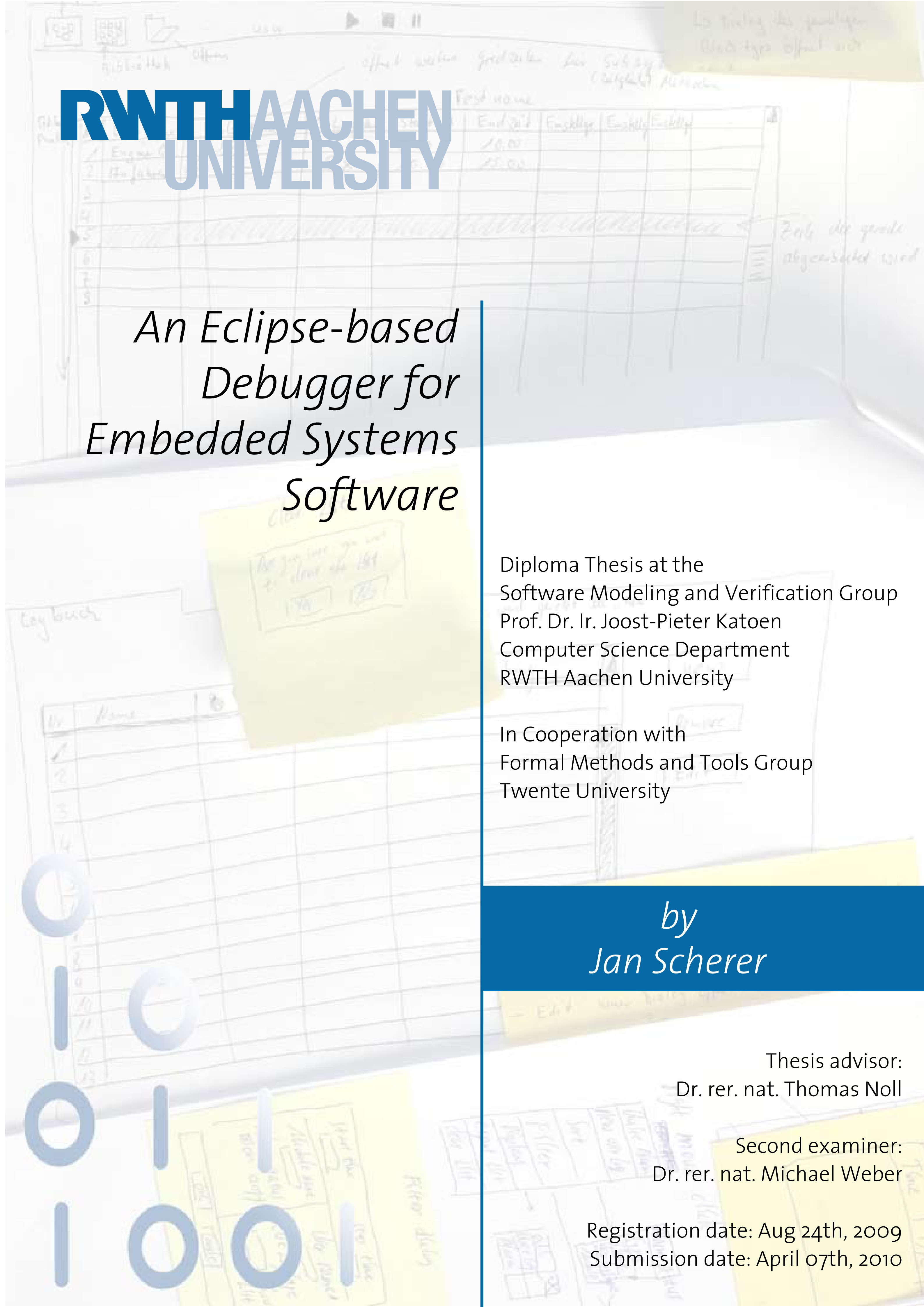


RWTH AACHEN UNIVERSITY

An Eclipse-based Debugger for Embedded Systems Software

Diploma Thesis at the
Software Modeling and Verification Group
Prof. Dr. Ir. Joost-Pieter Katoen
Computer Science Department
RWTH Aachen University

In Cooperation with
Formal Methods and Tools Group
Twente University

by
Jan Scherer

Thesis advisor:
Dr. rer. nat. Thomas Noll

Second examiner:
Dr. rer. nat. Michael Weber

Registration date: Aug 24th, 2009
Submission date: April 07th, 2010

Talk: An Eclipse-based Debugger for Embedded Systems Software

Abstract

The identification of software failures in embedded systems software has been recognized as tremendously important activity due to the fatal consequences caused by erroneous software. The focus of this talk addresses debugging as technique for the localization and correction of failure causes.

I will discuss different approaches for embedded systems debugging and in the following introduce the embedded systems debugger that has been developed in the context of this diploma thesis. The ESD debugger does not only provide support for standard execution control capabilities such as breakpoints and single-stepping, but more importantly also supports reverse execution. Allowing to run execution backwards to previous moments in time is a very powerful concept and helps to localize defects with more comfort and less time.

While debugging allows to reason about known failures it provides no support to guarantee system correctness. Model Checking on the other hand is a formal verification technique that allows to prove system correctness, however, existing model checkers usually provide insufficient support for counterexample analysis. The ESD debugger creates synergies between both techniques and allows to analyse model checking counterexamples. The talk is concluded by a tool demonstration that presents the capabilities of the ESD debugger in more detail.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, 07. April 2010
Jan Scherer

Contents

1 Introduction

The extensive and increasing use of embedded systems across industries designates an evolution in science and technology. They are integrated into everyday products, spanning consumer electronics such as mobile phones or digital cameras, transportation systems including automobiles, trains or airplanes, and medical equipment like MRI scanner or radiation therapy machines. Embedded systems are information processing systems that are embedded into larger products and that are normally not directly visible to the user [Mar06]. They consist of both hardware and software and are designed to perform a dedicated function. The dedicated task often comprises of controlling the embedding system and requires frequent interaction with the physical environment. Embedded systems are most often associated with the employment of microcontrollers, however, viable hardware also includes Digital Signal Processors or Programmable Logic Controllers.

An integral part of the design and development of embedded systems consists of validating the system under study. System validation describes all activities that can be employed to make sure that the developed system satisfies the given requirements. The respective requirements have been determined at an earlier point in the development process, and are are usually captured by natural language descriptions. It is of uttermost importance for the manufacturer to perform the validation task with great care to make sure that a shipped product does not exhibit failures. A failure thereby corresponds to a system behaviour that violates the specification. Failures which are detected after shipment often cause a large economic burden and in extreme cases may even lead to the manufacturer's ruin. For safety-critical embedded systems, such as flight control systems or airbag control units, the economical consequences are secondary in case the erroneous system behaviour has consequences on human health or life.

One approach to system validation is described by the activity of testing. Here, the system under study is stimulated with certain inputs and the resulting execution is analysed whether the shown system behaviour satisfies the specification. The total number of existing program executions usually exceeds the number of tested executions by several orders of magnitude. The difficulty is then to choose the inputs in a way that a broad range of possible execution are observed, including rare corner cases. A further technique concerned with system validation is described by the method of simulation. Simulation follows a similar approach as testing and analyses the system behaviour with respect to specific input. In contrast to testing, a simulation study does not work on the system itself, but instead on a system model.

The objective of testing and simulation is the detection of erroneous system behaviour. However, neither activity provides help to reason about failure causes. The main focus of this work addresses debugging as technique for the localization and the correction of failure causes. Although incorrect system behaviour can be caused by both erroneous hardware and software, the focus of this work is limited to software problems. Before we continue with an introduction of source-level debugging, it is important to discuss the terminology associated with the concept of failure as the respective terms, such as bug, defect, error, fault or infect, are used in a non-consistent manner across literature. Following the terminology introduced by Kowalewski [Kow06] we will use the term *defect* to denote an incorrect part within the program code. By this means the term defect is used synonymously with failure causes. Defects may go unnoticed in case the incorrect code is not executed but usually result in a *fault*, that is, an incorrect program state. Faults are internal to the program being executed and similar to defects may go unnoticed for a long period of time. The moment the program's behavior observably violates the specification we speak of a *failure*. The connection between the introduced terms can be summarized by the following cause-effect chain: *Defect* $\rightarrow$ *Fault* $\rightarrow$ *Failure*. Debugging can then be rephrased as activity which is concerned with the localization and correction of defects for known failures. An emphasis is on the adjective *known*, because debugging is not concerned with detecting failures in the first place.

