color. See Section 6.2.3 for the definition of clusters and how their colors are generated.

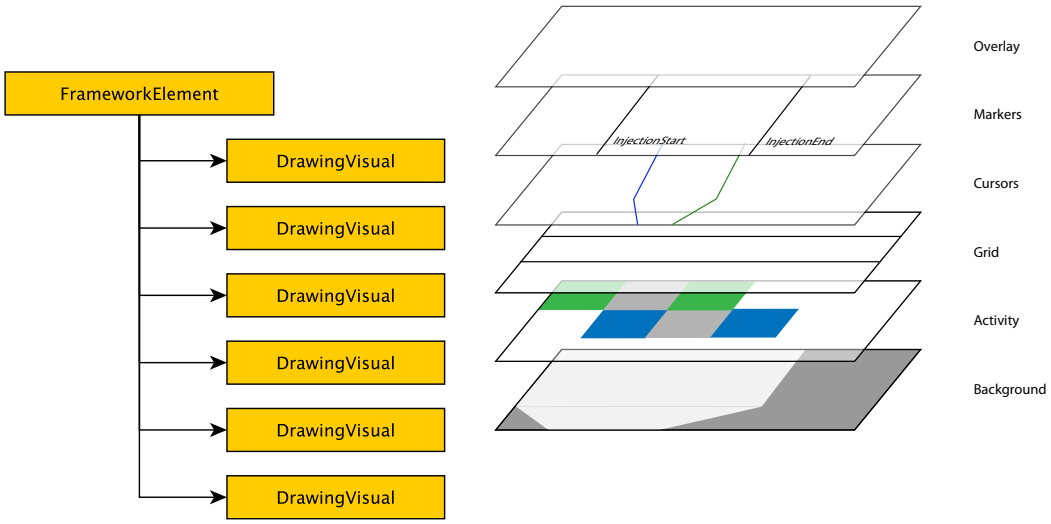
The task scheduling view displays a certain range of time, which is defined in real time. The resolution of these times is a few nanoseconds. Since a trace can be as long as a few minutes, any valid time range can be between nanoseconds (10^{-9} s) up to a few minutes. That means the ratio between the smallest time range and the entire trace is in the order of 10 to the power of 10. However, it is reasonable to ignore any time ranges above a few milliseconds, since at that point chunks would become smaller than single pixels. Thus, the view has a built-in maximum time range limit. Any time range greater than 100 ms does not display any data, but instead shows the text "too much data to display".

6.1.2 Layered rendering

The task scheduling view is directly derived from `FrameworkElement`. This means it is part of the dependency property system and supports styling. However it does not define a control template. The class is specialized in order to override its `OnRender` method and implement its own drawing logic at a very low level. This is also known as *Visual Layer Programming*[18]. The `OnRender` method has a `DrawingContext` as parameter. Using this object, low level drawing functions can be called which directly manipulate the visual layer. This is the fastest way to draw in WPF's retained graphics system. Drawing this way is very efficient, but it requires

implementing advanced drawing techniques manually.

However, the task scheduling view does not override the `OnRender` method. Instead it acts as `DrawingVisual` host container [17] and uses multiple layers for rendering. Each layer is represented by a `DrawingVisual` object. Figure 6.3 shows the visual tree of the task scheduling view and illustrates how the different layers are stacked on top of each other.



(a) Visual tree (b) Layers

Figure 6.3: Layered structure of the task scheduling view

The final version of the view uses six different layers:

Background This layer highlights active regions by drawing a white background while inactive regions remain gray.

Activity Contains the chunks drawn in compressed and uncompressed time scale.

Grid Draws a grid above the chunks, separating different tasks. This layer usually only redraws during resizing.

Cursors Various positions in the trace are marked using cursors which are colored lines with reverse arrow heads at both ends. These can change independently from the visible area of the view.

Markers Fixed positions in the trace which are given a name. A vertical line is drawn at the position and the name written next to it.

Overlay Visual elements which are only visible for a short period of time and caused by user interaction are drawn in the overlay layer.

If one layer changes, it does not require others to be redrawn. For example, if a cursor moves along the timeline, there is no reason why the background should be redrawn. Deciding which layers should be updated requires copies of unchanged layers in memory. This is achieved by using a `DrawingVisual` object for each layer. These objects allow opening a separate `DrawingContext` using the `RenderOpen` method and drawing on them. Once an update is finished, the `DrawingContext` is closed. The `DrawingVisuals` content does not change until its context is reopened. Clearing it is done by simply opening its `DrawingContext` and closing it without drawing anything.

So layers can be updated independently using their `DrawingVisual` object. In order to display these objects, a `DrawingVisual` host container must tell the WPF drawing system about its visual tree. By default, a `FrameworkElement` has an empty visual tree. To change this, all `DrawingVisuals` are put into a `VisualCollection` object. The item added last is the one on top. A `VisualCollection` maintains the connections of the logical and visual tree [16]. This collection is then used to override the protected methods `VisualChildrenCount` and `GetVisualChild`. There are various implementations of these two methods. Listing 6.1 shows a variant which preserves existing visuals added to the visual tree by calling the base class implementation of these methods.

```
private VisualCollection Layers = ...;

protected override int VisualChildrenCount
{
    get { return Layers.Count + base.VisualChildrenCount; }
}

protected override Visual GetVisualChild(int index)
{
    if (index < 0 || index >= VisualChildrenCount)
    {
        throw new ArgumentOutOfRangeException();
    }

    if (index < base.VisualChildrenCount)
    {
        return base.GetVisualChild(index);
    }
    return Layers[index - base.VisualChildrenCount];
}
```

Listing 6.1: Implementation required by a `DrawingVisual` host container

Once the modified visual tree is exposed this way, the default implementation of the `OnRender` method renders it. If a layer changes internally once its `DrawingContext` is closed, it is updated automatically.

6.1.3 Navigation

Navigation inside the scheduling view is done using the mouse and keyboard. The current implementation of this view supports three different operations:

- panning by dragging the view with the left mouse button
- zooming in and out of a specific position using the mouse wheel while pointing at the position
- zooming into a specific region by selecting it using shift + left drag

Since navigation is a common task shared by various views, this was implemented using an `Adorner`. This is a special visual element, which draws itself above another control. It can catch UI events, like mouse events, of the adorned element. This is used to draw highlights and fire navigation events like `ZoomToPoint`, `ZoomToRegion` and `Pan`. Each view which uses this adorner reacts to these events and redraws itself.

6.1.4 Interaction with other views

Since other views are better for finding the location of a problem, mechanisms to coordinate the different views were needed. The scheduling view allows two different connection mechanisms.

One of them is displaying multiple cursors. Each cursor is positioned using a `TracePosition` object (see Section 6.2.2) and may have a color. Some views expose their current position as a cursor, which can then be displayed in other views. One of these cursors can be set to be the scheduling view's active cursor. Double clicking on the view moves the active cursor to the clicked position. Another view can then show more detail about this location.

The second mechanism used to share data between views is connecting them. Although views may use different definitions of time (real time, simulation time, node ranges, cycle ranges), they can be connected by sharing their visible frame which is a pair of `TracePosition` objects. Each view updates the shared frame by using its own definition of time. `TracePosition` translates these definitions and notifies listeners about changes.

6.2 Execution Path View

The Execution Path View is a high-level view of the program behavior. As described earlier, an execution path is the path taken through the static source code of a task program. When tasks are executed in a cyclic way, every cycle goes through a specific path. Each of these paths has been given a number, uniquely identifying it. Thus, for each task, there is a list of path numbers describing which path has been taken during each cycle (compare Section 3.1.1).

These cycles are visualized along a timeline. Each task is displayed as a row. In this row time progresses along the horizontal axis and shows the paths of the execution cycles of the task. Instead of using the exact times of each cycle, defined by its *BeginCycle* and *EndCycle* nodes, a task cycle is represented by a rectangle which stretches over its entire cycle time.

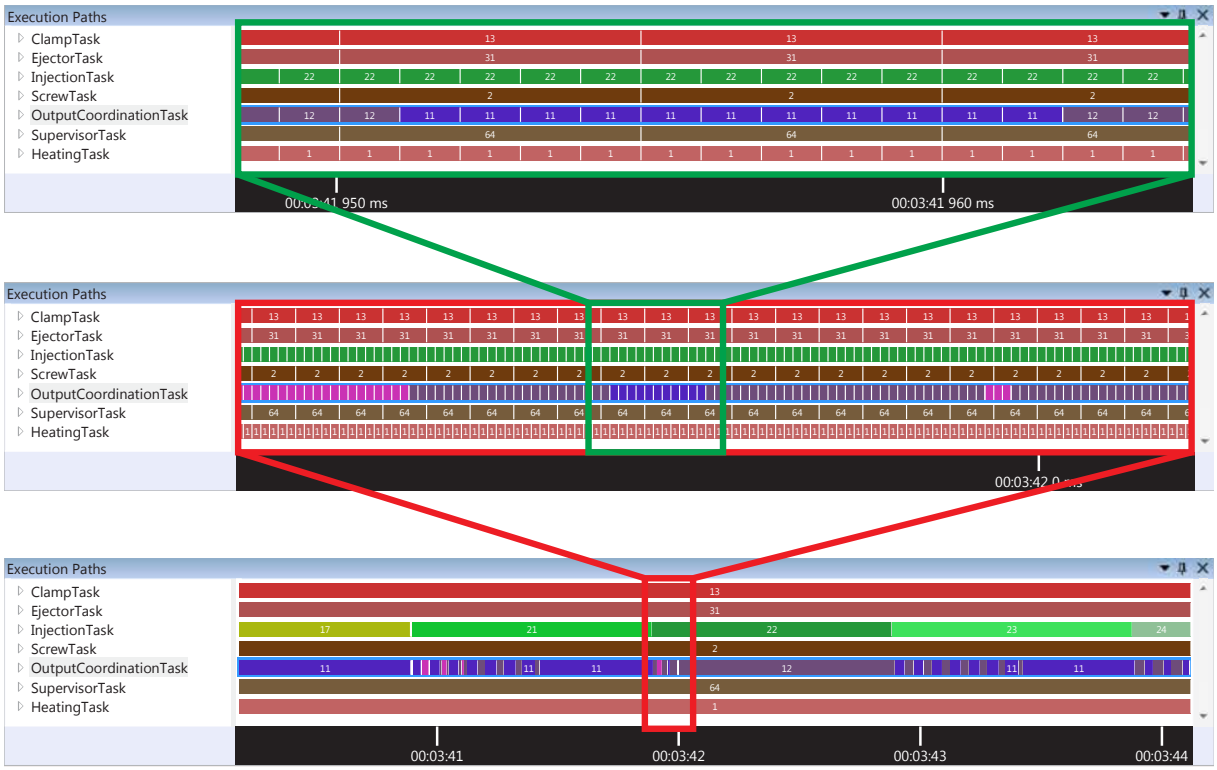


Figure 6.4: Multiple zoom levels of the execution path view (top: high, bottom: low)

Figure 6.4 shows the resulting view at multiple zoom levels. This example was taken from the KePlast case study, which is described in Chapter 7. At the highest zoom level (top), the view shows individual cycles. Each cycle is colored in a special way, delimited by a white border and displaying its execution path id in its center. Note that Figure 6.4 contains execution cycles of 1 ms and 5 ms tasks. A task with a 5 ms cycle time has rectangles spanning 5 ms. The rectangles of a task with a 1 ms cycle time only span 1/5 of that size, and therefore 5 cycles of the 1 ms task occur during one cycle of the 5 ms task.

Zooming out decreases the size of individual execution cycles while showing a larger time span. At some point there is not enough space to display individual path ids. However, individual

cycles are still visibly separated by their white border. Reducing the zoom level even further shows that the view no longer displays individual cycles. Instead it groups cycles with the same path id, shows a rectangle for that group and displays the path id in the group's center.

Looking at the view using a low zoom level, one will discover that the progression of execution cycles helps comprehend the reactive behavior of the application itself. In bottom part of Figure 6.4 a timespan of about four seconds is shown. During this time the *ClampTask*, *EjectorTask*, *ScrewTask*, *SupervisorTask* and *HeatingTask* are all executing a single execution path. This means they are either waiting for something or are in a long lasting operation. At the same time the *InjectionTask* seems to be moving through a sequence of actions, each lasting approximately one second. The task doing most of the work seems to be the *OutputCoordinationTask* which shows a high frequency of execution path changes.

To further analyse what tasks are doing, the execution path view allows to expand each task. Then, it displays the static call tree of the task, recursively listing all called POUs as a tree. Figure 6.5 shows the expanded call tree of the controller task *Auto* in the portal robot example (see Section 2.3).

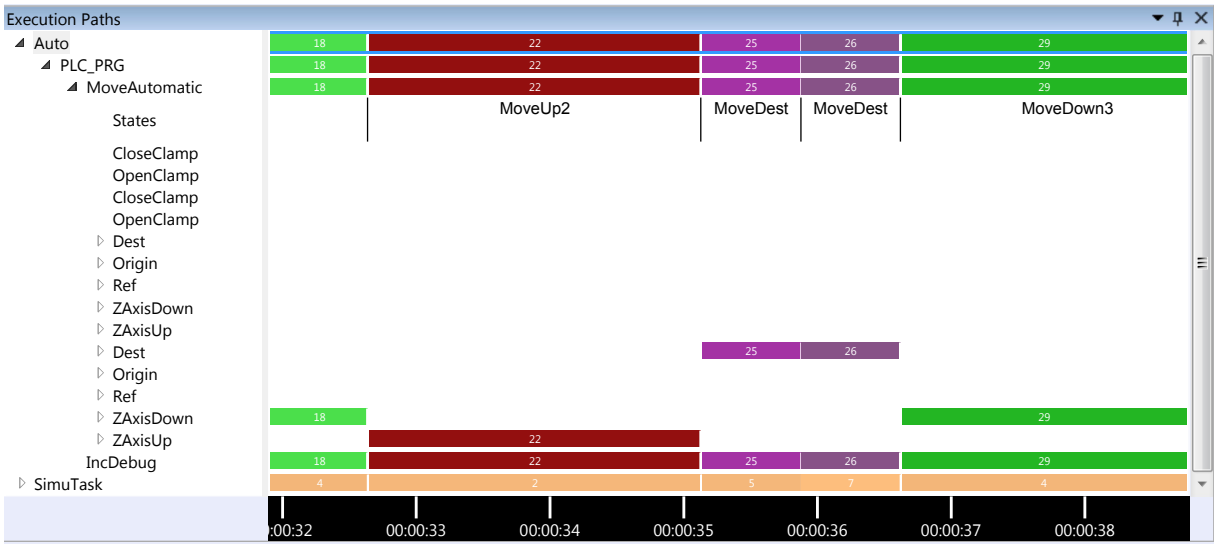


Figure 6.5: Expanding the call tree of a task

Each called POU is displayed as a row and shows execution cycles which contained the call. It shows that the task program `PLC_PRG` calls the function block `MoveAutomatic`. This POU is implemented by an SFC. Since each action is represented by an action POU, the static call tree of the `MoveAutomatic` POU contains all calls to these actions. The right side view shows when a call was executed. For example, the action `ZAxisUp` is executed in execution path 22 for about two seconds, before `Dest` becomes active in execution path 25. Besides listing all POU calls, the tree also contains a special row which shows the progression of SFC states. While `ZAxisUp` is active, the SFC is in the state *MoveUp2* and the action `Dest` is executed in the *MoveDest* state.