The localization of software defects can be achieved by manual inspection of the source code and is, for example, successfully applied in the pair programming technique to prevent the introduction of defects during the development phase. However, in following development stages, a manual source code inspection promises only little success to locate defects. This is primarily because the programmer is required to construct a hypothesis with respect to the current program state but has no possibility to verify his conception. Instrumenting the source code with print statements to output the values of key variables can partially make up for this problem. But at the same time the usage of print statements has the disadvantage that the programmer has to specify what to display before executing the program. If one realizes that key information is missing, the program has to be edited to add additional print statements and then recompiled and restarted. Trying to output as much information as possible usually makes things worse, as considerable time is necessary to make sense of the output.

1.1 Source-level debugging

The approach to debugging we are interested in is characterised by the employment of a special tool, that is, a source-level debugger. Embedded systems software is usually developed in a low-level programming language such as C and/or assembly and is compiled to machine code in a subsequent step. The resulting machine code has little resemblance with the original code, but as developers think in terms of their source code it is essential that the debugger is able to provide the illusion that the developed code is directly executed on the target. Debuggers that possess this ability are referred to as source-level debuggers, henceforth just debuggers. The responsibility of the debugger is to control the execution of the considered program, and to provide information about the current program state once execution is suspended. Instead of executing the program until the system crashes or the program terminates the developer is able to employ breakpoints to suspend execution whenever specified conditions are met. While the program remains in suspended state the developer is able to inspect provided program context information, such as the current source line, variable values, or the program call stack. The listed context information target the program's source-level view. The provision of architecture specific information, such as the values of existing registers, the memory content or the current machine instruction is important to receive a complete picture of the current program state. In order to continue execution, a debugger usually provides at least two possibilities: An unconditional resume, resumes the program's execution until either another breakpoint is hit or the program terminates. Single-stepping allows to execute the program step-wise, whereas the granularity of a step usually varies between a single machine instruction or a statement in the source program. An advanced execution control concept is described by the possibility to reverse execution. It frequently happens that during debugging the programmer lets execution pass beyond a particular interesting point. Reverse execution is a powerful concept and allows the programmer to resume backwards to reach the interesting point without the necessity to restart the program.

1.2 Approaches for Embedded Systems Debugging

Debugging is an activity that is not restricted to embedded systems software but can be performed on other types of software as well. However, debugging embedded systems software is more difficult than debugging software written for general purpose computers. This is because embedded systems face tight resource constraints and are usually not able to host a debugger. Even if the embedded system has sufficient resources to run a debugger, the situation is complicated further because typically there does not exists a conventional way to control the debugger, for example, with a keyboard and a monitor. The solution to this problem is to host the debugger's user interface on the machine used to develop the program and either remotely control the target system or simulate the system in software. In the following we will discuss three possible approaches to embedded systems debugging. The first two approaches remotely connect to the system under study and rely on modified or additional hardware to provide debugging functionality. The third approach is software-based

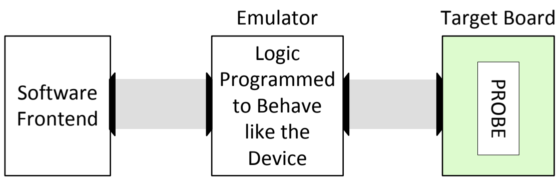


Figure 1.1: In-Circuit Emulator Block Diagram

and employs an instruction-set simulator.

1.2.1 In-Circuit Emulator

An In-Circuit Emulator (ICE) replaces the microcontroller on the target system with an emulation processor, that takes over the tasks of the original microcontroller. The emulation processor is located on the in-circuit emulator, which is connected to the target hardware with special adapters. A software frontend, which runs on a host system, is the developer's handle to the ICE. The ICE provides full visibility and access into the inner workings of the emulated microcontroller. Besides the support for standard debug features like run-control, break- and watchpoints, most ICEs provide additional high-end debug features such as complex triggers, trace memory or performance analysis. Complex triggers support the detection of complex events such as a write access to a specific memory location in conjunction with a particular value that is written. With the help of a trace memory component it becomes possible to record all or specific instructions.

The emulation of a microcontroller is a complex task and is oftentimes realized by using a special version of the microcontroller, a so-called bond-out device. The term *bond-out* refers to the fact that some of the internal signals are brought out to external pins to provide access to internal data registers, program memory and peripherals. Bond-out devices are expensive to produce because they basically require a complete redevelopment of the original processor. The usage of Field Programmable Gate Arrays (FPGA) has been considered as less expensive alternative to bond-out devices. However, FPGAs will not run as fast as bond-out devices or emulate analog circuitry such as Digital to Analog Converters.

1.2.2 On-Chip Debugger

On-Chip Debuggers also represent a hardware-based approach to embedded systems debugging but in contrast to in-circuit emulators the debug circuitry is integrated into the target MCU. The on-chip debug circuitry typically interfaces to the outside world via a serial interface, with JTAG being the most popular standard used. (JTAG, stands for Joint Test Action Group, the name used for the IEEE 1149.1 standard titled Standard Test Access Port and Boundary-Scan Architecture). The user's PC interfaces to the target via an adapter device. In the case of a JTAG connection this adapter device corresponds to a JTAG Emulator which provides a JTAG interface and a USB/Ethernet connection to the host PC. The on-chip debug solution implies a trade-off between the power of the debug logic and the cost for producing

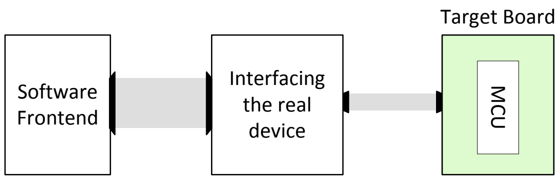


Figure 1.2: On-Chip Debugger Block Diagram

this additional functionality. As consequence on-chip debuggers provide only limited debug features, typically only run-control and simple breakpoints are supported. From a user standpoint of view, both approaches, that is on-chip debugging or the usage of a in-circuit emulator are quite similar. Differences can be found in the price to pay and the powerfulness. Commercial product names sometimes add an additional burden when trying to distinguish between these approaches; the AVR JTAG ICE [Atm09] for example is not an in-circuit emulator but rather an on-chip debugger. Nevertheless, because of the problems with emulating high speed processors, the choice is usually not between on-chip debugging and in-circuit emulation. The choice is between on-chip debugging and nothing.

1.2.3 Instruction-Set Simulator

A different approach to embedded systems debugging employs an instruction-set simulator (ISS) that creates a virtual representation of the target system. This software-based approach allows developers to compile the program code for the target microcontroller but then execute the resulting machine code on their workstations. The instruction-set simulator provides insight into the target microcontroller and allows fine grained execution control. Depending on the simulator's complexity, it is possible to keep track of the machine cycle count and also maintain caching and pipelining behaviour of the target microcontroller. A strong point in favor of the software-based approach is the fact that an ISS may exist prior to actual hardware availability. This makes it possible to integrate debugging much earlier into the design and development process. In contrast to a hardware-based approach and the typical resource limits introduced by the high costs of the adapter or emulation devices, the software-based approach allows multiple developers to work simultaneously without additional costs. An ISS also eases retargeting, as much of the simulator's code base can be reused and only architecture specific parts must be redeveloped to support a different target system. Despite the advantages, the usage of an ISS for embedded systems debugging faces difficulties not found in the former setting. The target system operates in an environment which greatly affects its behaviour and in order to make meaningful statements about application code executed within the ISS it becomes necessary to additionally model the target environment. This additional effort often proves to be very useful as the availability of a microcontroller simulation model together with an environment model can not only be used for debugging application code but also allows for formal verification techniques, such as Model Checking.