Note that zooming into the region between these two states reveals single execution cycles which form the transitions between the states. This is shown in Figure 6.6. The red text

underneath the state name *MoveUp2* is the name of the transition which is active during that execution cycle.

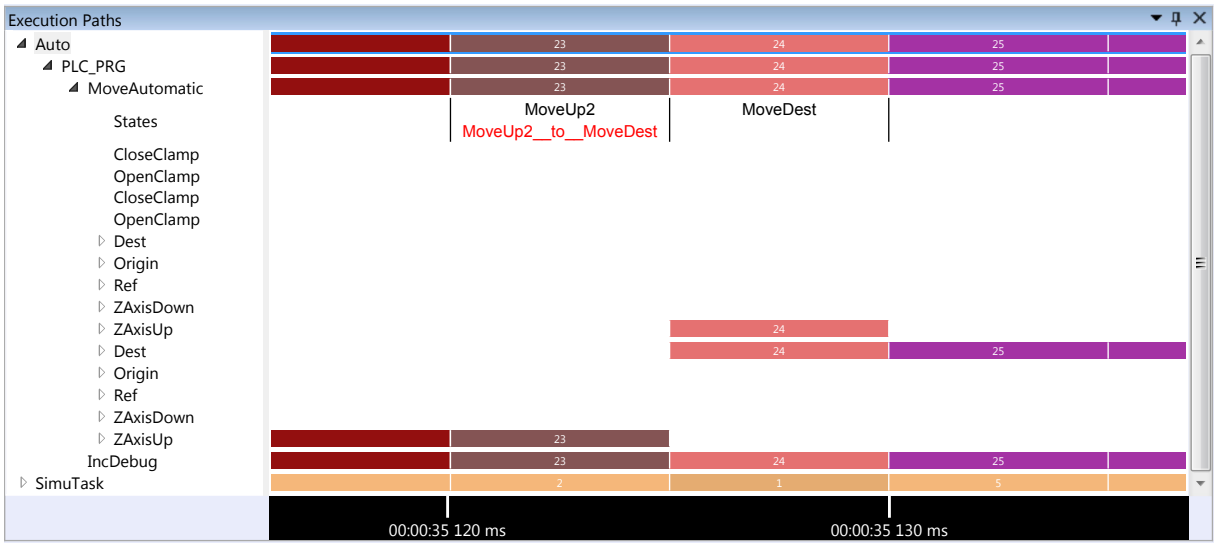


Figure 6.6: Transitions of a SFC

6.2.1 Simulation Time

The execution path view does not draw execution cycles along a regular timeline. This would require to determine each cycle's *BeginCycle* and *EndCycle* timestamp. Instead each execution cycle is displayed as multiple of its cycle time ΔT. This is shown in Figure 6.7. Each task cycle spans from $\tau_i = i \cdot \Delta T$ to $\tau_{i+1} = (i+1)\Delta T$. In the following, the time measure τ is called *simulation time*.

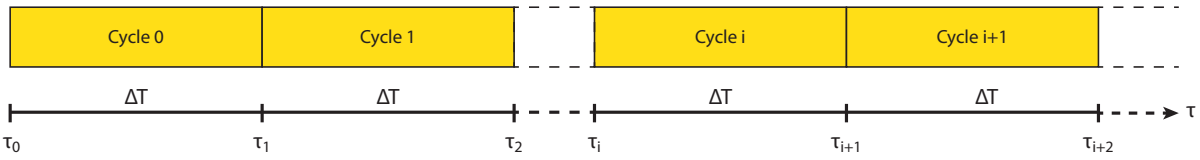


Figure 6.7: Graphical illustration of simulation time

There is a problem with the definition of simulation time when displaying multiple tasks. Figure 6.8 illustrates a simple example of two task cycles executed in real and simulation time. In this example both tasks have a cycle time of 1 ms and their execution cycles do not interrupt each other.

In real time, both task 0 and task 1 complete their first execution cycle within the first millisecond. Their execution time is constrained by the real-time operating system. Each task must execute within its cycle time and run to completion before the next cycle starts. Usually the execution time is much shorter than the cycle time. Additionally, the execution order of tasks with the same cycle time is not fixed and may vary.

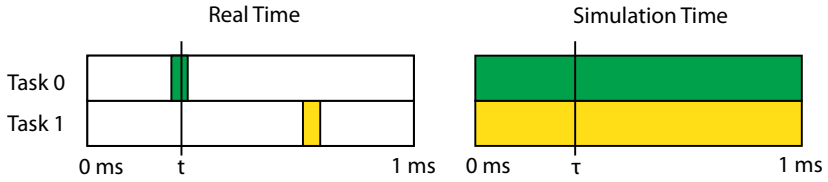


Figure 6.8: Real Time vs. Simulation Time

In simulation time, the first execution cycle of each task starts at τ_0 and lasts until τ_1. Because both tasks have the same cycle time ΔT, the execution cycles of both tasks span from 0 ms to 1 ms. This picture pretends that tasks are truly running in parallel and that each task is active during the entire cycle time.

The problem now is that there is no clear definition of how a real time t is converted to a simulation time τ. The same applies for the inverse. However, these transformations are essential if:

- the Execution Path View should be positioned relative to real time
- a cursor defined as node index or real time must be drawn in this view
- a user clicks on a specific position and wants to look at it in detail

What is therefore necessary is a mapping between the actual cycles, executed in real time, and the idealistic cycles created by using simulation time. Figure 6.9 illustrates how this mapping is done for the simple example used above. Since both 1 ms task cycles run independently, each of them can be seen as one chunk. Both chunks would map to same the 1 ms time range in simulation time. To avoid this collision, the simulation time range is partitioned evenly between the two chunks. This means, the first half of the 1 ms simulation time range belongs to the first chunk, while the second half of the simulation time range belongs to the second chunk.

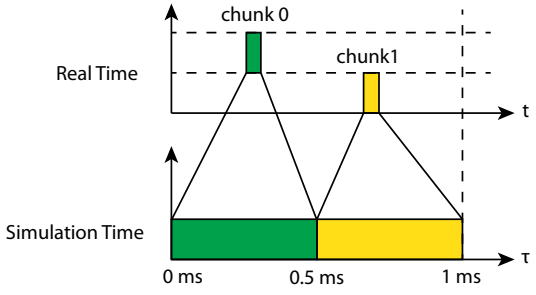


Figure 6.9: Mapping real time to simulation time in the simple example

Conversion of real time to simulation time

Using this mapping a real time t can now be converted to a simulation time τ. This process is illustrated in Figure 6.10. It can be separated into four steps:

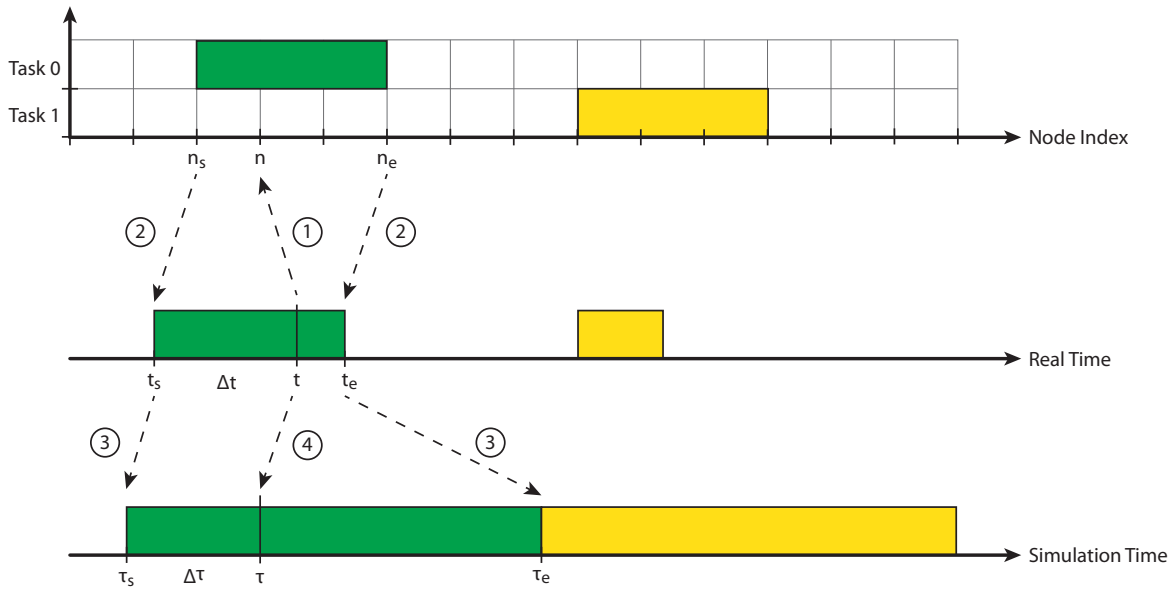


Figure 6.10: Calculating simulation time τ from real time t

1. Calculating the simulation time τ of a given real time t first requires to determine which chunk contains the timestamp t. Since all chunks define a node range $[n_s, n_e]$, looking for a specific chunk means looking for a chunk which contains a specific node index. It is therefore necessary to convert the timestamp t to a node index n. This is done using the function `toNodeIndex` which calculates the node index of a timestamp through inverse linear time interpolation.

$$n = \text{toNodeIndex}(t)$$

This node index n is then used to determine the chunk index c using the `toChunkIndex` function. Note that if n is outside of any chunk's node range, the last chunk which conforms to $n_e < n$ is chosen. In the above example, n is in the first chunk.

$$c = \text{toChunkIndex}(n)$$

2. The next step is to convert the determined chunk's node range $[n_s, n_e]$ to a real time range $[t_s, t_e]$ by applying the linear time interpolation implemented in the `toRealTime` function.

$$t_s = \text{toRealTime}(n_s)$$
$$t_e = \text{toRealTime}(n_e)$$

This allows calculating the time offset Δt between t and t_s:

$$\Delta t = t - t_s$$

Normalizing this offset to the interval $[0, 1]$ leads to definition of the *chunk offset* Δc

$$\Delta c = \frac{t - t_s}{t_e - t_s} = \frac{\tau - \tau_s}{\tau_e - \tau_s} \qquad \Delta c \in [0, 1]$$

The chunk offset describes the relative offset inside a chunk. It is independent of any time measure as long as all components use the same time measure during its calculation.

3. The third step is to use the mapping between chunks and simulation time to determine the chunk's simulation time range $[\tau_s, \tau_e]$. Using the mapping shown in Figure 6.9 this leads to $\tau_s = 0$ ms and $\tau_e = 0.5$ ms in the example.

4. Finally, the chunk offset is used to calculate the relative simulation time offset $\Delta\tau$

$$\Delta\tau = \Delta c\,(\tau_e - \tau_s)$$

Using this offset, the simulation time τ can be calculated.

$$\tau = \tau_s + \Delta\tau$$

By that approach calculating a simulation time τ is reduced to finding the correct chunk and determining the relative chunk offset. Combining the chunk offset Δc with the chunk index c is called a *chunk position* c_p

$$c_p = c + \Delta c$$

Therefore, if the chunk position is known, the simulation time can be calculated. It can be easily verified that a simulation time τ can be transformed into a chunk position c_p as well. Once this has been done, any other time measure can be calculated.

Time mapping with multiple cycle times

The example only showed the most simple case of task execution. However, there are much more complicated variations which must be dealt with by the conversion algorithm. Figure 6.11 shows the time mapping of the first five milliseconds of an application running two 5 ms tasks and two 1 ms tasks. It also illustrates the final output which is generated by the execution path view.

As in the earlier example, the chunks of the execution cycles partition a certain time span. This time span is determined by the smallest cycle time. In this case, it is 1 ms. During the first real time millisecond, only two chunks are executed. Therefore, the first millisecond of the simulation time is evenly divided between the two chunks. In the next real time millisecond three chunks are executed. This means that the second simulation time millisecond is evenly divided between these three chunks. Note that one of the chunks comes from a 5 ms task. The

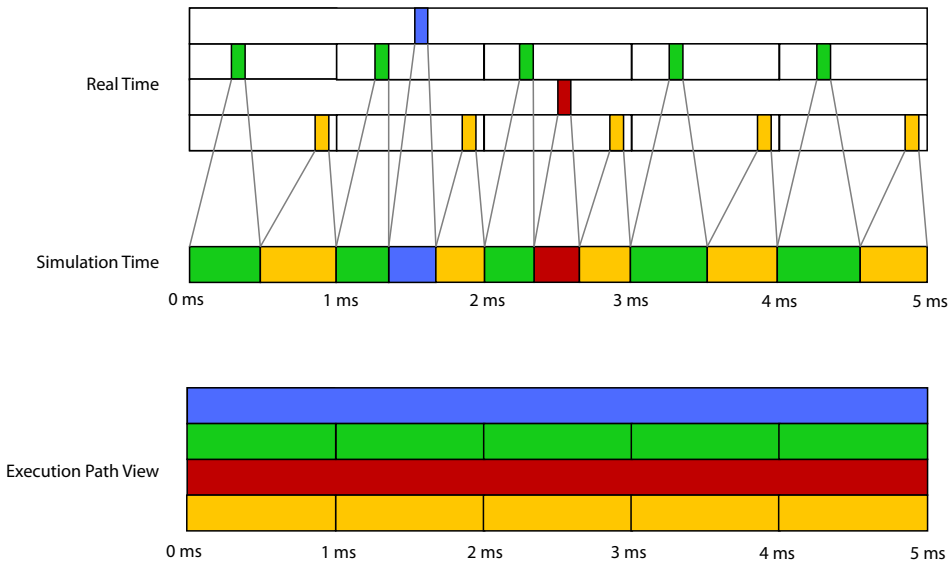


Figure 6.11: Mapping real time to simulation time

other 5 ms task is executed in the third millisecond. Thus, the location where a 5 ms task is executed is not fixed. The time mapping ensures that the execution order is maintained during conversion.

Time mapping with task interruptions

So far only time mappings of uninterrupted execution cycles have been discussed. Recall that task execution is often interleaved. This is the reason why chunks have been used to explain the time mapping and time conversion. In the interleaved case, the basic algorithm is the same as in the simple cases. Chunks partition the smallest time unit and the chunk offset is used to calculate relative offsets. The main challenge is to reduce the amount of chunks being traversed while looking for the correct chunk and determining the number of partitions per unit.

Final note on conversion performance

The overall cost of mapping the different time measures can be quite high. In the worst case data has to be loaded from the hard drive while using `ChunkReader`. Since these steps are only necessary when passing messages to other views or vice versa, they can easily be ignored. Zooming and panning of simulation time based views is done completely without conversions.

6.2.2 TracePosition

The previous section has described some of the difficulties when defining a specific time measure and trying to link it with other views. These types of problems have emerged multiple times during the implementation of various views. To unify the conversion process a new class has

been introduced to represent a position in the trace: the `TracePosition` class. Figure 6.12 shows the public interface of this utility class.

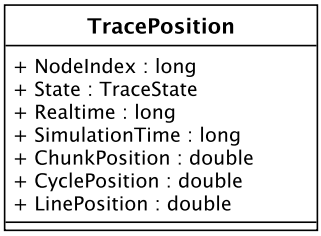


Figure 6.12: UML diagram of the `TracePosition` class

It contains readable and writable properties of all used position definitions. Since it implements the `INotifyPropertyChanged` interface, it reports any value changes by firing the `PropertyChanged` event. Changing one of the properties invalidates all others. The next read operation of a property updates the value through conversion. This conversion result is cached until the next write operation operation occurs.

6.2.3 Coloring of execution cycles

As can be seen in Figure 6.4, colors are used to distinguish the different execution cycles. These colors are influenced by several factors. First, each task has a unique set of colors for its execution paths. The used colors should also emphasize which execution paths are similar. In order to find such a color mapping, the execution paths of each task are grouped into *clusters*. A cluster is a group of execution paths which are similar. See [20] on how clusters are formed.

These clusters are mapped to the HSV color space by assigning each cluster to a specific hue value. This is done by evenly splitting the 360° of hue between all clusters of a task. Saturation and value of an execution path color are generated on demand using a random number generator. Once a color has been generated, it is preserved for later access.

Figure 6.13 shows how the 11 execution paths of a task are distributed in a hue-saturation diagram. Each execution path color is highlighted by a small circle containing its execution path id. Hard contrasts are used to distinguish completely different paths. These steps are repeated for each task. However, to fully utilize the color spectrum, each task's hue is rotated by an offset $\Delta\varphi_i$. Figure 6.14 shows how this is done. In Figure 6.14 (a) the clusters of two tasks are shown. Both have the four clusters and therefore the task clusters are separated by 90°. By rotating the hue of task #1 by $\Delta\varphi_1 = 45°$ the cluster's hue values are separated evenly and with the greatest possible distance.

Figure 6.14 (b) shows a second case with tasks having a different amount of clusters. While the four clusters of task #0 are separated by 90°, the three hue values of task #1 clusters differ by 120°. To avoid hue collisions the values of task #1 are rotated by an offset $\Delta\varphi_1 = 15°$. This

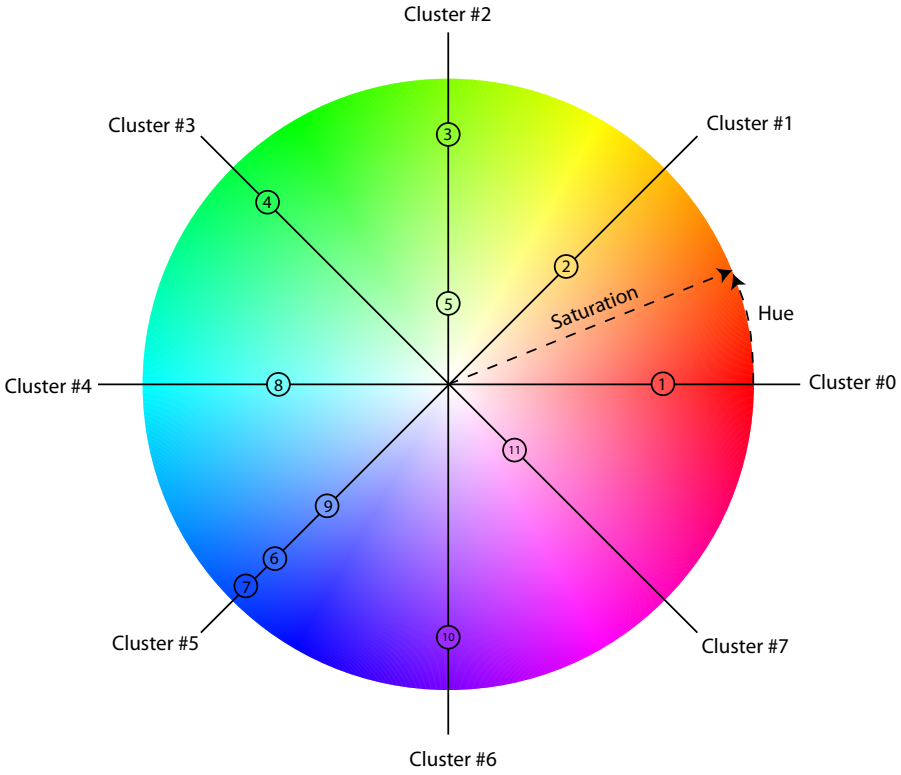
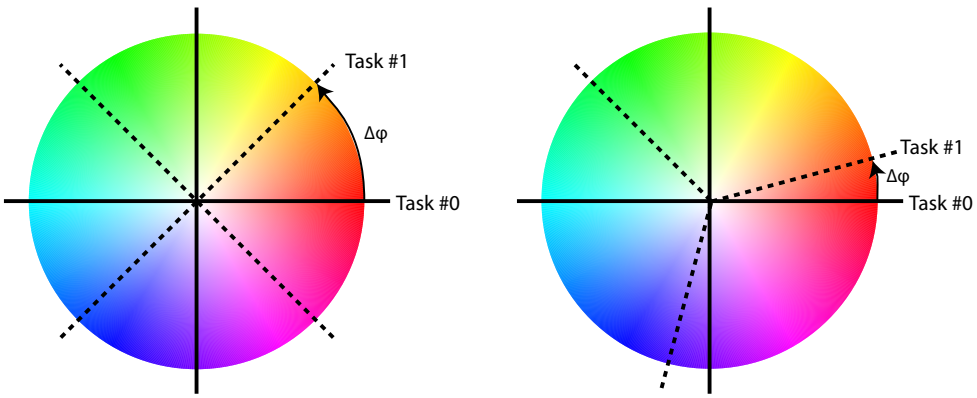


Figure 6.13: Hue-Saturation Diagram, showing how clusters are evenly divided among hue. Paths are visualized as circles with numbers. This is not the complete picture, since there is an additional dimension added by the value of a color.



(a) $C_0 = 4, C_1 = 4$ (b) $C_0 = 4, C_1 = 3$

Figure 6.14: Task clusters are separated by an offset $\Delta\varphi$

offset is calculated by the following heuristic formula:

$$\Delta\varphi_i = \frac{i}{n}\frac{360^\circ}{\operatorname{lcm}(C_0, C_1, \ldots, C_{n-1})}$$

n is the number of tasks in the application. C_i is the number of chunks found in task i.

6.2.4 Implementation in WPF

The implementation of the execution path view is done in a more traditional way than the task scheduling view. Instead of drawing everything on screen manually, this view is built using multiple layers of standard controls. The basic structure can be specified using an XAML user-control as seen in Listing 6.2. It contains two tree views which are positioned in a grid side by side. The tree view on the left side displays a text representation of all tasks and their static call tree. On the right side the execution cycles of tasks and POU calls are drawn. In order to allow scrolling both trees at once, they are contained in a scroll viewer.

Both tree views use the same `ItemsSource` to populate the tree. This property is bound to the `CallTree` property of the view's ViewModel. This tree structure contains the static call tree of all tasks. Each call is represented by a `TreeElement` object. Since there are two supported programming languages at this time, there are two subclasses: `CodeTreeContext` is used to represent calls of POUs implemented in ST and `SFCStateContext` represents calls of POUs implemented in SFC.

By using a tree view to display this data, each task and call is displayed on a separate line. The left tree view only displays names of tasks and POUs. The right shows execution cycles where the corresponding call was executed. The only difference between the two tree views is how each element is presented to the user. This is done by using different styles, which override the default look.

```
<UserControl>
    <ScrollViewer>
        <Grid>
            <TreeView Grid.Column="0"
                Style="{StaticResource PlainTextStyle}"
                ItemsSource="{Binding ViewModel.CallTree}" />

            <TreeView Grid.Column="1"
                Style="{StaticResource CustomDrawingStyle}"
                ItemsSource="{Binding ViewModel.CallTree}" />
        </Grid>
    </ScrollViewer>
</UserControl>
```

Listing 6.2: Simplified version of the `ExecutionPathView` `UserControl`

Changing how a `TreeView` displays its items

`TreeView` is a subclass of `ItemsControl`. This type of container is used to display collections of data, which are data-bound to the `ItemsSource` property. Figure 6.15 illustrates the structure of the visual tree in an `ItemsControl`. For each element of the `ItemsSource` collection an `ItemContainer` is generated and added to the `ItemsPanel`. `ItemsControl` uses `ContentPresenter` as container. This utility control first tries to find an appropriate *data template* which specifies how the element data should be represented in a visual tree. If there is no such template, it uses the `ToString()` method of the object to create a text block. Therefore, there are two components which can be changed in order to modify the appearance of collection elements. The `ItemContainerStyle`, which is applied to all item containers and thus is the same for all elements, and the data templates, which define the look and behavior of specific data objects.

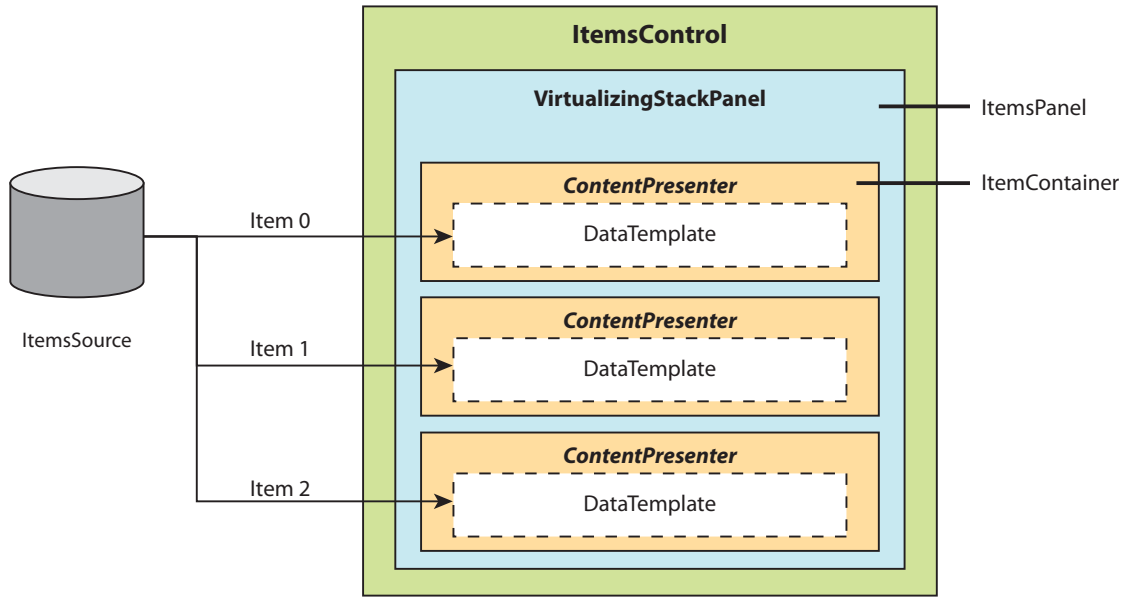


Figure 6.15: Structure of a `ItemsControl`

Figure 6.16 shows the structure of the visual tree in a `TreeView` control. `TreeView` uses `TreeViewItem` as `ItemContainer`. Inside this container a `ContentPresenter` which tries to load an appropriate data template for the data object. The main difference between regular `ItemControl`s and the `TreeView` is that the `TreeView` uses hierarchical data templates. These define the look and feel of a data object, but also allow data binding to another `ItemsSource` which contains the children of the tree item. `TreeView` uses this information to traverse the tree and create item containers.

Listing 6.3 shows the style used by left tree view. It sets a hierarchical data template as default item template which displays the name of each `TreeElement` as text.

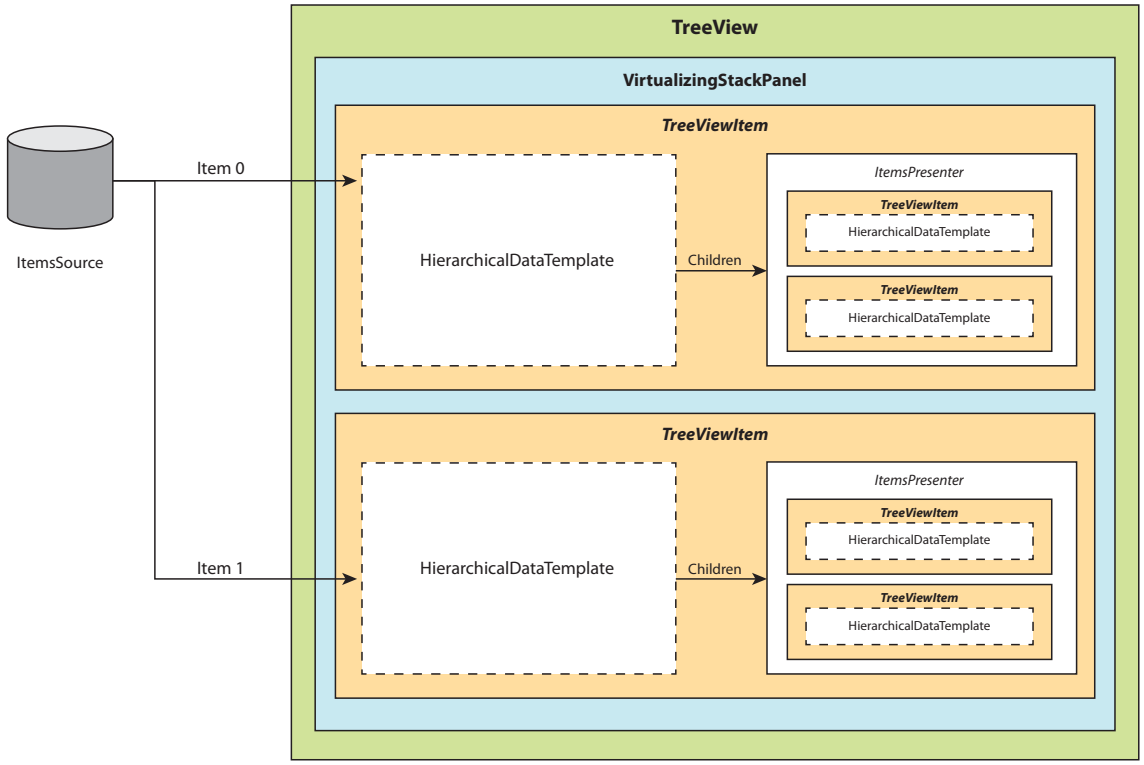


Figure 6.16: Structure of a `TreeView`

```
<Style x:Key="PlainTextStyle" TargetType="TreeView">
    <Style.Setters>
        <Setter Property="ItemTemplate">
            <Setter.Value>
                <HierarchicalDataTemplate
                    DataType="{x:Type TreeElement}"
                    ItemsSource="{Binding Children}">
                    <TextBlock Text="{Binding Name}" />
                </HierarchicalDataTemplate>
            </Setter.Value>
        </Setter>
    </Style.Setters>
</Style>
```