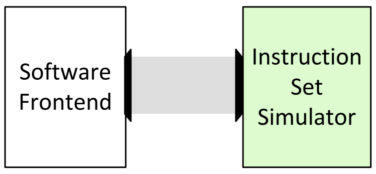


Figure 1.3: Instruction-Set Simulator Block Diagram

1.3 Model Checking

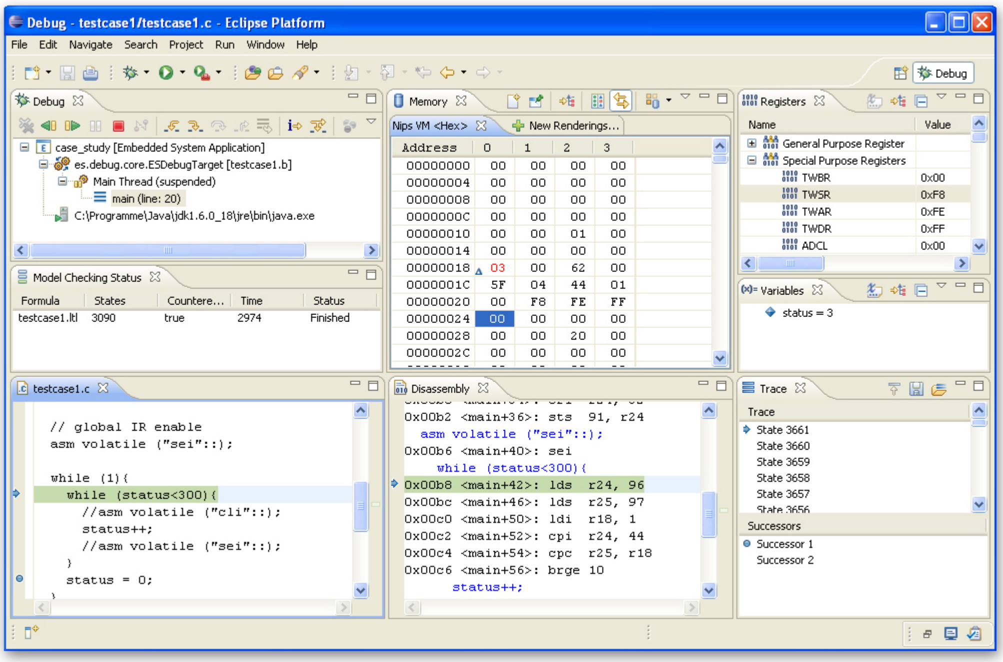
As embedded systems are oftentimes employed in environments where failure is unacceptable, the need for validation strategies to prove or disprove the system's correctness is obvious. The above mentioned validation strategies, that is testing and simulation, are able to show the existence of failure, however, they are not able to prove their absence. Model Checking on the other hand, is a validation strategy that is able to prove the correctness of systems and has been successfully applied to embedded systems software. Model checking operates on a model of the system and verifies it against temporal logics specifications. In case the model violates a specification a counterexample is generated that exhibits a failure. The analysis of counterexamples is an important part of the model checking process, but unfortunately this importance is not reflected in existing model checking tools. A minor objective of this work is to study how source-level debugging can improve the analysis of counterexamples.

1.4 Contribution

We have developed a source-level debugger for embedded systems software based on an existing instruction-set simulator, namely the NIPS VM. The working title of the developed debugger is ESD and stands for Embedded Systems Debugger. The employed NIPS VM is a virtual machine for state space generation and runs a custom-designed bytecode language. Transforming an input program to a NIPS bytecode program assigns an operational semantics and is a prerequisite for a subsequent program simulation. Our decision to employ a software-based approach to debug embedded systems software has primarily been made to study the symbiosis between debugging and model checking. The NIPS VM supports this aim as it provides integrated model checking capabilities on top of the state-space generation functionality.

Integrating both activities, that is debugging and model checking, as close as possible into the development process has been a further goal of this project. Building on an integrated development environment is the best conceivable approach in our opinion. The Eclipse platform is a precision fit candidate and has been chosen due to its high market share, acceptance and participation in industry and its support for custom tool integration [Ecl10b].

The ESD debugger does not only provide support for standard execution control capabilities such as breakpoints and single-stepping, but more importantly also supports reverse execution. Allowing to run execution backwards to previous moments in time is a very powerful



```
case_study [Embedded System Application]
  es.debug.core.ESDebugTarget [testcase1.b]
    Main Thread (suspended)
      main (line: 20)
  C:\Programme\Java\jdk1.6.0_18\jre\bin\java.exe
```

Formula	States	Countere...	Time	Status
testcase1.ltl	3090	true	2974	Finished

Address	0	1	2	3
00000000	00	00	00	00
00000004	00	00	00	00
00000008	00	00	00	00
0000000C	00	00	00	00
00000010	00	00	01	00
00000014	00	00	00	00
00000018	03	00	62	00
0000001C	5F	04	44	01
00000020	00	F8	FE	FF
00000024	00	00	00	00
00000028	00	00	20	00
0000002C	00	00	00	00

Name	Value
General Purpose Register	
Special Purpose Registers	
TWBR	0x00
TWSR	0xF8
TWAR	0xFE
TWDR	0xFF
ADCL	0x00

```
status = 3
```

```
  // global IR enable
  asm volatile ("sei"::);

  while (1){
    while (status<300){
      //asm volatile ("cli"::);
      status++;
      //asm volatile ("sei"::);
    }
    status = 0;
```

```
0x00b2 <main+36>: sts  91, r24
  asm volatile ("sei"::);
0x00b6 <main+40>: sei
    while (status<300){
0x00b8 <main+42>: lds  r24, 96
0x00bc <main+46>: lds  r25, 97
0x00c0 <main+50>: ldi  r18, 1
0x00c2 <main+52>: cpi  r24, 44
0x00c4 <main+54>: cpc  r25, r18
0x00c6 <main+56>: brge 10
      status++;
```

```
Trace
State 3661
State 3660
State 3659
State 3658
State 3657
State 3656
Successors
Successor 1
Successor 2
```

Figure 1.4: Embedded systems debugger screenshot

concept and helps to localize defects with more comfort and less time. Despite the unquestioned advantages of being able to resume backwards, only few existing debuggers provide support for reverse execution.

With respect to program context information, our debugger provides source-line information, values of variables and the program call stack. Context information is also provided for the machine-level and includes disassembly information, values of registers and memory contents. A screenshot of the running ESD debugger is depicted in Figure 1.4.

1.5 Organization of the thesis

The thesis is organized as follows:

Chapter 2 (Using the Nips VM as Instruction-Set Simulator) – introduces the NIPS VM and presents the toolchain for building a NIPS bytecode program. This includes an overview of an example microcontroller architecture, that is, the architecture of the AVR ATmega16 microcontroller, followed by an explanation how a particular microcontroller architecture is represented within the virtual machine.

Chapter 3 (Eclipse) – gives an overview of the Eclipse platform and its underlying architecture. The focus is thereby on the mechanisms required for custom tool integration. Additionally we will also discuss the platform's support for building custom debuggers.

Chapter 4 (Embedded Systems Debugger) – presents the high-level architecture of our debugger and introduces the existing functionality with respect to controlling the program's execution and inspecting program context information.

Chapter 5 (LTL Model Checking) – states the model checking problem for finite transition system and the logic LTL and presents an explicit state algorithm based on a nested depth-first search. Afterwards we will study the connection between debugging and model checking.

Chapter 6 (Case Study) – exemplifies the described content by provision of a case study. The example program gives a good intuition of the challenges one is faced when trying to locate a defect. At the same time the case study demonstrates the powerfulness of the ESD debugger.

Chapter 7 (Technical Realization) – provides details regarding the implementation of the ESD debugger and discusses the developed components including their dependencies. A focus is thereby on the debug protocol which establishes a connection between debugger and debuggee.

Chapter 8 (Related Work) – presents work that is related to this project.

Chapter 9 (Summary and Perspective) – summarizes the described content and discusses open issues.

2 Utilizing the NIPS VM as Instruction-Set Simulator

Debugging embedded systems by the use of an instruction-set simulator provides several advantages compared to a hardware-based approach. Nevertheless, the efficient implementation of such a simulator is a complex task. This is especially true, if the simulator should be retargetable and provide support for different microcontrollers. In this context, *different* refers less to similar members of a microcontroller family and more to microcontrollers of different architecture and vendor. Matters get complicated further, if we take into account that software for embedded systems does not exist in a uniform representation but can be found in various forms and language levels. Languages include low-level programming languages such as C or assembly but also high-level modeling languages such as UML.

The existing NIPS VM [WS05] is not only able to simulate embedded systems, but more importantly is retargetable and supports embedded systems software in different input languages. Taking into account that its capabilities also includes model checking, we decided to employ the NIPS VM as instruction-set simulator for the ESD debugger. The introduction of the NIPS VM is topic of the following section.