Listing 6.3: Style of the left `TreeView` of the execution path view

The right tree view needs to distinguish between calls to regular POUs and POUs which are implemented in SFCs. SFCs not only display their executed paths, but also show the progression of states and transitions over time. In either case, drawing is performed by a custom control which takes data from the `TreeElement` subclasses and displays it. Regular POUs use the `TaskPathControl` for drawing, SFCs use `SFCStateControl` instead. However the `ItemTemplate` property can only be set to one data template. To support more than one type, the resource system of WPF is used. Each control or style can define a set of resources, which are

visible to all children in the visual tree. If a content presenter looks for a data template of a specific type, it goes up the resource tree and tries to find a match. Listing 6.4 shows the definition of two different data templates in the resource section of the `CustomDrawingTextStyle`.

```
<Style x:Key="CustomDrawingTextStyle" TargetType="TreeView">
    <Style.Resources>
        <HierarchicalDataTemplate DataType="{x:Type CodeTreeContext}"
            ItemsSource="{Binding Children}">
            <Controls:TaskPathControl CodeContext="{Binding}" />
        </HierarchicalDataTemplate>

        <HierarchicalDataTemplate DataType="{x:Type SFCStateContext}"
            ItemsSource="{Binding Children}">
            <Controls:SFCStateControl StateContext="{Binding}" />
        </HierarchicalDataTemplate>
    </Style.Resources>
</Style>
```

Listing 6.4: Style of the right `TreeView` of the execution path view

Synchronizing the expansion of `TreeViewItem`s

Because two separate tree views are used, there are two trees of `TreeViewItem`s which display the element data. The expanded or collapsed state of a tree element is stored in the container objects. This means both trees are independent of each other, since their container tree is different. To create the illusion of one tree, split into two columns, it is necessary to move the `IsExpanded` state into the `TreeElement` objects. Listing 6.5 shows how this is done by setting an item container style. It binds the `IsExpanded` property of the `TreeElement` to the `IsExpanded` property of the container. This propagates any changes made in one tree view to the element's `IsExpanded` property. Through implementation of the `INotifyPropertyChanged` interface by `TreeElement`, these changes are then further propagated to any binding to this property. Therefore, other tree views which bind to the `IsExpanded` property update their container's `IsExpanded` property after notification.

```
<Setter Property="ItemsContainerStyle">
    <Style TargetType="TreeViewItem">
        <Setter Property="IsExpanded"
            Value="{Binding IsExpanded, Mode=TwoWay}" />
    </Style>
</Setter>
```

Listing 6.5: Changing the `ItemsContainerStyle`

6.3 Regions View

Multiple views of the trace visualization tool highlight regions of a program run in a timeline. These regions can be specified in several different ways, such as node index ranges, time ranges and cycle index ranges.

Figure 6.17 shows a screenshot of the execution phases view, displayed above the execution path view. Execution phases are detected by an algorithm described in [20] and define regions spanning multiple execution cycles. In this case the phases of the *Auto* task are shown. Each phase is given a name, consisting of the task name and a number.

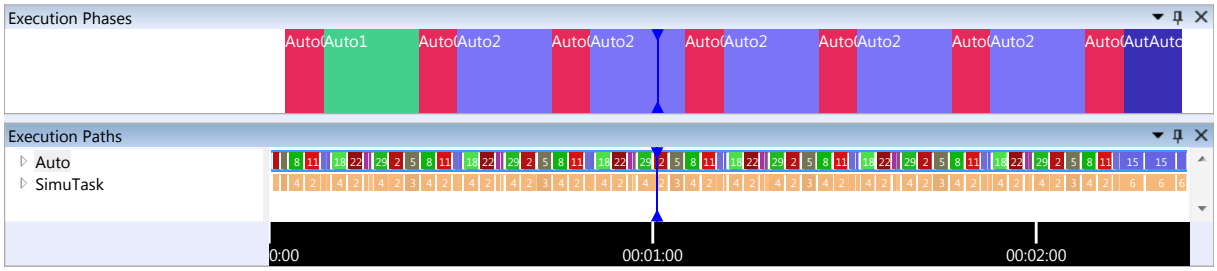


Figure 6.17: Execution Phases displayed using a `RegionsView`

Depending on the requirements of a view, regions should not only be displayed as simple rectangles with an optional text. Sometimes they need to incorporate a context menu or buttons and other controls. What is therefore needed is a control that layouts these regions along a timeline and allows customizing how regions are displayed. This is done by the `RegionsView`, which is a specialization of `ItemsControl`. This control uses a custom layout panel to arrange elements along a timeline. Customization of region display is achieved using data templates. The following sections describes how such a custom layout panel is implemented and how layout information is propagated from region data objects to the layout panel.

6.3.1 Writing a custom layout panel

As mentioned earlier, collections of objects are displayed using specializations of `ItemsControl`. In addition to using data templates to specify a region's interior visual representation, the `RegionsView` overrides the default layout as well. By default, `ItemsControls` use a `StackPanel` to display elements one after another. This panel can be replaced by setting the `ItemsPanel` property of the `ItemsControl`. A custom panel must then do two things:

Determine the size of its children `ItemsControl` instantiates the Panel and uses it as container for all its children. Each data element is put into an `ItemContainer` and added to the panel. During the first layout pass, also known as *measure pass*, the panel's responsibility is to measure all its children and return its own required size. This is done by overriding the `MeasureOverride` method of the panel and calling the `Measure` method of its children. The basic structure of this method can be seen in Listing 6.6.

```
protected override Size MeasureOverride(Size availableSize) {
    foreach(UIElement element in base.InternalChildren)
    {
        Size childSize = element.Measure(...);
    }

    return new Size( ... ); // required size of panel
}
```

Listing 6.6: Implementation of the measure pass

Arrange its children: In a second phase, known as *arrange pass*, the layout system gives the panel its final size. It must then arrange its children in this confined space. The size of each child might be smaller or greater than what children expect. Depending on the implementation, children might be stretched, cropped or hidden. Listing 6.7 shows the basic structure of the arrange pass of a panel, which is implemented by overriding the `ArrangeOverride` method and calling the `Arrange` method of each child.

```
protected override Size ArrangeOverride(Size finalSize)
{
    // put each child on its final location
    foreach(UIElement element in base.InternalChildren)
    {
        element.Arrange(new Rect( ... ) );
    }

    return finalSize; // final size of panel
}
```

Listing 6.7: Implementation of the arrange pass

6.3.2 Forwarding layout information from ViewModel to View

As described in Section 5.4, a ViewModel is an abstract representation of a view, containing its state and behavior. The ViewModel of the `RegionsView` defines a `Frame` property, which contains two `TracePosition` objects. It specifies which range of the trace is visible. When the `Frame` changes, the ViewModel updates its `Regions` collection. This is an observable collection of regions. This collection is data bound to the `ItemSource` property of the `ItemsControl`. Each region uses a data template to display itself and is put inside an item container (`ContentPresenter`). These containers are added to the custom panel, which should then layout these region containers along the timeline. Since the `Frame` of the ViewModel can be given to the layout panel as data binding, it knows how its visual size is mapped to the `Frame`. However the panel does not know anything about regions directly. All it sees is a collection of `ItemContainers` which is managed by the `ItemsControl`. There is no direct link back the model creating these containers. So how does WPF pass information relevant for layout?

In order to add layout information to children of a panel, WPF uses the concept of *attached properties*. Attached properties are dependency properties defined by an external class, but saved

for a specific dependency object (see Chapter 5). They allow extending existing UI Elements with additional properties. A layout container can then look up the value by calling the static getter function of the external class.

This can easily be explained by looking at how the `Grid` control is used. Listing 6.8 shows a grid with two rows and two columns. It contains four children, each assigned to one cell of the grid. This is done by setting the attached properties `Grid.Row` and `Grid.Column`.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefiniton />
        <RowDefiniton />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <TextBlock Grid.Row="0" Grid.Column="0" Text="00" />
    <TextBlock Grid.Row="0" Grid.Column="1" Text="01" />
    <TextBlock Grid.Row="1" Grid.Column="0" Text="10" />
    <TextBlock Grid.Row="1" Grid.Column="1" Text="11" />
</Grid>
```

Listing 6.8: Passing layout information to a grid

The same mechanism is now utilized to forward region bounds to the custom layout panel of the `RegionsView`. This is done by defining the two attached properties `StartPosition` and `EndPosition` in `RegionPanel`. Listing 6.9 shows the definition of the `StartPosition` attached property. It is registered as `TracePosition` property of the `RegionPanel` class. Additionally it sets the framework metadata option `AffectsParentArrange`. By setting this option, changing the property value invalidates the arrange layout pass, which is then reevaluated.

As shown in Listing 6.10, these attached properties are added to the container by setting them in an item container style. The property value is set by data binding to each region's `TimeFrameStart` and `TimeFrameEnd` properties. Using this information the custom layout panel of the `RegionsView` can position each container along the timeline.

```
public static readonly DependencyProperty StartPositionProperty =
    DependencyProperty.RegisterAttached(
        "StartPosition",
        typeof(TracePosition),
        typeof(RegionPanel),
        new FrameworkPropertyMetadata(null,
            FrameworkPropertyMetadataOptions.AffectsParentArrange)
    );

public static TracePosition GetStartPosition(UIElement element)
{
    if(element == null) throw new ArgumentNullException();
    return (TracePosition)element.GetValue(StartPositionProperty);
}

public static void SetStartPosition(UIElement element, TracePosition val)
{
    if(element == null) throw new ArgumentNullException();
    element.SetValue(StartPositionProperty, val);
}
```

Listing 6.9: Setting attached properties to region containers

```
<Setter Property="ItemContainerStyle">
    <Setter.Value>
        <Style>
            <Setter Property="Controls:RegionPanel.StartPosition"
                    Value="{Binding TimeFrameStart}" />
            <Setter Property="Controls:RegionPanel.EndPosition"
                    Value="{Binding TimeFrameEnd}" />
        </Style>
    </Setter.Value>
</Setter>
```

Listing 6.10: Setting the `ItemsContainerStyle` of an `ItemsControl`

6.4 Execution Transcript

The Execution Transcript View displays the entire executed code of a program run. Unlike other views, it does not use a horizontal timeline. Instead it lists all executed lines of code and shows the progression of statements along the vertical axis. Recall that the executed lines of code are generated by code reconstruction (see Section 4.3).

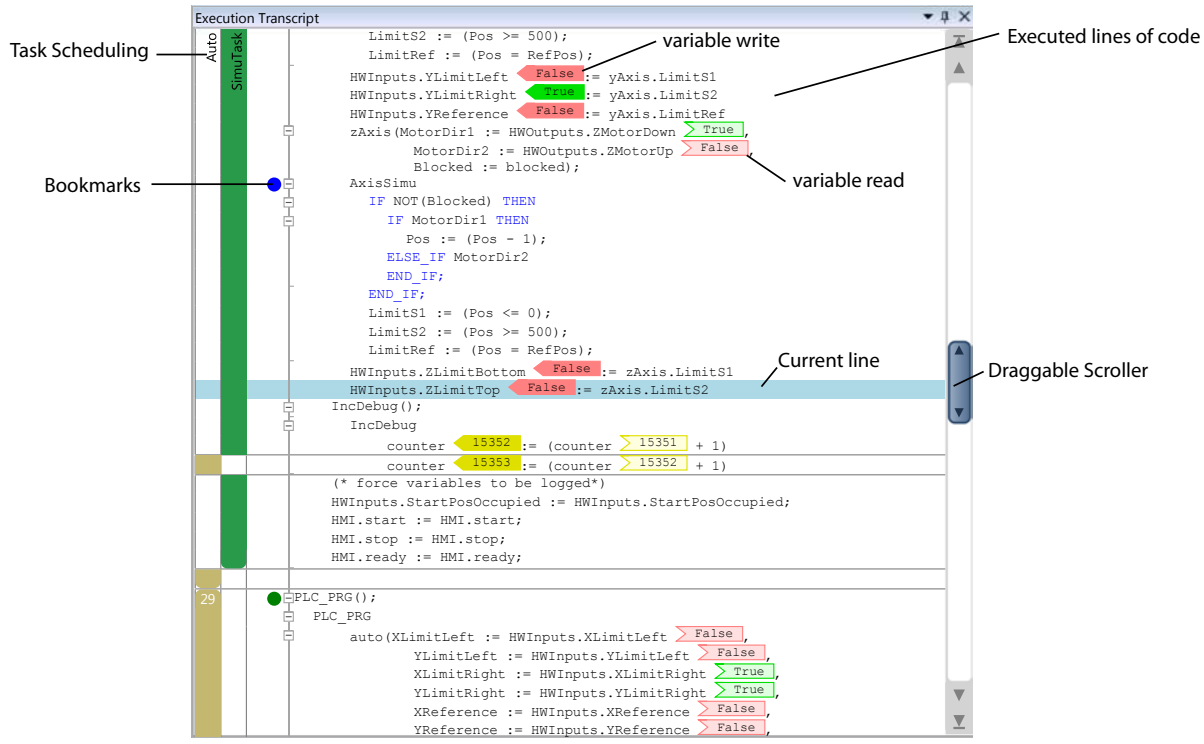


Figure 6.18: Execution transcript view

Figure 6.18 shows how these executed lines of code are displayed. The view can be split up into a main middle section, which displays the executed statements, and a small left section which shows the task interleaving.

In the left section each task is assigned a column. To determine which column belongs to which task, the task name is displayed above the column. These columns are used to visualize the task interleaving of the executed code. If a task column is filled with color, it means that the code next to it is executed by that task. Note that this creates a visualization similar to the task interleaving view. Each chunk creates an uninterrupted sequence of code lines. These sequences are delimited by horizontal gray lines spanning across the entire width of the view. The filled sections of each column between these gray lines can therefore be seen as the chunks of the program. These chunks are part of an execution cycle. Chunks drawn with a rounded top border mark the beginning of an execution cycle. Chunks with a rounded bottom border mark its end. The colors of chunks are the same as in the task scheduling and execution path view. They encode which execution path has been visited during the current execution cycle. A

further connection is created by drawing the execution path ID at the top of the first chunk of an execution cycle. Double clicking on the column of a task hides all lines of code of that task. This feature can be used to only show the code specific tasks.

Figure 6.19: Variable access types

The source code is displayed with syntax highlighting and indentation to increase readability. Indentation is based on the call depth and control structures. Note that while looking at the source code, several colorful objects can be seen embedded into the lines. These objects are also shown in Figure 6.19. They represent the variable values which are read or written by trace operations. Variable reads are illustrated as a colored rectangle containing the variable value. To emphasize that it is a read operation, an arrow points into this rectangle (to the right). Values of variable write operations are added at the left side of the assignment operator and recognisable by an arrow pointing outside (to the left). The colors of reads and writes depend on the data type of a variable. Boolean variables use red for **`FALSE`** and green for **`TRUE`**. Other types use a yellow color. Read and write operations are further distinguishable by using different fill colors. Writes are filled a dark version of their color, while reads remain lightly colored. The embedded variable values are also used for navigation. Clicking on them positions the active cursor of the view. This can be used to position a cursor in the variable view or any other timeline based view.

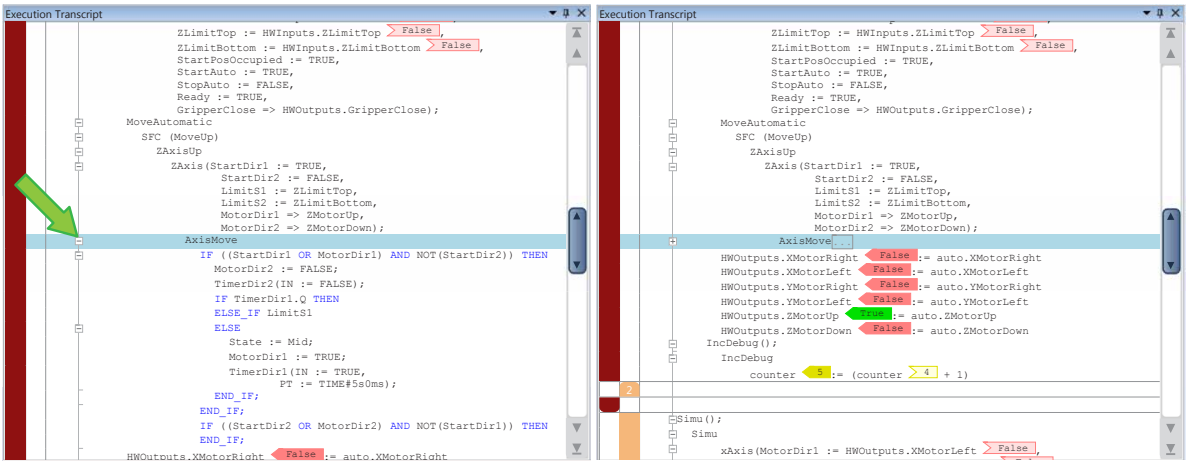


Figure 6.20: Folding of code blocks

Between the main section and the section showing task interleaving, a narrow vertical line is visible with multiple small boxes. It spans across each chunk. Developers which have used modern IDEs will immediately recognize this feature as code block folding. Figure 6.20 illustrates

this feature. It allows collapsing certain portions of code by clicking boxes. It even allows collapsing portions which are separated by task switches. Portions of code which have been collapsed are marked by a grey rectangle containing three dots (...). Hovering over this rectangle displays the hidden code in a tool tip.

Scrolling upward and downwards is done using a draggable scroller (the blue rounded rectangle seen in Figure 6.18 on the right). Depending on the vertical offset from its resting position in the center, the view scrolls slower or faster, i.e., it changes the scroll velocity. If the scroller is released, it moves back to its center position. Scrolling this way not only changes the visible portion of code, but also moves the current line cursor. This is a highlighted line which stands for the current position of this view. It tries to remain in the vertical center of the view while scrolling. Since scrolling to any specific location can take a very long time, the execution transcript can only be used reasonably in conjunction with other views. It exports its current line position as a cursor, which can be moved by other views, allowing it to perform large jumps.

Last but not least, several convenience features have been added. Since other views can force the execution transcript to jump to another locations, the history of such jumps is recorded. Using the back and forward buttons of a mouse or the back and forward buttons on the toolbar allows going back to previous positions. To remember a specific location, it is possible to set *bookmarks* next to each line. Clicking on the left of each line allows adding such a bookmark. Each bookmark can be given a unique color, a label and a description. They are drawn as round circles next to their line of code. Double clicking on a bookmark again removes it. All bookmarks are added to a bookmark list which is saved and persists over multiple program runs. Clicking on them performs a jump to the bookmarked line. Furthermore, bookmarks are not an execution transcript specific feature, but are also drawn on the timelines of other views.

6.4.1 Implementation

This view is implemented using a custom control, which derives from `ListBox`. Although there is some shared functionality between the two containers, the execution transcript really does not have anything in common with a regular `ListBox`. Creating a subclass and replacing the control template were the primary mechanisms used to write this control.

UI Virtualization

The reason why a traditional `ListBox` is not sufficient is the following: The control template of a `ListBox` or any other `ItemControl` subclass contains an `ItemsPresenter`. This placeholder object is used to add the `ItemsPanel` into the visual tree and fills it with elements from the `ItemsSource`. A `ListBox` puts this object into a `ScrollViewer` and uses a `StackPanel` as default `ItemsPanel`. Thus `ScrollViewer` contains an `StackPanel` object and allows scrolling if it get bigger than the available space. Listing 6.11 shows a `ListBox` style that contains the definition of a `ControlTemplate`.