2.1 NIPS Virtual Machine

The development of the NIPS VM has been motivated by the fact that most model checkers are monolithic and as such do not provide a separation between model, state space generation and verification algorithms. This makes it cumbersome and most often even impossible to extend their functionality by integrating additional verification algorithms or to reuse exist-

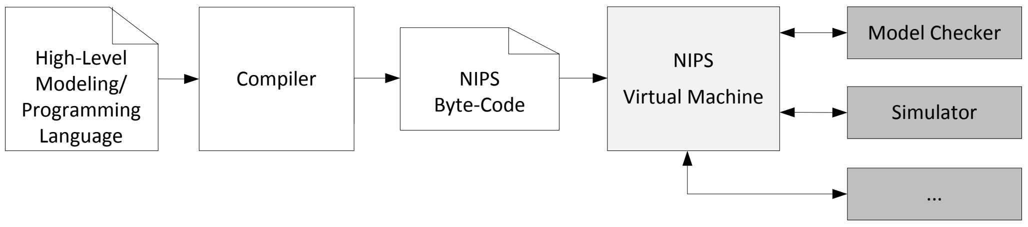


Figure 2.1: NIPS-based tools

ing models with different tools. In this context the NIPS VM has been developed as virtual machine for state space generation which is easily embeddable into custom tools. For the application in this work it is important to point out, that possible tools do not only include model checkers but also simulators. Embedding the NIPS VM into a custom tool requires minimal work due to the provided simplistic interface. The NIPS VM is implemented as a successor function and an embedding tool interacts with the machine by requesting the generation of successor states for a given machine state and by attending callbacks from the machine when a successor state is produced. Integration into a host application yields a modular architecture with a clean separation between model, state space generation and algorithms working on top of the state space. Besides the achievable modular architecture, the NIPS VM is also designed to be language-independent: It runs a simple bytecode which captures the operational semantics of a high-level modeling or programming language. The bytecode language has been developed as an abstraction layer between high-level input language and tools and its generality allows the translation of arbitrary high-level languages. The support for embedded systems software in various languages and levels is therefore fulfilled. A high-level overview of a NIPS-based tool interacting with the NIPS VM is shown in Figure 2.1.

Design

The design of the NIPS VM has been influenced by an initial aim to allow the execution of PROMELA models. From a user's perspective this influence can be recognized in the virtual machine's support for executing concurrent processes that communicate with each other through channels. At implementation level this influence is manifested in the virtual machine's state representation, which corresponds to a state in the model's state space. It has been ensured that this representation and the means to operates on it are general enough to not only support PROMELA, but to allow the execution of arbitrary high-level modeling and programming languages.The virtual machines state representation is depicted in Figure 2.2 and consists of a global variable space, a set of processes and a set of channels.

The global variable space is of fixed size and cannot be changed during the execution of a bytecode program. Variables in the global variable space can be identified by their offset and size in a corresponding byte array.

As already mentioned the NIPS VM supports the dynamic creation of processes. Processes

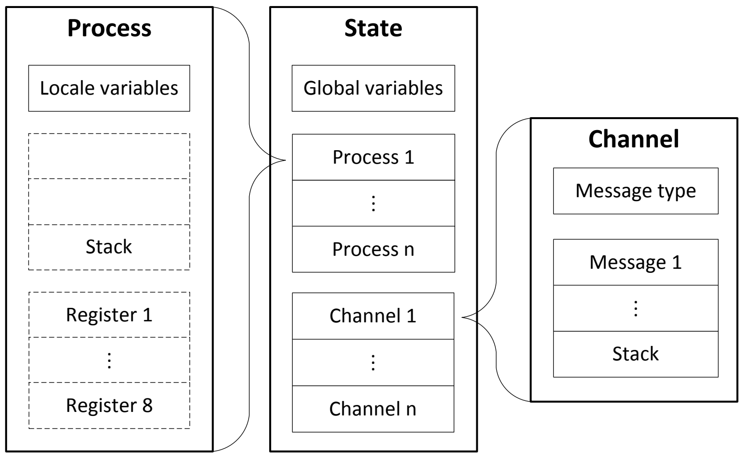


Figure 2.2: NIPS Virtual Machine State

have access to the global variable space and additionally may contain process-local variables in their local variable space. Each process is identified by a numeric process id and further contains a program counter which holds the address of the bytecode instruction that the process will execute next. At any moment a single process will be active and allowed to execute an atomic process step. A process step is not identical with a single bytecode instruction but is explicitly defined by two delimiting *Step* instructions. All bytecode instructions between those two *Step* instruction are executed by a process while the remaining processes have no possibility to interrupt. The active process can rely on a temporary operand stack and temporary registers for expression evaluation. After a process step is finished, the choice which process will be active next is either resolved by global non-determinism or is already determined by the fact that the current process is privileged to execute until it decides to release this exclusive execution right.

The NIPS VM also supports inter-process communication by the use of channels. Channels can be dynamically created by processes and exists in two variants for synchronous as well as asynchronous communication.

At this point we have seen, that the NIPS VM enables the execution of high-level modeling and programming languages but until now we have not discussed how the microcontroller hardware is represented by the virtual machine. In the following we will provide an overview of the ATmega16 microcontroller as example for a typical microcontroller used in embedded systems. Subsequently we will investigate how the microcontroller architecture can be captured by the NIPS VM.

2.2 The ATmega16 microcontroller

The ATmega16 is a CMOS 8-bit microcontroller manufactured by Atmel [Atm05]. The microcontroller is part of the AVR microcontroller family and based on an enhanced RISC architecture. The instruction-set contains 131 instructions, and with only a few exceptions these instructions execute in a single clock cycle. The ATmega16 is typically used to control sensors and actuators in industrial control systems.

Memories

The ATmega16 contains 16K bytes of in-system programmable flash memory for program storage. Flash memory is non-volatile, meaning memory contents are retained when microcontroller power is lost. Since all instructions are 16 or 32 bits wide, this memory component is organized into 8K locations, with 16 bits at each location. For addressing these 8K program memory locations, the ATmega16 features a 13 bit program counter. The ATmega16 is additionally equipped with 512 bytes of byte-addressable EEPROM for the storage of data that must be retained during a power failure or reset. Examples for this kind of data include logging data, fault data or system parameters. While this memory component is also non-volatile it is too slow to allow efficient computations. The ATmega16 therefore features 1120 byte of volatile SRAM, which consists of 32 general purpose working registers GPR_i $(0 \leq i \leq 31)$, 64 I/O registers IOR_i $(0 \leq i \leq 63)$ and 1024 bytes of internal data. During program execution, the internal data section is used to store global variables, support dynamic memory allocation of variables, and provide a location for the stack. Addressing the SRAM memory is supported by five different addressing modes: Direct, Indirect, Indirect with Displacement, Indirect with Pre-decrement, and Indirect with Post-increment. The general purpose registers 26 to 31 feature the indirect addressing pointer registers. These addressing registers are each implemented as two 8-bit register and are named X, Y and Z.

Status register

The Status register which is located in IOR_{63} is updated after an arithmetic operation and contains information about the result of the executed operation. This information includes flags which indicate a zero or negative result or a carry or half carry. The status register additionally contains the *Interrupt Enable Flag* I and the *Bit Copy Storage Flag* T.

Stack and stack pointer

A stack is used by the ATmega16 for storing temporary data, for storing local variables and most importantly for storing return addresses after function and interrupt calls. The stack is implemented in the SRAM data space and grows from the highest memory location to lower memory locations. A stack pointer is used to locate the top of the stack and is implemented as IOR_{61} and IOR_{62} in I/O space.

Digital input/output

The ATmega16 is equipped with four 8-bit digital I/O ports designated PORT A to D. Each port has three registers associated with it: the *Data Direction Register (DDRx)* is used to set a specific port pin to either output (1) or input (0). The *Data Register (PORTx)* allows to write output data to the port or alternatively control the activation of pull-up resistors in case the port pin is configured as input. Lastly, reading the actual value of the port is supported by the *Input Pin Address (PINx)*. Besides the input/output functionality, most port pins have additional functionality.

Timer and Counter

The ATmega16 features two 8-bit timers (Timer 0, Timer 2) and one 16-bit timer (Timer 1) to perform tasks, such as the generation of a pulse width modulation signal or the counting of external events. The operation of these timers relies on a number of registers which we will discuss in the following.

Each timer possesses a *Timer Counter Control Register* $TCCR_i$ which allows to specify the operational mode the timer should operate in. Modes of operation include Normal operation, Clear Timer On Compare Match, Phase Correct PWM and Fast PWM mode. Additionally, the control register is used to select the source of the timer clock. Besides the possibility to choose the internal system clock or an external clock as timer clock, a prescaler can be used. A prescaler allows to divide the system clock frequency by a factor of 8, 64, 256 or 1024. The current value for each timer unit is stored in the *Timer/Counter Register* $TCNT_i$ and the *Output Compare Register* OCR_i allows to specify a user-defined value that is continuously compared with the timer value kept in the former register.

As already mentioned, the timers can be used in different operational modes. In Normal mode the timer will continuously count from 0 to the highest possible value. Whenever the timer register value is reset to 0, the *Timer Overflow Flag* TOV_i will be set. The *Output Compare Flag* OCF_i is set when the timer value is equal with the value found in the compare register. Together with the *Timer/Counter Interrupt Mask Register* TIMSK these flags can be used to trigger internal interrupts. In Clear Timer on Compare Match (CTC) mode the value of the compare register defines the highest possible timer value

Interrupts

A typical requirement for embedded systems is the ability to react to planned but unscheduled events. The requirement usually manifests itself in an interrupt system, that provides means to execute certain actions with the occurrence of a specific event. The specific set of actions assigned to an event is known as an interrupt service routine. The ATmega16 features an an interrupt systems that contains 21 different interrupt sources. Three of these interrupt sources are external interrupts that are triggered by the occurrence of a specific event, for example, a rising or falling edge on a specific port pin. The remaining 19 interrupt sources support the operation of the peripheral subsystems and include various timer and counter events. The activation of an interrupt source is controlled by an individual enable bit in combination with the Global Interrupt Enable bit located in the Status Register. The assembly language instructions SEI and CLI allow to easily change the global status of the interrupt system, and activate or deactivate the interrupt system by modifying the Global Interrupt Enable bit. In case the interrupt system is enabled, the microcontroller will check the occurrence of possible interrupt events before executing a normal instruction. If an enabled interrupt is triggered, the corresponding interrupt service is executed. When entering an interrupt service routine the ATmega16 globally deactivates the interrupt system to prevent the occurrence of nested interrupts.

Sleep Modes

The ATmega16 features six different sleep modes (Idle, ADC Noise Reduction, Power-down, Power-save, Standby or Extended Standby) that can be utilized to minimize the microcontrollers power consumption. The utilization requires the selection of a specific sleep mode and a global activation by employing the *Sleep Mode Select Bits* and *Sleep Mode Enable Bit* located in the MCUCR register (IOR_{63}). The microcontroller is placed into the selected sleep mode by using the SLEEP assembly instruction and awakened with the occurrence of an enabled interrupt. The microcontroller then executes the interrupt handler and resumes execution from the instruction following SLEEP. Differences between the existing sleep modes can be found in the associated interrupt sources that cause the microcontroller to awake.

2.3 Toolchain

Previously, we have stated that the instruction-set-based simulation of a microcontroller program in the NIPS VM requires the translation of the application program to NIPS bytecode. This translation step is an essential part of the simulation process, as it not only has to preserve the semantics of the application program, but also creates a mapping between the microcontroller state and the NIPS VM state. For a specific microcontroller and an associated instruction-set the creation of a translation component is a one-time effort. However, this approach does not provide much flexibility and trying to simulate a different microcontroller would require the redevelopment of the translation component. This issue has been has been addressed by Rohrbach in the Model Checking Embedded Systems Software (MCESS) project [Roh06] and resulted in the design of a domain-specific language for microcontroller specification. The project further demonstrated the capabilities of the NIPS VM as state space generation component for custom tool development: a model checker for embedded systems software has been established. Relevant for this work is the availability of an existing tool chain to utilize the NIPS VM as an instruction-set simulator for embedded systems software.

While software for embedded systems may exists at various levels and languages it is a widely known fact that C- and/or assembly code is used in nearly every embedded systems software project at some point. We have therefore decided to concentrate our development effort to embedded systems software written in C and/or assembly code. Instead of trying to directly translate C source code to NIPS bytecode, it was decided to compile it with an off-the-shelf C compiler first. From the generated binary executable it is then possibly to extract assembly code which is translated to NIPS bytecode in a further step. This way we can not only process C and Assembly code in one go but also find software defects introduced by the used C compiler. The toolchain for utilizing the NIPS VM as instruction-set simulator is depicted in

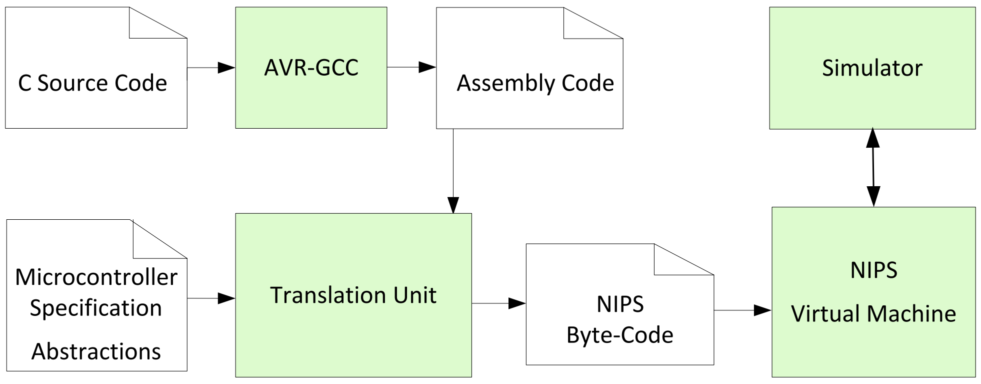


Figure 2.3: Toolchain for utilizing the NIPS VM as instruction-set simulator

Figure 2.3. Besides the mentioned assembly code, the translation unit that is responsible for generating NIPS bytecode, takes a microcontroller specification and environment abstractions as input. The following sections will introduce the last two artifacts in more detail.

2.4 Domain-specific language for microcontroller specification

The domain-specific language for microcontroller specification (RTL-DSL) allows to describe the characteristics of a microcontroller state and the semantics of its assembly instructions. An excerpt from the RTL-DSL specification for the ATmega16 is depicted in Listing 2.1. The specification begins with a description of the microcontroller's memory layout. This section includes the number of general purpose registers, a detailed description of the special purpose registers and the size of the SRAM. Additionally it is possible to declare mappings which assign a symbolic name to two successive registers. The ATmega16 for example contains a symbolic register X, which allows to access the GPR_{26} and GPR_{27} as a single value. The memory layout information is usually available from the vendor's description and can be taken from there. In addition to the memory layout information it is necessary to provide details about the peripheral subsystems, for example the interrupt system or informations about the timers. The specification file is concluded by a description of the instructions semantics. The Register Transfer Language is employed for this purpose.

2.5 Microcontroller Environment

A microcontroller does not work in isolation but rather interacts with a given environment. Environment interactions can manifest themselves for example as pressing of a push-button which is connected to a port pin and which results in an external interrupt being triggered. More complex interactions are possible by employing the functionality of the various peripheral subsystems, such as the *Serial Peripheral Interface*, the *USART* or the *Two-wire Serial Interface*. Independent of the complexity of a particular environment interaction, every interaction can be observed at a specific memory location. When an application program for

```
GPRS
  32
SPRS
  _TWBR    "Two-wire Serial interface Bit Rate Register";
  _TWSR    _TWS7 [...] __    _TWPS1 _TWPS0 248;
  _TWAR          _TWA6 [...] _TWA0  _TWGCE 254;
  io _TWDR "Two-wire Serial Interface Data Register"  255;
  i _ADCL  "ADC Data Register Low Byte";
  i _ADCH  "ADC Data Register High Byte";
  [...]
SRAM
  1024
MAPPINGS
  _X 26 27  "X register";
  _Y 28 29  "Y register";
  _Z 30 31  "Z register";
  _SP 93 94 "stack pointer"

ADD Rd, Rr;
  Rd =. Rd + Rr;
  C = Rd(7) & Rr(7) | Rr(7) & !R(7) | !R(7) & Rd(7);
  Z = !R;
  N = R(7);
  V = Rd(7) & Rr(7) & !R(7) | !Rd(7) & !Rr(7) & R(7);
  S = N ^ V;
  H = Rd(3) & Rr(3) | Rr(3) & !R(3) | !R(3) & Rd(3)

BREQ K;
  if (Z == 1) then PC = PC + K + 1
```

Listing 2.1: Excerpts from the RTL-DSL specification for the ATmega16

example reads the value of an port pin connected to a push-button, the value solely depends on the button's state. To achieve a deterministic program execution it is not sufficient to model the microcontroller and to keep track of its state. For a deterministic execution it is also necessary to provide a model of the environment and to keep track of the environment's state. In case a model of the environment and the contained devices is not provided, the state of the environment has to be regarded as non-deterministic. This means that when requesting data from the environment, every possible value has to be assumed. Although there is only a single execution path in reality, there may exist multiple execution paths when simulating the microcontroller application, due to the non-determinism introduced by the missing environment state. On the level of the NIPS VM the existence of multiple execution paths can be recognized by the fact that some NIPS states have multiple successors.