```
<Style TargetType="ListBox">
    <Setter Property="ItemsPanel">
        <Setter.Value>
            <StackPanel />
        </Setter.Value>
    </Setter>
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="ListBox">
                <ScrollViewer>
                <!-- StackPanel is inserted here -->
                <ItemsPresenter />
                </ScrollViewer>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

Listing 6.11: Definition of a `ListBox` `ControlTemplate`

Although this might seem to be a reasonable choice for the execution transcript view, the main problem of this implementation is that it is only suitable for small collections. When the control is first displayed, every element of its `ItemSource` is loaded, put into a `ListBoxItem` container object, and added to the `StackPanel`. Not only does this create a huge `StackPanel`, it also wastes memory and increases loading times. Some collections, like the executed source code of a program, are too large to display this way. The visual representation of it uses too many controls. Luckily, there is a mechanism to handle large data sets. Microsoft calls this UI-Virtualization. Instead of using a regular `StackPanel`, `ListBox` actually uses the class `VirtualizingStackPanel` as `ItemsPanel`. This panel does not load everything at once, but only elements which are currently visible.

Using virtualization changes how scrolling is done. On MSDN, this is referred to as *physical scrolling* versus *logical scrolling* [13]. Physical scrolling is done by a fixed physical increment, like a pixel. It requires that panels know their own size (also known as *Extent*, see Figure 6.21). If this size is known, `ScrollViewer` can move a viewport smoothly. Yet, by activating virtualization, some information is lost. `ScrollViewer` can no longer rely on the result of the measure layout pass, since not all elements are loaded. Instead it requires the container panel to implement scrolling on its own by implementing the `IScrollInfo` interface. In its current implementation, `VirtualizingStackPanel` performs this task and switches to logical scrolling. This is done by scrolling element by element. The extent is set to the amount of elements in the collection. The current vertical offset is an element index. For small elements, this is a reasonable compromise between smooth scrolling and performance. However, once elements become almost as large as

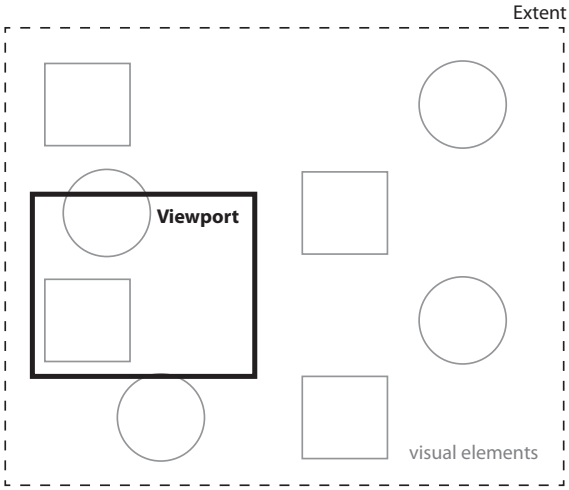


Figure 6.21: Relationship between viewport and extent of a scrollviewer

the available screen space, scrolling becomes clunky and unusable.

Implementing a custom virtualized scrollable view

As of writing this, there was no way to do smooth scrolling of large data sets and larger visual objects. Since `VirtualizingStackPanel` does not fit our needs, a custom virtualized panel had to be developed. Additionally, there are a number of reasons why `ScrollViewer` can not be used as well. For one, using scroll bars to navigate through millions of lines of code is very unintuitive. Moving the the scroll bar makes huge jumps, which is definitely not desired. Finding specific positions on this vertical axis also is almost impossible. Instead, other views are used to do jumps. The scroller only needs to allow moving forward or backward. This triggered the design of a scroller which does not scroll to a specific offset, but scrolls with a variable speed. This was inspired by a scroller which can be seen in Google Picasa [3].

The result of these efforts is a combination of physical and logical scrolling. Since there is no way of knowing the visual extent of each element before loading and measuring its container, the primary mechanism used to define the vertical position of the view is its element index. However, unlike logical scrolling, this index is no longer an integer. Instead, floating point numbers are used. The integer part of the number specifies the index of the current element. Its fractional portion defines the relative offset inside that specific element.

Scrolling and jumping to specific locations is fully separated in the `ICustomScrollInfo` interface. Setting the vertical offset is done by using an index and its fractional portion. Scrolling is done by specifying a vertical delta, which moves the visible portion and loads new elements when necessary.

```
var children = InternalChildren; // access before using ICG!
var generator = ItemContainerGenerator;
var itemsControl = ItemsControl.GetItemsOwner(this);
var pos = generator.GeneratorPositionFromIndex(itemIndex);
```

Listing 6.12: Creating a generator position object

```
using (generator.StartAt(pos, GeneratorDirection.Forward, true))
{
    while ( ... )
    {
        bool isNewlyRealized;
        UIElement child = generator.GenerateNext(out isNewlyRealized);

        if (isNewlyRealized)
        {
            AddInternalChild(child);
            generator.PrepareItemContainer(child);
        }
    }
}
```

Listing 6.13: Basic usage of a the `ItemsContainerGenerator` as iterator

Implementing a virtualizing panel

The heart of the execution transcript view is a custom virtualizing panel which manages item containers and scrolls them. Microsoft provides a basic framework for creating such views by subclassing `VirtualizingPanel` [2]. This hides many of the complicated details, such as resolving data templates and creating item containers. The `VirtualizingPanel` class provides a utility property, named `ItemsContainerGenerator` (ICG), which simplifies the generation of item containers. It helps create and manage a list of item container objects and filling them with data objects. Note that its implementation is independent of which subclass of `ItemsControl` is used. This means ICG creates `ListBoxItems` for `ListBox` and `ContentPresenters` for `ItemsControl`.

The ICG allows the forward or backward creation of list items in a very efficient way. The basic usage is to first create a position object, which marks the starting point of container generation. This is shown in Listing 6.12. Note that before accessing the ICG, a virtualizing panel must access its `InternalChildren` property. This is a known bug, which should hopefully be fixed in future releases of the .NET framework. The position object is an internal marker used by ICG to specify an element of an `ItemsSource`.

Using this position, the ICG is then accessed as a disposable iterator. It usage is shown in Listing 6.13. The direction of the iterator is set by the enumeration `GeneratorDirection`. Once the iterator is initialized, calling `GenerateNext` returns an items container suitable for the `ItemsControl` subclass. ICG reports back if a container has already been created for a given index by setting `isNewlyRealized` to FALSE. The virtualized panel's responsibility is to

add or insert newly created containers into the `InternalChildren` collection. Once this has happened, the ICG must further do final steps to prepare each item container for display. Once a containers is not needed, it must be removed from the generator by calling its `Remove` method.

Measure Pass

The main logic of the custom virtualizing panel is in its measure pass. All methods specified by the `ICustomScrollInfo` interface change the panel's state and trigger a new measure pass. Depending on this state, the panel either jumps to a new position or scrolls up or down.

Jumping to a new location is done by first removing all generated containers. Since the jump location is specified by an element index and a relative offset, this element is created first. Depending on the location of the object, further elements are loaded before and after the specified index. The current strategy is to create containers forward and backwards until the entire space of the panel is filled.

Figure 6.22 illustrates how UI-Virtualization works. Scrolling is done by modifying the visible area. New elements are loaded when necessary. Others are removed if they are no longer visible.

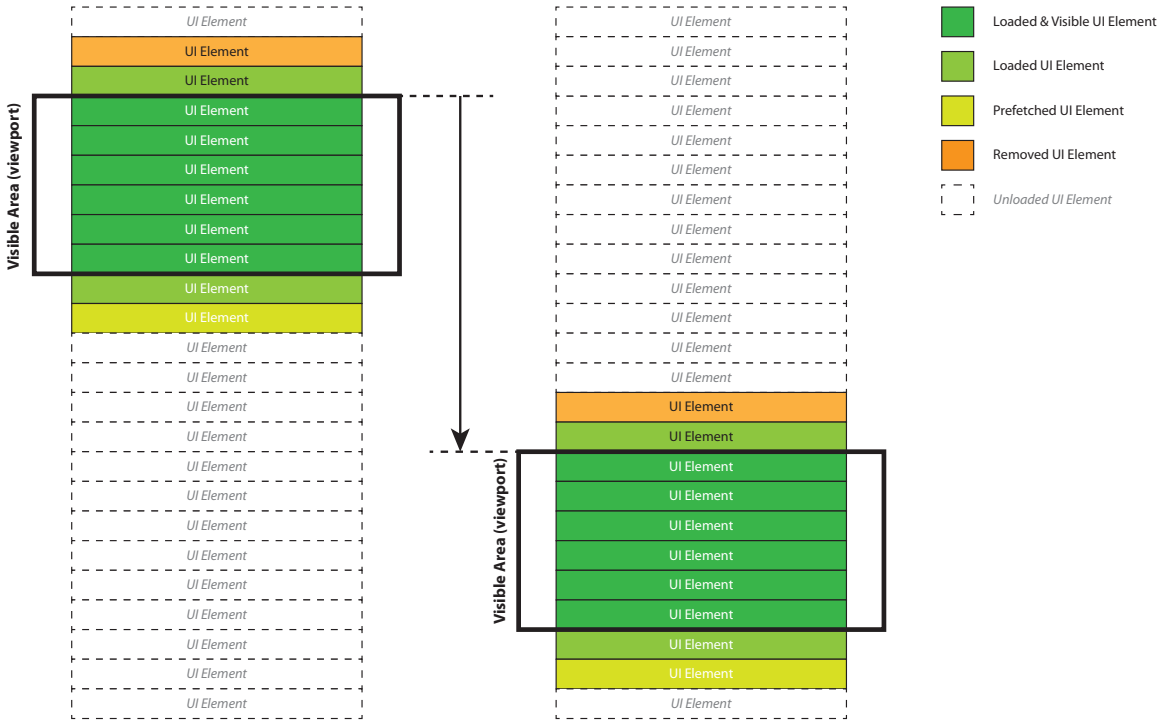


Figure 6.22: UI Virtualization

6.4.2 Optimizing performance

Several different mechanisms have been implemented to improve performance. In the following, some of them are described.

Using a background thread to load data

One of the problems when displaying large data sets is that loading pieces of information requires a certain amount of time. As a general rule, long lasting operations, such as accessing the data layer, should not be done in the UI-Thread. Doing so causes parts of the application to freeze and make it unusable until the operation finishes. It is therefore very common to delegate work to background threads, which notify the UI thread about changes. Data virtualization therefore takes place in such background threads. The primary mechanisms used to tackle these problems are the `Dispatcher` object and the utility class `BackgroundWorker` which are described in [15] and [6].

Virtualized Collections

As seen in Chapter 4, a lot of effort has been put into optimizing how data is loaded and into minimizing the memory footprint. Data layers introduced so far expose a completely synchronous API. What is still missing is exposing the generated/loaded data in a way which is compatible to WPF data binding. The last piece of the chain is a group of classes that act like collections, but internally use the lower level data layers to get their data. Containers like `ItemsControl` are optimized for using collections implementing the `ICollection` or `IList` interfaces. Because of the usage of UI-Virtualization in our custom panels, only a limited amount of data is loaded for display. Thus implementations like `VirtualizedCodeList` use these interfaces to expose their data, however, they do not actually have to load the entire collection at once. They form the link between UI virtualization and data virtualization.

In order to become a fully bindable collection, which intelligently updates the UI if it changes, they must also implement the `INotifyCollectionChanged` interface. This allows WPF controls to listen for changes and update their bindings when necessary. The interface itself contains an event which is fired every time something inside the collection happens. The default implementation of this interfaces can be seen in `ObservableCollection<T>`.

The final collections used by the execution transcript and other virtualized views are therefore observable, lightweight wrappers to the data layer. Since access to this data layer is sometimes quite expensive, element access is done using background threads. Access requests are put into a work queue of this background thread. Since loading takes some time, the collection immediately returns a more or less useful placeholder object. Once the background thread reports back its result, the collection sends a `CollectionChanged` event notifying all UI Elements bound to the data element that they should get the new, now loaded value.

Reducing visual complexity

Despite all the efforts to reduce loading times and memory usage on the data side, one bottleneck remained. In the first version of the execution transcript, each chunk was displayed using a `TreeView` for each filtered tree. Depending on the size of the tree, creating such `TreeViews`

turned out to be expensive. In one of the case studies, a chunk's code tree would take up to a 100 ms to create. Adding up a few of these delays rendered navigation unusable. To overcome this problem, chunks were further split up into single lines of code. Each line could be easily loaded by much simpler UI controls, which added syntax highlighting, indentation, folding, and bookmark support. The view model of each line, which is represented by `FilteredTreeNodes`, already contained enough information about the tree structure to recreate the look and feel of the tree view manually. The result of using smaller, much simpler UI controls is greatly improved UI element generation times. However, it also came with a cost. Since the virtualized collection still has to know its own size, the total amount of lines of code executed has to be calculated, which adds a somewhat costly precessing step.

Nevertheless, by making the execution transcript line based, navigation inside of it turned out to be much more productive. Developers can now jump to any specific line of code and add bookmarks to them. The benefits gained are greater than the cost of preprocessing data, which must be done anyway for many other reasons.

Chapter 7

KePlast Case Study

One of the first case studies created for this project was an application called KePlast. The original goal was to create a simplified version of already existing applications maintained by our industrial partner. The name KePlast originates from KEBA's application framework used to write control systems for injection molding machines.

These machines produce plastic parts by injecting liquid plastic into a mold. They consist of several moving parts and are usually powered by hydraulics. Vendors of such machines buy control systems from KEBA and implement their software on top of their existing application framework.

Since KEBA uses its own Kemro implementation of the IEC standard, writing an application which resembles the usage of this framework was not as easy as just taking existing code and porting it to CoDeSys. Therefore this application had to be written from scratch, by first understanding the basic structure of such an application and then implementing a simplified version in the CoDeSys system. Its primary goal was to capture the essence of what must be part of such a controlling application and how parts interact with each other. Note, that the application should provide a case study for validating both Capture & Replay, as well as the debugging tool. Therefore is was necessary to build the control system and a simulator model of the machine in CoDeSys. The following sections elaborate on how the machine was modeled and how the control application is structured.

7.1 Simulator model of an injection molding machine

Given the task of writing a control application for an injection molding machine, one first has to define how the machine should be modeled. Using this model a simulator of the machine was built, which provides a test setting for the control application.

Figure 7.1 shows a simple schematic of such a machine. The *clamp* (1) is a movable part which contains the mold. On the other side is the *injection component* (2). During each cycle, plastic is transported into the injection by a *screw* (3). Its temperature is regulated to a constant level, causing the plastic to melt. Each charge of plastic is then pushed into the *mold* (4) in a

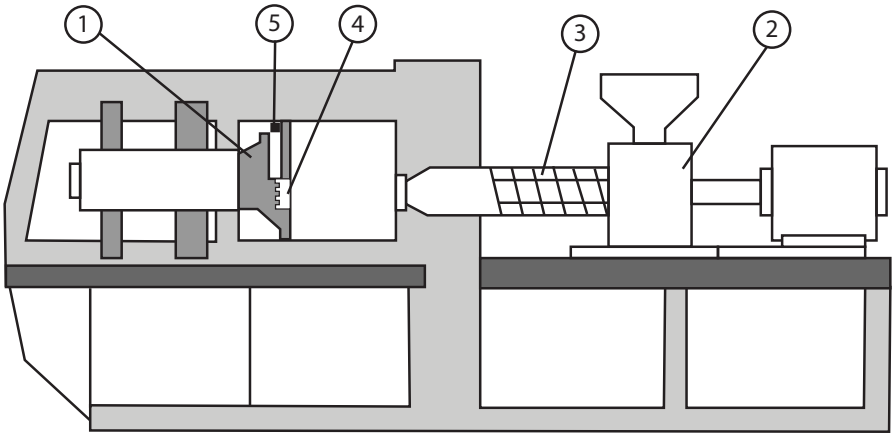


Figure 7.1: Injection molding machine

controlled manner. Finally, the clamp moves backward and an *ejector* (5) pushes the finished part out of the mold.

All movements are performed by hydraulics. As additional constraint, the machine only uses one hydraulic circuit, including one pump and a hydraulic motor. Numerous control valves are used to cause the different parts to move. Figure 7.2 shows the basic hydraulic circuit. It was taken from one of the handbooks of a machine vendor, but reduced to the minimum, excluding all things like filters and relief valves.

Using this definition of the target machine and some assumptions about size and weight of the various components, each movement was then modeled separately. These models were implemented in ST code and are run by a separate simulation task. To illustrate the derivation of such a model, the model of the clamp movement is explained in the following.

7.1.1 Clamp movement

The clamp's movement is caused by a hydraulic cylinder. It is controlled by two valves. One determines the flow direction of the oil, the other controls flow and pressure. Both clamp and the piston of the hydraulic cylinder can be seen as one rigid body which is influenced by pressure. It is hindered by speed proportional damping. Therefore the following movement equation can be formulated:

$$m\ddot{x} + d\dot{x} = F_p, \qquad F_p = p_1 A_1 - p_2 A_2$$

F_p is the force caused by the difference in pressure between the two chambers. A_1 is the effective area of the piston in chamber 1 and A_2 in chamber 2. This differential equation can be

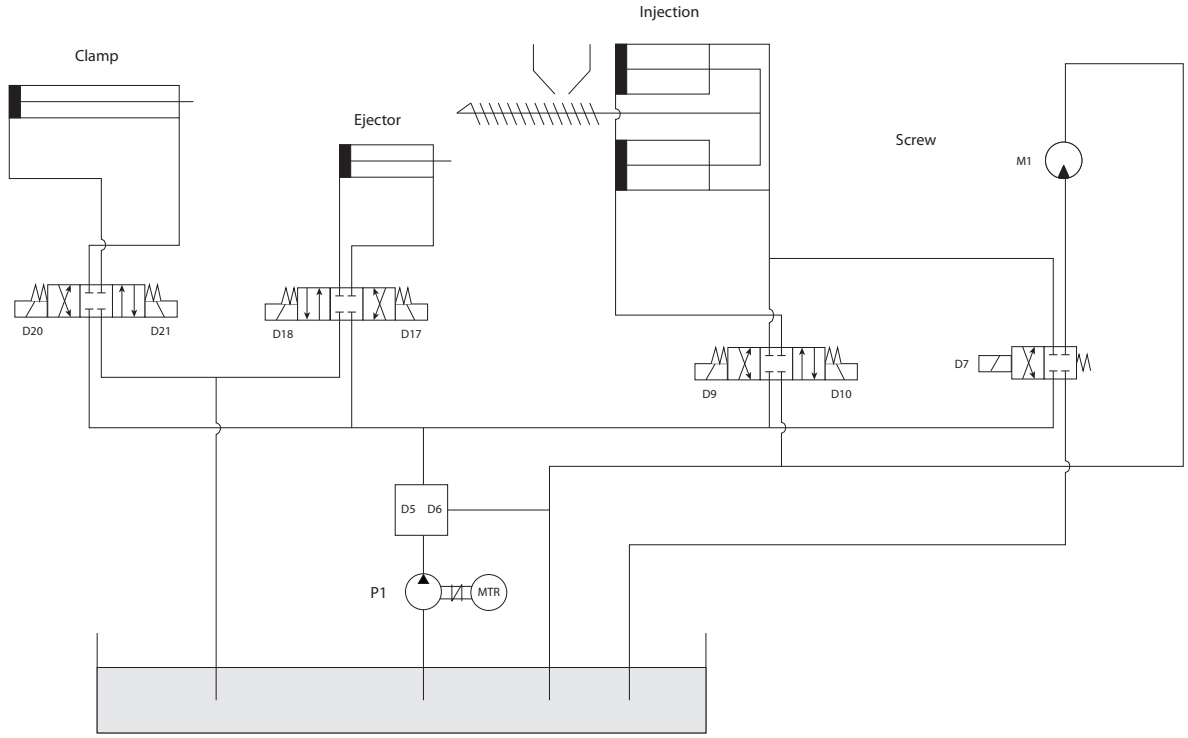


Figure 7.2: Simplified hydraulic circuit of the machine

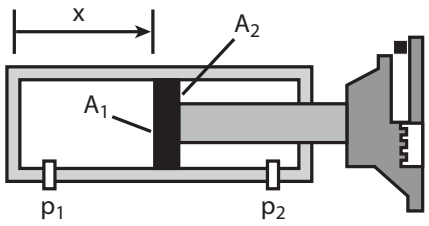


Figure 7.3: Modeling of the clamp's movement

formulated as continuous state-space model:

$$\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 \\ 0 & -\frac{d}{m} \end{bmatrix}}_{A_c} \begin{bmatrix} x \\ v \end{bmatrix} + \underbrace{\begin{bmatrix} 0 & 0 \\ \frac{A_1}{m} & -\frac{A_2}{m} \end{bmatrix}}_{B_c} \begin{bmatrix} p_1 \\ p_2 \end{bmatrix}$$

The continuous model can not be used directly for the simulation. A simulator runs in a cyclic task which is executed every millisecond. This means the state of the machine is only updated at discrete times. Both input and outputs are sampled with $T_a = 1$ ms. In the real world, the machine would still moves between each sample. To take this into account, one must implement the equivalent sampled model as discrete state-space model [12] instead:

$$\begin{bmatrix} x_{k+1} \\ v_{k+1} \end{bmatrix} = A_d \begin{bmatrix} x_k \\ v_k \end{bmatrix} + B_d \begin{bmatrix} p_{1,k} \\ p_{2,k} \end{bmatrix}$$

This model calculates the next state $\mathbf{x}_{k+1} = [x_{k+1}, v_{k+1}]^T$ after each cycle based on the current state $\mathbf{x}_k = [x_k, v_k]^T$ and input values $\mathbf{u}_k = [p_{1,k}, p_{2,k}]^T$. The matrices A_d and B_d can be derived by hand. They can also be easily calculated using the `c2d` function from the Control System Toolbox in MATLAB. However, knowing the symbolic solution simplifies parametrization of the source code. Note that these matrices are constant and therefore implementing this discrete system is quite easy.

Using the above model, a simulation of the clamp movement can be implemented. This simulator uses the discrete state-space model while the clamp is moving. Additionally it must check various boundary conditions. For example, the clamp can not move beyond the right wall in Figure 7.1. The model is connected to two global shared variables, one being the pressure applied to the piston and the other used to report the current position. These two are used by the controller task to position the clamp.

The clamp movement model is just one example of how the KePlast simulator is implemented. Other parts of the machine are modelled in a similar fashion.

7.2 Control application

The control application of this machine consists of several separate subsystems, which together create the desired process. These subsystems can be grouped into three different layers:

Output coordination The lowest layer accesses the hardware using shared variables mapped to hardware output and inputs. It ensures that valid outputs are sent to the hardware. In this case study, the hardware is replaced by a simulation task which reads and writes shared variables.

Movement control On top of the previous layer are subsystems dedicated to controlling one

particular task or component of the process. Most of them are unaware of each other and expose an interface for controlling their state.

Supervisor The movement subsystems of the previous layer must be coordinated and their execution triggered. This task is performed by the supervisor. It is basically a high level model of what the machine can do and what mode of operations are supported.

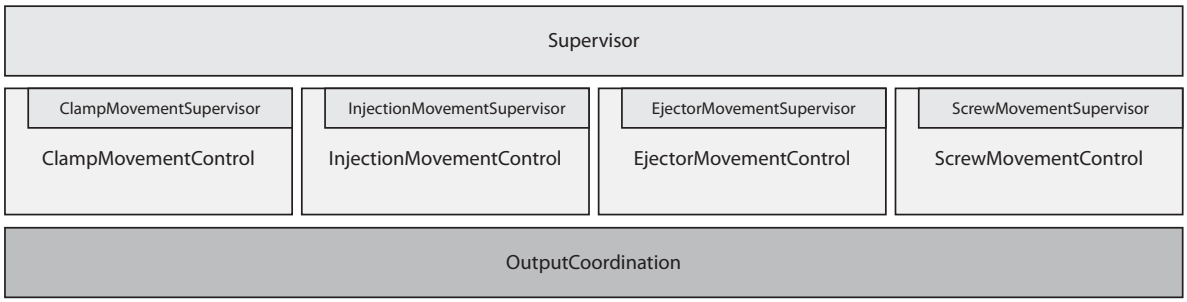


Figure 7.4: Layers of the control application

7.2.1 Output coordination

All subsystems controlling movements need to utilize the single hydraulic circuit. This means there has to be one task with mutual exclusive rights to access the control valves. These access rights and the current valve configuration is managed by a high priority task, named Output-Coordination. It acts like a switch, listening to output ports written by the different movement control systems and passing through the one who is active.

In order to enforce such a behaviour, a strict protocol has to be followed when accessing shared resources. Subsystems must first request access from the OutputCoordination. Only if the resource is not used, access is granted. It is then locked until the active process releases it.

7.2.2 Movement control

Each movement control system runs as a separate task and is split up into two parts. The main function block of the system implements the protocol stated above and turns on controllers when active. The second part runs in parallel and checks if the system is still in an error free state. If not, the current controller is stopped and the entire subsystem goes into an error state. Figure 7.5 shows the flow of control of these movement control blocks. They are implemented as SFCs.

Usually a control data structure is used to tell the movement system what should be done. Listing 7.1 shows how such a trigger message is prepared. Subsystems react to `bStart = TRUE` by first requesting access to the control valves. This is done by checking the control structure of the shared resource. It contains a Boolean flag `bFree` which indicates that no one is using the resource, as well as a `DeviceId` which stores the current control system ID locking the system.