In the existing toolchain the interaction between microcontroller and environment is covered by additional NIPS bytecode which is executed on access to specified memory locations. A first but minor purpose of these additional instructions is to figure out whether a memory access infers an environment interaction. If an environment access is detected the code can either use encoded information about the environment to return a single data value or otherwise rely on the NIPS local non-determinism to create successor states for all possible data values. The information which memory locations need further specification is stated in the memory layout section of the RTL-DSL specification. Prefixing a register byte or bit with the *i* modifier indicates the necessary execution of abstraction code on read access. The code for a read access has to write the value which corresponds to the value of the memory location to the temporary NIPS stack. The *o* modifier can be used in case of a write access. In this case the value to be written is already on the NIPS stack, and it is required by the abstraction code that this value is deleted after execution. It is also possible to use both modifiers at the same time. The abstraction code is not located in the RTL-DSL specification but in a separate file.

As an example to demonstrate the use of abstraction code for modeling environment interactions we will refer to the PINA register. The execution of abstraction code is required for both read and write access to each port pin. On read access the abstraction code has to check whether the accessed pin is configured as input or output pin by inspecting the corresponding bit in the *Data Direction Register* DDRA.

```
SPRS
  ...
  _PINA io _PINA7 io _PINA6 io _PINA5 io _PINA4 ...
  ...
```

In case the port pin is set as input the abstraction code is responsible for representing the state of the external device and the current value has to be written on the stack. If this information is not available the abstraction code can use local non-determinism to create two successors for the possible values 0 and 1. If the port pin is configured as output, the value to be written is deterministic and can be read from the corresponding bit on PORTA.

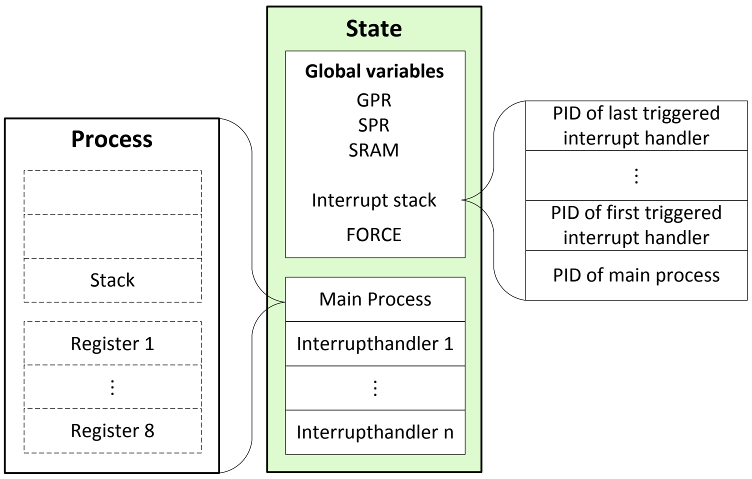


Figure 2.4: Microcontroller state in the NIPS VM

2.6 Microcontroller state in the NIPS VM

At this point we have examined the toolchain for employing the NIPS VM as instruction-set simulator and discovered how to specify the microcontroller, the associated instruction-set and the given environment. The purpose of this section is to explore how the state of the microcontroller can be captured within the NIPS VM. As the microcontroller's state is completely specified by the state of its memories a mapping of the specified memory layout to NIPS memory space is required. There are actually two possible locations for storing the memory layout, namely the global variable space and the local variable space of a process. Considering the design decision to model the main program and the interrupt handlers in separate processes the decision was in favor of the global variable space to allow an easy access by all processes. As indicated by Rohrbach [Roh06], modelling the main program and the interrupt handlers in separate processes has advantages with respect to static code analysis.

Figure 2.4 gives an overview of this situation and depicts how a microcontroller state is mapped to a NIPS state. The main program and interrupt handlers are mapped to NIPS processes, whereas the general purpose registers, special purpose registers and the SRAM are stored in the global variable section of the NIPS state. Furthermore, the global variable space contains an interrupt stack to keep track of the currently active interrupt handlers. This information is necessary to decide which process (main program or interrupt handler) is privileged to execute a process step. The *FORCE*-Byte is used to make sure that once an interrupt handler returns, the interrupted process is able to execute at least one instruction before another interrupt may occur.

At this point we would like conclude this chapter and continue with an introduction of the Eclipse platform. More detailed information about the NIPS VM and the MCESS project is available in [WS05] and [Roh06], respectively.

3 Eclipse

An important aim of this project is to integrate embedded systems debugging closely into the development process. An integrated development environment is a natural fit for this purpose. As indicated in the introductory we have found all our requirements fulfilled by the Eclipse platform. This chapter gives an overview of the Eclipse platform and discusses aspects required for custom tool integration.

3.1 History and Overview

The development of Eclipse started in November 1998 at OTI and IBM. Besides the need for a tool platform to consolidate existing products build at various labs, it was IBM's aim to stop the fragmentation of Java tools in the competitive ecosystem within Java space, and advance with respect to Microsoft. In November 2001, the Eclipse consortium consisting of nine initial members was established and Eclipse 1.0 was released as open-source platform under the Eclipse Public License. The portal *eclipse.org* was created at the same time to serve as communication medium for developers all over the world dedicated to Eclipse development. Nevertheless, with the release of version 2.1 in March 2001, public perceived Eclipse as an IBM controlled effort and major vendors were reluctant to make strategic commitment. This situation changed when in February 2004 a non-profit organization named Eclipse Foundation was created to make Eclipse vendor independent. This strategy proved to be very successful and a dramatic growth in members at all levels combined with a deeper commitment has been the result. Currently, Eclipse has reached version 3.5 and the Eclipse Foundation hosts over 50 projects which build on or extend the Eclipse platform.

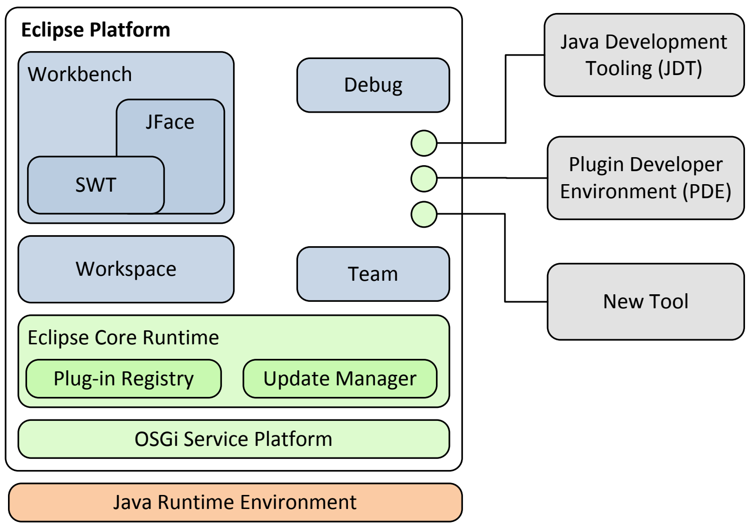


Figure 3.1: Eclipse Architecture

3.2 Architecture

One of the most important design goals when building the Eclipse platform was to allow a seamless integration of tools to support the full software development life-cycle. This aim led to an architecture that can be characterized best as plugin architecture. The Eclipse platform is basically just a small core which provides functionality for discovering, integrating and running modules called *plugins*. While the functionality provided by the platform is very generic, *plugins* provide additional functionality and allow to focus the generic functionality to something specific. The Java development tools (JDT) for example are a collection of plugins that add the capabilities of a full-featured Java IDE to the platform. The major components of the Eclipse platform that help to achieve this design are pictured in Figure 3.1. In order to get a better understanding of how this architecture has been realized from a technical point of view our discusssion will begin with the first component above the Java Runtime Environment, namely the OSGi Service Platform.