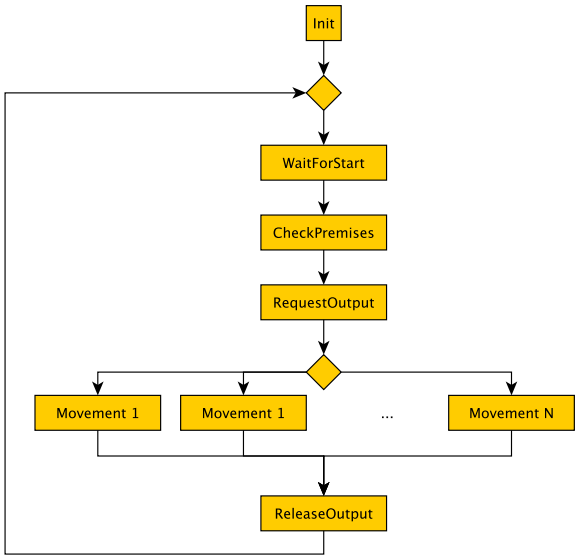


Figure 7.5: Structure of a movement control SFC

```
g_clampCtrl.pData.MoveId  := MV_CLAMP_CLOSE;
g_clampCtrl.pData.MoveDir := Fwd;
g_clampCtrl.bStart := TRUE;
g_clampCtrl.bDone  := FALSE;
```

Listing 7.1: Trigger message used to close clamp

The protocol now requires that if the shared resource is locked by the same `DeviceId` as requested, the resource stays locked and is immediately available for usage. If the `DeviceId` differs, one must wait until `bFree` is set **`TRUE`**, set it **`FALSE`** and then lock it with the new DeviceId. Once this is done, the saved `MoveId` is used to select the appropriate valve settings.

After checking any additional premises, the movement control system selects a controller based on the trigger message. These controllers run for a certain time until a given target is reached (location, pressure). Each movement is concluded by releasing the shared resource. This is done by setting `bFree = TRUE` and `DeviceId = NO_DEVICE`.

7.2.3 Supervisor

The control application provides an automatic, semi-automatic and manual mode of operation. In automatic mode, the machine produces as many parts as possible. Each movement is triggered automatically, according to the predefined list of operations. The same procedures are performed in semi-automatic mode, yet after each movement the end-user must trigger the next. Manual mode can be used to do any kind of single movement. This high-level view of the machine and its mode of operations are controlled by the Supervisor-Task.

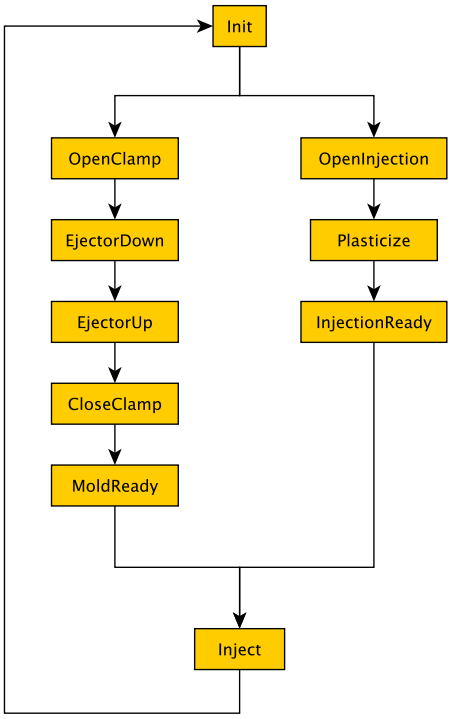


Figure 7.6: SFC for automatic mode

Automatic mode

This mode is implemented in a function block named `FB_Auto`. During each automatic cycle the machine is first brought into a predefined state. The clamp opens, any part inside the mold is ejected, the injection moves backwards, new material is plasticized and loaded by the screw. After closing the clamp, plastic is injected into the mold and the cycle starts again.

Figure 7.6 depicts the flow of control of the supervisor, which is implemented as a SFC function block. Each state triggers the responsible movement control subsystem and then waits until it reports back. It consists of a large parallel branch, which triggers two movement sequences that are independent from each other. One sequence consists of the movements of clamp and ejector, the other of injection and screw. Because of the hardware restrictions, only one parallel branch is active at a time. After both groups of the machine are ready, injection is performed.

7.3 Simulator Visualization

To illustrate the operation of the injection mold machine, a visualization of the machine simulator has been created using `CoDeSys`'s built-in visualization engine. Figure 7.7 shows a screenshot of this visualization. While connected to a PLC Simulator, this visualization displays the movement and state of the machine in real time. The *Measurements* tab seen in Figure 7.7 illustrates how the mold is filled. It also displays pressure and temperature values of the system. Figure 7.8 shows the *Valves* tab showing the active valve configuration of the hydraulic circuit.

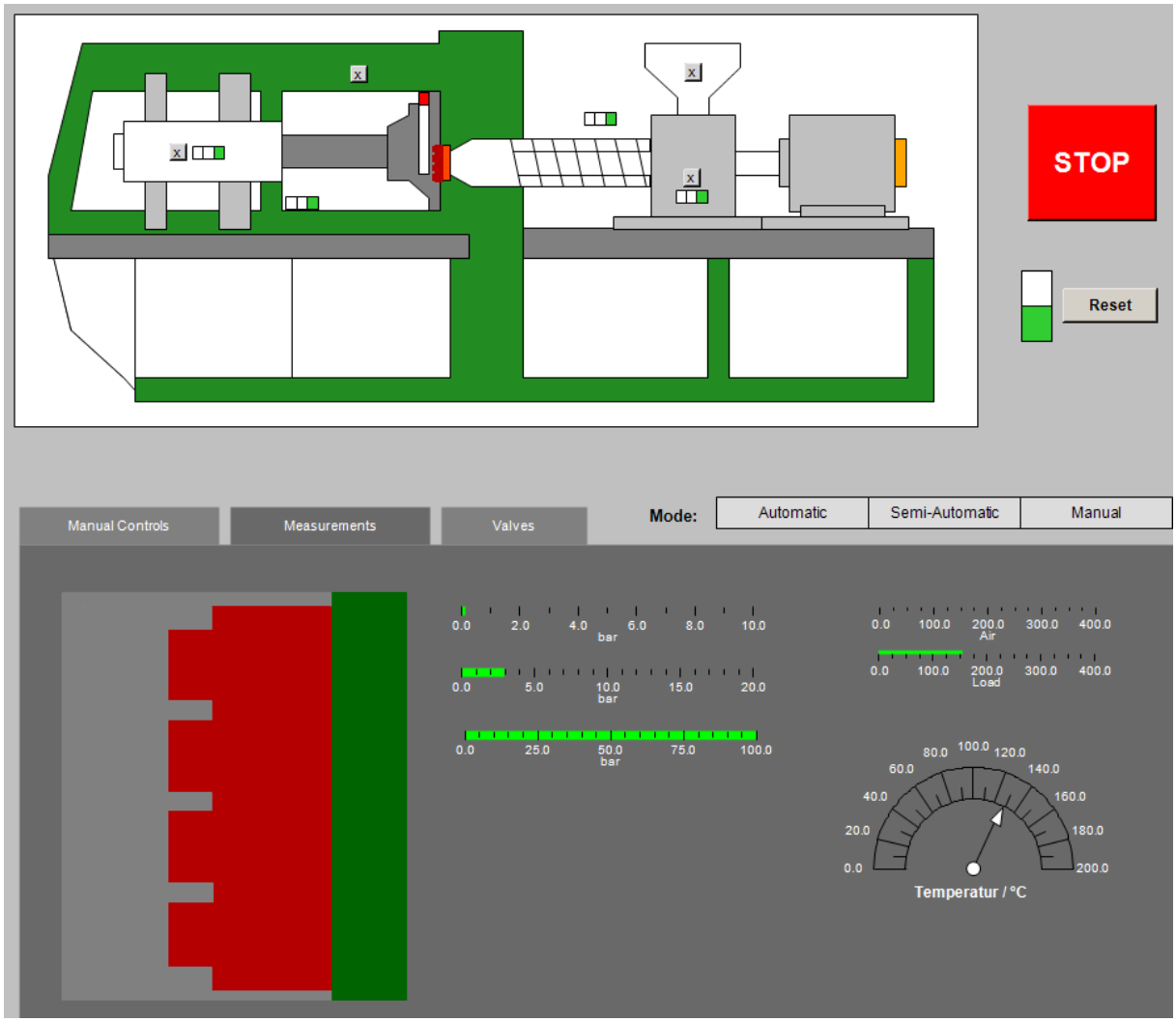


Figure 7.7: Visualization of the injection mold machine simulator

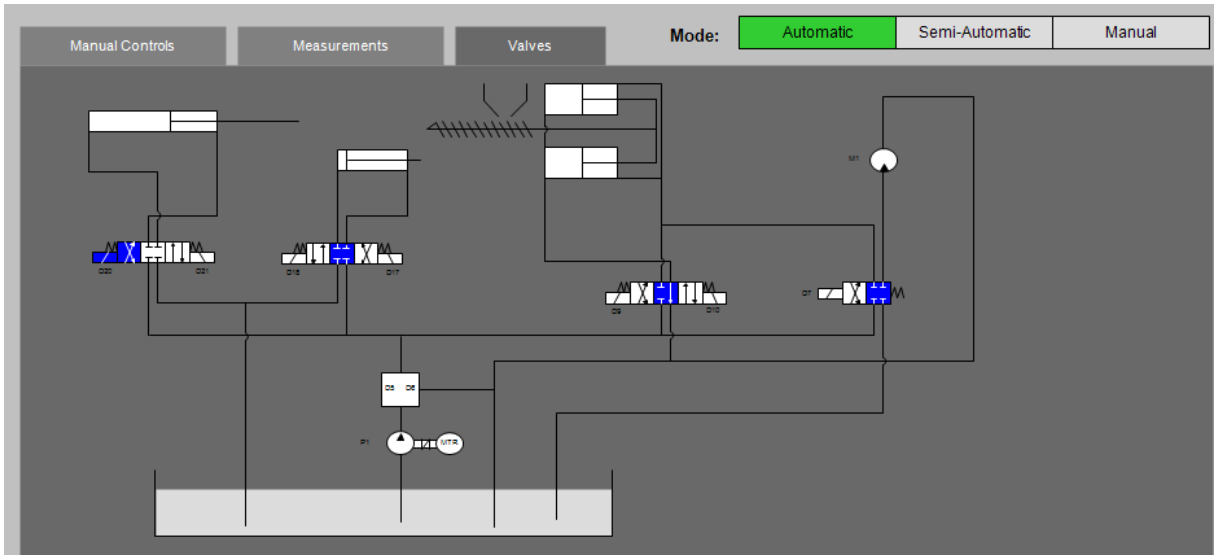


Figure 7.8: Visualization of the hydraulic circuit

Trace duration (hh:mm:ss):	00:12:42
Size of the combined trace file:	682 MB
Number of nodes:	145,178,215
Number of chunks:	2,899,230
Number of execution cycles (ClampTask)	152,591
Number of execution cycles (EjectorTask)	152,591
Number of execution cycles (InjectionTask)	762,955
Number of execution cycles (ScrewTask)	152,591
Number of execution cycles (OutputCoordinationTask)	762,956
Number of execution cycles (SupervisorTask)	152,591
Number of execution cycles (HeatingTask)	762,955
Total number of executed lines of code:	60,778,272

Table 7.1: Statistics of the shown KePlast recording

The Visualization also acts a frontend for control the machine's opereration. Switching operation modes is done by clicking on one of the Mode buttons. The active mode is highlighted in green.

7.4 Trace Visualization of KePlast

In the following multiple screenshots of the trace visualization are shown which displays a recording of the KePlast case study system. They illustrate the capacity of the tool to handle large recordings. Table 7.1 lists several key figures of the trace recording. This 12 minute long recording creates a combined trace file of 682 MB. It contains about 145 million trace operations which are loaded as nodes. These result in 2.9 million chunks.

Figure 7.9 shows about 20 milliseconds of the recording in the Task Scheduling View. In this visualization, both 5 ms and 1 ms execution cycles are recognizable, i.e., while there are many small active white regions which contain execution cycles of 1 ms tasks, only some active regions are wider and contain executions of 5 ms tasks as well.

The number of execution cycles in the recording depends on the cycle time of each task. Tasks with a cycle time of 5 ms have run about 152,591 times, tasks with a cycle time of 1 ms about 762,955 times. Figure 7.10 shows the Execution Path View displaying the entire recording at once. Thus it illustrates all execution cycles. The Execution Phase View which is placed above shows the execution phases of the *ClampTask*. In Figure 7.11 both views are shown at at higher zoom level. Figure 7.12 shows the Execution Path View at a zoom level, which illustrates individual execution cycles.

During all of these cycles, approximately 60 million lines of code are executed. Figure 7.13 shows the Execution Transcript View displaying the executed code of all seven tasks.

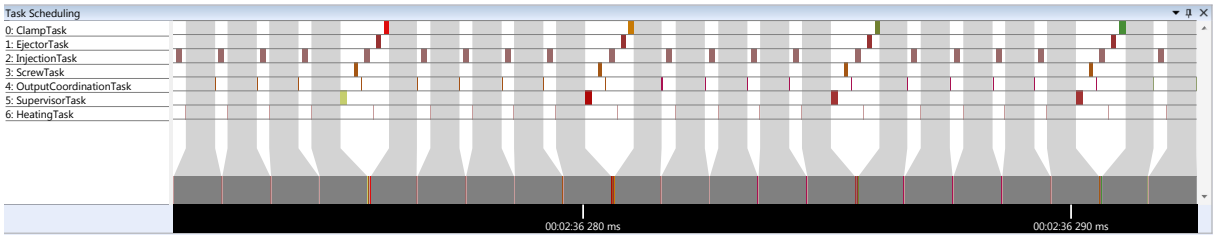


Figure 7.9: Task Scheduling View of the KePlast case study.

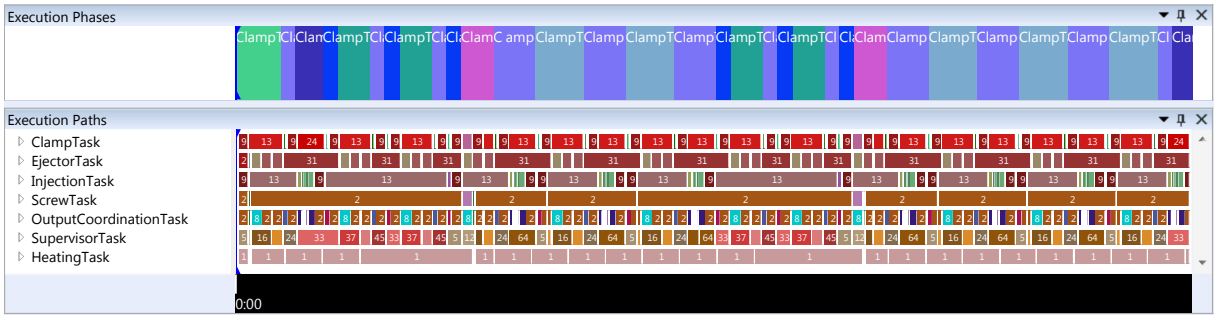


Figure 7.10: Execution Phase and Execution Path View of the KePlast case study

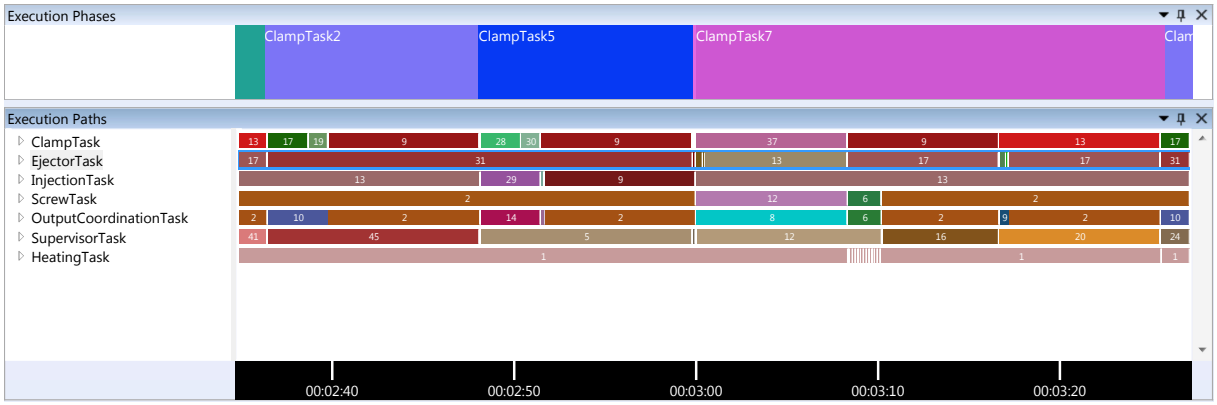


Figure 7.11: Execution Phase and Execution Path View at a high zoom level

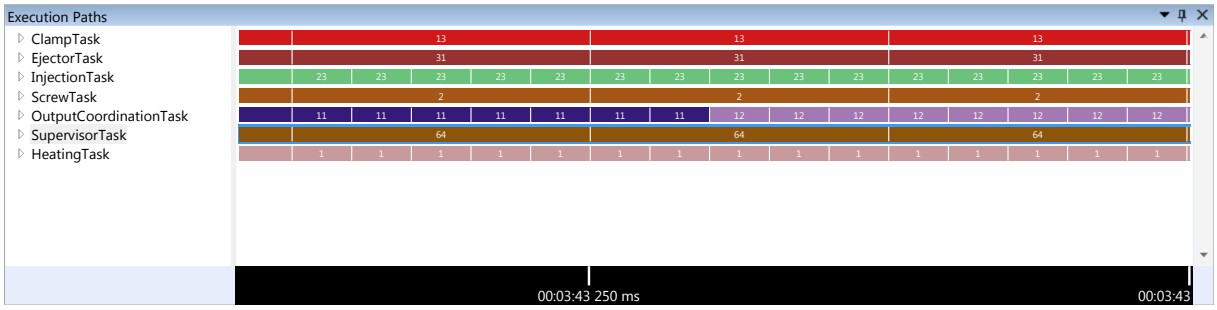


Figure 7.12: Execution Path View showing individual cycles

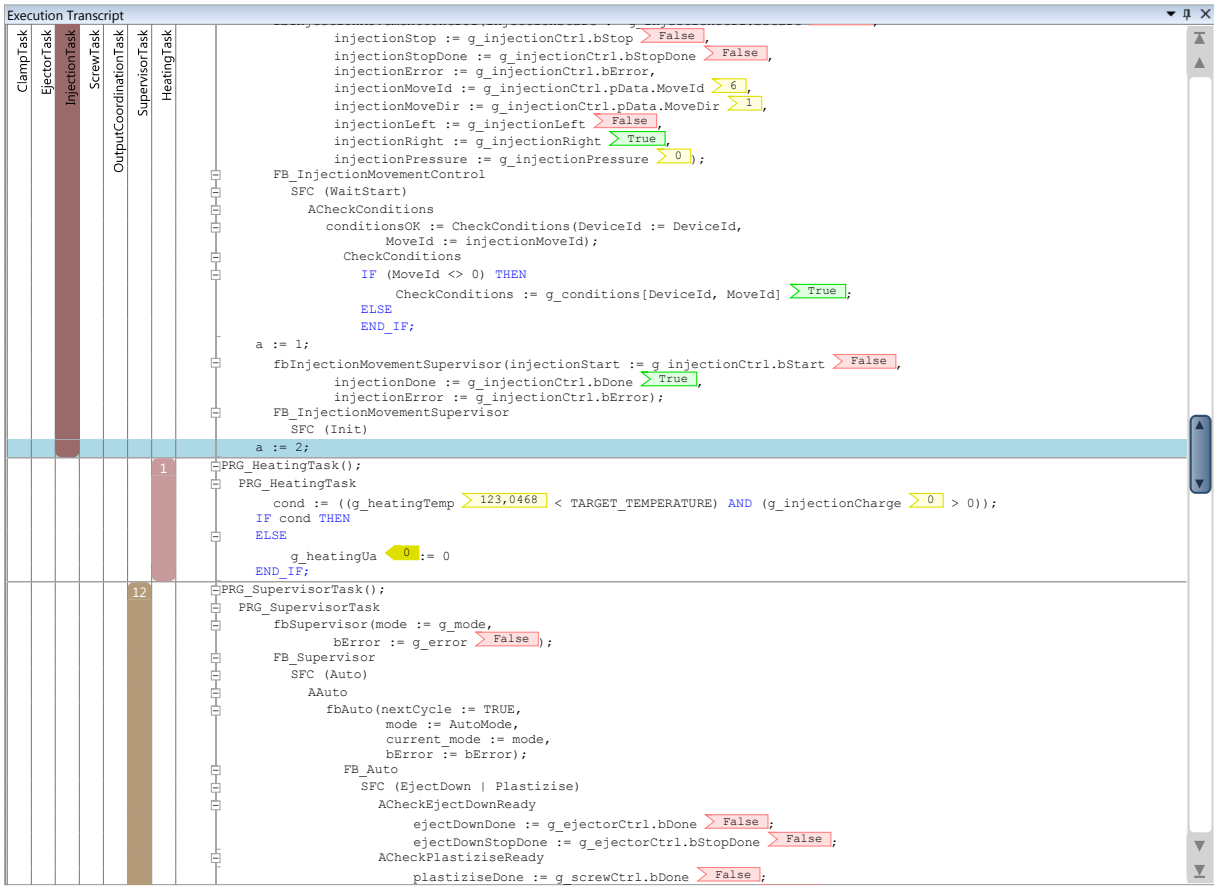


Figure 7.13: Execution Transcript View of the KePlast case study

Chapter 8

Conclusion

This thesis described the results of my work on the *Trace Visualization Tool* developed at the Christian Doppler Laboratory for Automated Software Engineering. It was part of a larger project, Capture & Replay, whose goal is the development of methods and tools for recording real-time PLC applications for deterministic replay, off-line debugging, and dynamic program analysis. The project is conducted in cooperation with and funded by KEBA AG.

This tool allows visualizing program runs and displaying it in various views to identify problems. High-level views are used to locate problematic phases during a recording. In a more detailed view, execution paths of task programs are displayed along a timeline. At the lowest level, task interleaving is shown to identify locations of possible data races. To find the cause of a defect, the entire executed code can be explored in an execution transcript. Unlike a regular single-step debugger, it allows navigating the entire program run and examining the program's state at any point in time. Support features such as bookmarks or jump lists, variable plots and diff views further support a PLC programmer.

The main challenges in creating this tool was handling the huge amount of data and preparing it for rendering complex visualizations. While writing basic UI applications is well documented, producing complex, interconnected and effective user interfaces is not. Sometimes producing a specific visual representation was possible using very simple methods. However, in many cases, standard solutions did not work. In this thesis I have outlined well established ways for structuring complex WPF applications and have shown how advanced visualizations tools for very large data sets can be created.

Developing this tool also required the development of a case study system. The KePlast case study was one of the PLC applications used to validate the approach of my colleagues and the functionality of my tool. It became a reference of the tool's capacity to display huge data sets.

Overall, the entire project is seen as a success. Our industrial partner has started the development of a new product which is based on the work of this project. The visualization tool will be a main component of this new product.

Bibliography

[1] *Avalon Dock Library*. URL: http://avalondock.codeplex.com/.

[2] Dan Crevier. *Implementing a virtualized panel in WPF (Avalon)*. Feb. 2006. URL: http://blogs.msdn.com/b/dancre/archive/2006/02/06/implementing-a-virtualized-panel-in-wpf-avalon.aspx.

[3] *Google Picasa*. URL: http://picasa.google.com/.

[4] John Gossman. *Introduction to Model/View/ViewModel pattern for building WPF apps*. Oct. 2005. URL: http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx.

[5] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321154916.

[6] *How to: Use a Background Worker*. URL: http://msdn.microsoft.com/en-us/library/cc221403(v=vs.95).aspx.

[7] *IEC, IEC 61331-3, Programmable controllers - Part 3: Programming languages*. 2003. URL: http://www.iec.ch/.

[8] Rick van der Lans. *Clearly Defining Data Virtualization, Data Federation, and Data Integration*. Dec. 2010. URL: http://www.b-eye-network.com/view/14815.

[9] Herbert Prähofer, Roland Schatz, and Christian Wirth. "Detection of high-level execution patterns in reactive behavior of control programs". In: *Proceedings of the Eighth International Workshop on Dynamic Analysis*. WODA '10. Trento, Italy: ACM, 2010, pp. 14–19. ISBN: 978-1-4503-0137-4. DOI: http://doi.acm.org/10.1145/1868321.1868324. URL: http://doi.acm.org/10.1145/1868321.1868324.

[10] *Programmable Logic Controllers - SIMATIC Controller*. URL: http://www.automation.siemens.com/mcms/programmable-logic-controller/.

[11] H. Prähofer et al. "A Comprehensive Solution for Deterministic Replay Debugging of SoftPLC Applications". In: *Industrial Informatics, IEEE Transactions on* 7.4 (2011), pp. 641–651. ISSN: 1551-3203. DOI: 10.1109/TII.2011.2166768.

[12] Kurt Schlacher. *Automatisierungstechnik 2*. 2008.

[13] *ScrollViewer Overview*. URL: http://msdn.microsoft.com/en-us/library/ms750665.aspx.

[14] Josh Smith. *WPF Apps With The Model-View-ViewModel Design Pattern*. Feb. 2009. URL: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

[15] *Threading Model*. URL: http://msdn.microsoft.com/en-us/library/ms741870.aspx.

[16] *Trees in WPF*. URL: http://msdn.microsoft.com/en-us/library/ms753391.aspx.

[17] *Using DrawingVisual Objects*. URL: http://msdn.microsoft.com/en-us/library/ms742254.aspx.

[18] *Visual Layer Programming*. URL: http://msdn.microsoft.com/en-us/library/ms753209.aspx.

[19] C. Wirth, H. Prähofer, and R. Schatz. "A multi-level approach for visualization and exploration of reactive program behavior". In: *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*. 2011, pp. 1 –4. DOI: 10.1109/VISSOF.2011.6069463.

[20] Christian Wirth. "Trace-based Reengineering and Analysis of PLC Programs". Institute for System Software, Johannes Kepler University, Linz, Austria, in preparation.

[21] *WPF Graphics Overview*. URL: http://msdn.microsoft.com/en-us/library/ms748373.aspx.

Listings

List of Figures

Time Travelling Debugger

Fehleranalyse bei SPS-Programmen nach IEC-61131-3

Diplomarbeit für: Richard Berger

Matr.Nr.

Deterministic Replay Debugging (DRD) ist eine Technik, bei dem Programmläufe so aufgezeichnet werden, dass sie später in einem Entwicklungssystem vollständig nachgespielt werden können. Damit ist es möglich, Fehler, die in einem Programmlauf aufgetreten sind, im Nachhinein zu analysieren und zu debuggen. Der große Vorteil dabei ist, dass beim Wiederabspielen der Programmlauf exakt nachgestellt werden kann, man aber bei der Fehleranalyse keinen Einschränkungen unterliegt.

In einer aktuellen Forschungsarbeit am Institut für Systemsoftware wurde ein Verfahren für *Deterministic Replay Debugging* von SPS-Programmen nach der IEC-61131-3 Norm entwickelt. Mit diesem Verfahren kann man direkt im Echtzeitbetrieb die Programme aufzeichnen und es lassen sich damit auch sporadische Fehler, die nur im Echtzeitbetrieb auftreten, nachspielen.

Ein Problem bei der Fehler- und Ursachenanalyse stellt die große Menge der Aufnahmedaten und die Länge des zu analysierenden Programmlaufs dar. Existierende Debugging-Werkzeuge, wie z.B. der Debugger in der CoDeSys-Programmierumgebung, sind wenig geeignet, um damit lange Programmläufe zu analysieren. In dieser Arbeit soll daher ein neues Werkzeug für die Fehlersuche konzipiert und prototypisch umgesetzt werden.

Das Werkzeug, genannt *Time Travelling Debugger*, soll folgende Funktionen bieten:

Mehrebenendarstellung des Programmlaufs:

Ein aufgezeichneter Programmlauf soll in mehreren Ansichten dargestellt werden können. Dazu gehört eine Darstellung des reaktiven Verhaltens in einem Gantt-Chart, die Darstellung des Task Scheduling und eine detaillierte Darstellung der ausgeführten Anweisungen in der sogenannten Code-View.

CodeView:

Eine besonders wichtige Darstellung ist die Code-View. Hier werden auf der Ebene der Anweisungen im Quellprogramm die ausgeführten Anweisungen gezeigt. Zusäztlich werden in jeder Zeile die Werte der Variablen angezeigt. Die besondere Herausforderung ist dabei, aus den Aufnahmedaten die ausgeführten Anweisungen zu bestimmen.

Diff-Anzeige:

Mit diesem Werkzeug soll es möglich sein, die Unterschiede von zwei vergleichbaren Abschnitten eines Programmlaufs zu zeigen. Es sind die in beiden Abschnitten unterschiedlichen Anweisungen durch entsprechende Einfärbung darzustellen.

Beliebiges Navigieren auf der Zeitleiste:

Es soll möglich sein, beliebig im Programmlauf vor und zurück zu navigieren und Sequenzen beliebig zu wiederholen.

Die Arbeit umfasst:

- Konzeption des Werkzeugs *Time Travelling Debugger* (in enger Zusammenarbeit mit dem Team am Institut für Systemsoftware)
- Implementierung des Werkzeugs in .NET/C#/WPF
- Integration des Werkzeugs in die SPS-Programmierumgebung CoDeSys
- Implementierung mehrerer Fallstudien zum Test des Werkzeugs; Implementierung in CoDeSys IEC-61131-3
- Test und Demonstration des Werkzeugs anhand der Fallstudien

Betreuung: Dr. Herbert Prähofer

Linz, 18. 2. 2011

Dr. Herbert Prähofer

Curriculum Vitae

Name	Richard Berger
Email	berger@ase.jku.at
Date of Birth	February 24^{th}, 1986
Citizenship	Austria
Parents	Christian Berger
	Yolanda Berger

Education and Professional Experience

2008 - present	Researcher at the Christian Doppler Laboratory for Automated Software Engineering at the Johannes Kepler University, Linz, Austria
2005 - 2012	Student of Mechatronics (Diplomstudium) at the Johannes Kepler University in Linz, Austria
2005	Military Service in Linz, Austria
2004	Matura with distinction at Schulverein Kollegium Aloisianum in Linz, Austria
1999 - 2004	Schulverein Kollegium Aloisianum in Linz, Austria
1997 - 1998	Jubail Academy, International School (American Division), Al Jubail, Saudi Arabia
1996 - 1997	Akademisches Gymnasium Linz, Austria
1992 - 1996	Volkschule Leonding, Austria

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. wörtlich oder sinngemäß entnommene Stellen als solche kenntlich gemacht habe.

Linz, Jänner 2012 Richard Berger