3.2.1 OSGi Service Platform

The OSGi Alliance is an industry consortium founded in 1999 which provides the OSGi Service Platform specifications. This set of specifications essentially defines an open framework for building component-based and service-oriented applications with Java. The OSGi framework is divided in a number of layers whereas the service registry, the life cycle layer and the module layer are the most important ones. These layers build the environment for the OSGi components, the so-called *bundles* in OSGi terminology. A *bundle* is the smallest logical unit in the OSGi framework and consists of Java code in a JAR (Java archive) library, a manifest file for configuration information and most often also additional resources such as images and help texts. When exploring different means to evolve the Eclipse 2.1 architecture it was discovered that the OSGi framework not only fulfills all requirements posed by the Eclipse platform, but also provides a number of advantages compared to the proprietary format used in version 2.1. Eclipse 3.0 adopted the OSGi framework as a foundation, evolving and improving its own architecture accordingly. It is necessary to point out, that the OSGi framework is solely a specification. The project that represents the framework implementation found in Eclipse has become known as the Equinox project. The following sections will introduce the individual layers of the OSGi framework and also mention differences between the specification and the Equinox implementation.

Module layer

The module layer is responsible for the static structure of the OSGi framework and defines a bundle as unit of modularization. The module layer introduces the possibility to explicitly model bundle dependencies on a Java package level. A bundle explicitly imports the Java packages that it needs but does not itself define. Conversely, a bundle may export some Java packages that it defines. The dependency information is stored in the bundle manifest.

Life cycle layer

While the module layer provides the static aspects of the OSGi framework, it is the responsibility of the life cycle layer to introduce dynamic aspects. The life cycle layer adds the functionality to dynamically install, start, stop, update and uninstall bundles. Figure 3.2 depicts all possible states a bundle can acquire during its life-cycle in the OSGi framework. After deployment a bundle is in the *Installed* state and represents a candidate for execution. Before the execution of the bundle can be started, it becomes necessary to resolve bundle dependencies. Only if all dependencies can be resolved, it becomes possible to start the bundle. A started bundle transitions through the *Started* state and reaches the *Active* state after initialization. Active bundles are fully operational and will move again to the *Resolved* state when stopped.

Service Registry

The two previous sections explained the static and dynamic structure of OSGi bundles. Bundles have been introduced as unit for modularization that add custom functionality to a system, but work quite independently of other bundles. Typically, the situation is quite different and bundles rely on other bundles to achieve their purpose. Bundle dependencies can be specified at the Java package level, however, creating collaborative models within this environment poses difficulties. The concept of services and the service registry component in the OSGi framework leverage the design of bundles that collaborate. A service within OSGi is a plain Java object and is identified through its Java interface. A bundle can use the service registry to publish created service objects, and also find and bind services registered by other bundles. The mechanism is highly dynamic and an active bundle is able to register or

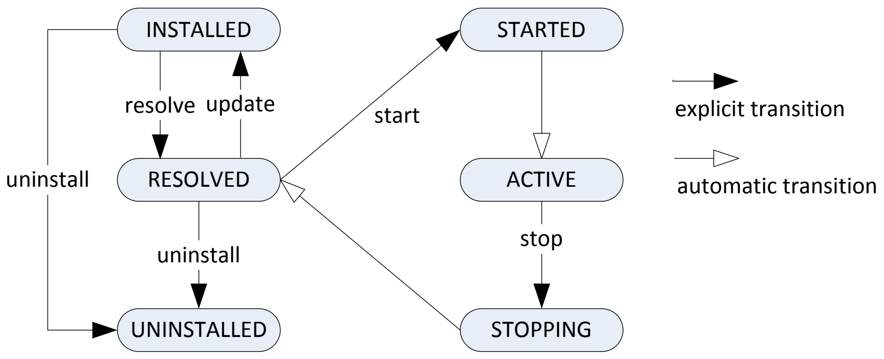


Figure 3.2: OSGi bundle life-cycle

unregister services at any time.

3.2.2 Extension Concept and Extension Registry

Although Equinox provides a full implementation of the service layer specification, currently almost all Eclipse plugins realize collaboration with the complementary concepts of extension points and extensions. These concepts have been a feature of Eclipse from its beginning and introduce tension as both collaboration models overlap. At the same time the concepts of services and extensions are different enough to make a merge impractical.

Extension points can be used by a plugin to allow other plugins to extend or customize portions of its functionality. The extension point declares a contract, typically a combination of XML markup and Java interfaces, that extensions must conform to. Plugins that want to connect to a particular extension point must implement that contract in their extension. The plugin which is extended has no information about the connected plugin apart from the data specified in the extension point contract. The information how a plugin extends the platform, what extensions it publishes itself, and how it implements its functionality is stored in the plugin.xml manifest file.

The extension registry is an important component within the Eclipse Core Runtime and manages plugin contributions. After a bundle is installed and enters the *Resolved* state, it is queried for its contributions to the extension registry. The extension registry is used by plugins to find out what extensions are provided for its extension points. An important result of this design is the fact that it becomes possible to obtain manifest information from the extension registry without activating the contributing plugin or loading any of its code.

3.2.3 Workbench

The Workbench is a fundamental component of the Eclipse architecture and not only provides the Eclipse UI personality but also supplies the structures for the interaction of the user and the tools [RW04]. The workbench is synonymous with the main window the user sees when the platform is running. The screenshot of the ESD debugger as depicted in Figure 1.4 provides an example of the Eclipse workbench. The workbench consists of a few basic building blocks, being Views, Editors and Perspectives which we will discuss in the following.

Editors allow the user to open, edit and save documents or input objects adn follow an open-save-close life-cycle much like system tools. Active editors can contribute actions to the workbench menus or tool bar. While a standard text editor is provided by the platform, more specific editors are contributed by other plugins.

Views provide information about an object the user is working with in the workbench. They often work in close combination with an editor to show additional information, but at the same time follow a simple life-cycle, i.e., modifications are generally made persistent immediately and the changes are reflected immediately in other related parts of the UI.

Perspectives are a workbench concept that determine the visibility and layout of editors, views and also menus and toolbars. Typically, there exist multiple perspectives in a workbench window, however, only one is active at a time. The user can switch perspectives to work on a different task and also customize perspectives to better suit a particular task.

The workbench API and implementation are build from two general UI toolkits, namely the Standard Widget Toolkit (SWT) and JFace. Both toolkits have been developed to support the Eclipse platform UI implementation, but can also be used outside the Eclipse framework. The importance of SWT manifests itself in the fact that in order to achieve a tight workbench integration of custom UI components, i.e. Views and Editors, these must be build by using SWT. To understand the contribution of SWT it is necessary to explain that a perennial issue in widget toolkit design is the tension between portable toolkits and native window integration. When SWT was developed in 2001 there already existed two Java UI toolkits: the Abstract Window Toolkit (AWT) and Swing.

Both toolkits both follow a rather extreme approach concerning the mentioned issue. The AWT provides low-level widgets such as list, fields and buttons but no high-level widgets such as trees or tables. AWT widgets are implemented directly with native widgets on the underlaying window system. Building a UI on top of AWT means programming for the least common denominator of all supported OS window systems. The Swing toolkit addresses this problem by emulating widgets like trees, table, and rich text. However, the emulated widgets invariably lag behind the look and feel of the native widgets. It is therefore difficult to compete with applications developed specifically for a particular native window system.

The Standard Widget Toolkit tries to find a better balance between the former approaches and relies on the emulation of widgets only in rare occasions. The developer is supplied with a common API for UI development on different window systems, and is able to achieve a native look and feel to the greatest extend possible. Despite the advantage of a consistent programming model in different environments, relying solely on SWT might be tedious at times. This is where JFace starts to shine, as it was developed as additional layer on top of SWT and provides helper classes to develop more complex UI features. Developing UI components with JFace allows a developer to make a clean separation between the data representation and the data itself, as JFace introduces the Model-View-Controller paradigm.

3.2.4 Plugin Development Environment

Another reason for the success of the Eclipse platform is its good plugin development support that is provided by the Plugin Development Environment (PDE). The PDE provides a manifest editor which simplifies extension point creation and usage. Developer documentation regarding individual extension points can be written and read in the PDE as well. A check of the static plugin structure is carried out at compile time. When developing with PDE, the Eclipse platform is not only used for the development of plugins but also for plugin execution. This requires a distinction between the host instance and the runtime instance, whereas both instances can have a different configuration and particularly host different plugins.

3.3 Debugging Support

In the first part of this chapter, we gave a general overview of the Eclipse platform and provided details concerning its architecture. This section leaves these general observations behind and concentrates on the facilities and mechanisms provided by Eclipse that support the integration of custom debuggers. The platform's debugging support goes beyond the Java-centric capabilities and is claimed to be language independent. All work regarding Eclipse's debugging support is contained in the Debug Project [Ecl10a]. The debug project itself, consists of two sub-projects: Platform Debug and JDT Debug. The Platform Debug project is concerned with the provision of language independent debug facilities, whereas the JDT Debug relies on the platform debug project to provide Java debugging capabilities within Eclipse.

The implementation of a debugger does not only require work on the target side. A substantial amount of resources is also necessary for the provision of the debugger's user interface and mechanisms that allow user interaction with the debuggee process. The availability of a generic debugger implementation which is easily customizable to support a specific environment and which comprises both debug target and frontend is ultimately what developers of debuggers are looking for. But the desired debug target implementation is not only dependent on the hardware architecture chosen for executing the application but also on other influence factors, such as an existing operating system, an interpretation environment or virtual machine, or employed compilers. This fact makes the existence of a generic debug target implementation which actually saves development time and is not just a completely abstract construct very unlikely, probably even impossible. The situation is different though for the graphical user interface: On this level debuggers have more in common and the necessity to display program context information and to provide means to control the debuggee process results in a similar set of views and control bars across different debug targets.

This observation has been and still is the motivation for the Platform Debug project and its main goal to provide a language and architecture independent debugger user interface. The project's contribution is referred to as debug framework. An important part of this framework is a generic debugger user interface that can be customized and enhanced with features specific to a particular debugger. The user interface is realized as separate perspective, namely the debug perspective, and includes views to display program context information and a toolbar and menu bar for execution control. The user interface is complemented by an abstract debug model which is realized as collection of Java interfaces and serves as presentation model for the individual user interface elements. From a high-level view the debug model is the intermediary between user interface and debug target and makes them independent from each other. Although the debug model makes no statement about the debug target's technical realisation, it does make a statement about certain artifacts that have to be known by the program under debug. These debug artifacts include threads, stack frames, variables, and breakpoints. The complete set of artifacts in total makes up the debug model and defines an abstraction level which corresponds to imperative programming languages such as Java, C or Python. By this means the debug framework simplifies the development of a debugger's user interface in case of imperative programming languages. For programming languages that fall out of the imperative programming paradigm, such as functional or logic programming languages, the prospective benefit by using the framework seems doubtful.

4 Embedded Systems Debugger

At this point we would like to introduce the established Embedded Systems Debugger (ESD) and start out with an overview of its architecture. In the subsequent examination we will then base our discussion on the two fundamental capabilities of a debugger: the ability to control the program under debug and the provision of program context information.

4.1 Architecture

During the initial design phase of this project we have been concerned with the development of a software architecture for our prospective debugger. Before we discuss related work on debug architectures and subsequently present our results, it is necessary to go one step backward and shed more light on the foregone requirement analysis. We have already discussed two major requirements in the introductory chapter, namely the debugger's integration into a development environment and the employment of an instruction-set simulator. By means of the Eclipse platform and the NIPS VM we have introduced and chosen two elements which are well suited to fulfill these requirements. At this point we would like to introduce two further requirements with respect to the simulation component:

The instruction-set simulator is an important element of our architecture and has a big influence on the debugger's performance due to its responsibility to run the program under debug. A performance evaluation between different simulators is a reasonable measure to optimize the debugger's over-all performance at a later stage. This step requires careful planning to make sure that the debugger is not tied to a particular simulator, but is build on a modular architecture that allows an easy integration of different simulators. Missing support for a particular microcontroller or discontinued development effort by the simulator's vendor are further arguments which may introduce the necessity to employ a different simulator. In other words, we require the prospective debug architecture to be independent from the employed instruction-set simulator.

A further requirement becomes apparent when we take into account that the NIPS VM has primarily been chosen because of its ability to verify the system's correctness by means of the model checking technique (see Chapter 5 for details). Whereas the performance and space resources of a modern desktop computer are sufficient to simulate the microcontroller's execution, this fact does not generally hold for the mentioned verification task. Executing the NIPS VM on more powerful hardware, probably a computer cluster, might become necessary. Our debugger must consider this fact by allowing to execute the instruction-set simulator on a remote system.

Before we present our established debug architecture we shall consider existing work on debug architectures and discuss whether these approaches fulfill our requirements and can be reused.

Related work

Published work on debug architectures is limited to a few resources only. The first work we would like to review has been made by Rosenberg and is described in chapter two of [Ros96]. The work by Aggarwal [AK02] follows an almost identical direction and the following discussion therefore applies to both approaches. Rosenberg's and Aggarwal's work focuses on compiled environments, that is, applications which have been compiled to native code and directly execute on the target CPU. The described architecture is centric to the hardware and operating system the target application runs on and the resulting debugger strongly depends on the debugging support provided by those artifacts. Hardware, operating system and the operating system's debug support make up the three bottom layers of the final architecture. These layers are complemented by the debug kernel which controls the process under debug with help of the operating system and extracts debug information from the executable file. The final layer consists of various user interface components that present different views on the program being debugged and provide the user's handle to the debugging task. The architecture implicitly assumes that the debugger as well as the target application run on the same system and makes no statement whether a remote execution is possible. Considering the fact that our focus is on applications which are firstly compiled to an intermediate form and are then interpreted (a mixed-mode environment so to speak), the sketched solution does not apply to our circumstances and requirements.

Another source of information regarding debug architectures exists in form of the Java Platform Debug Architecture (JPDA) [Ora10] which is a multi-tiered architecture to debug Java applications. JPDA has been a minor inspiration for this work, as it makes the development of a debugger platform-independent and also allows to debug applications running on remote systems. The main contribution of JPDA is the provision of three interfaces, namely the Java Virtual Machine Tools Interface (JVMTI), the Java Debug Interface (JDI) and the Java Debug Wire Protocol (JDWP). JVTMI specifies the functionality a virtual machine, or to be more precise, a Java Virtual Machine must provide for debugging, whereas JWDP defines the format and semantics of messages transfered between the process being debugged and the debugger. JDI defines a high-level view of the program being debugged and simplifies the development of debuggers. What makes JPDA important in the context of this work is its clear separation between the component which executes the application to be debugged and the component which is concerned with the user interface.

Embedded Systems Debugger

Under consideration of the given requirements and related work on debug architectures we created a two-tier, a so-called client-server architecture, as foundation for our desired embedded systems debugger. In the context of this work we do not use the term client and server but refer to both components by the name debugger and debuggee. The architecture is depicted in Figure 4.1 and consists of the mentioned debuggee and debugger component and furthermore a communication channel which connects both components. In the following, we will present these components in more detail.

Debuggee

The debuggee (or debug target) is responsible for executing the target application, that is, the application the user wishes to debug. This is achieved by employment of an instruction-set simulator, more precisely the chosen NIPS VM. As mentioned in section 2.1 the NIPS VM operates on a bytecode file that corresponds to the target application and which has to be created from the program sources in a preliminary step. Access to the virtual machine is provided by means of a simple API, however, implementing the functionality required for debugging on top of this interface will become a tedious task. For this reason we created an embedding component, namely the trace manager, which controls the virtual machine and builds an intermediary layer between backend and virtual machine. The backend is responsible for processing requests from the debugger and for communicating the response to these requests back to the debugger. The backend relies on the trace manager for those requests that require a state change, or alternatively employs the state analyser when the provision of program context information is of concern. The state analyser extracts program context information from the current program state and for this task requires debug information captured within the bytecode as well as the original binary executable.

Communication Channel

The communication channel establishes the connection between debugger and debuggee and is realized by two separate channels, namely a command and an event channel. Commands are employed by the debugger to control the application under debug and to retrieve information about its execution state. Communication by means of the command channel is carried out in a synchronous manner: the debugger sends a command to the debuggee and then waits to receive a reply message. This is contrasted by the event channel, that is used by the

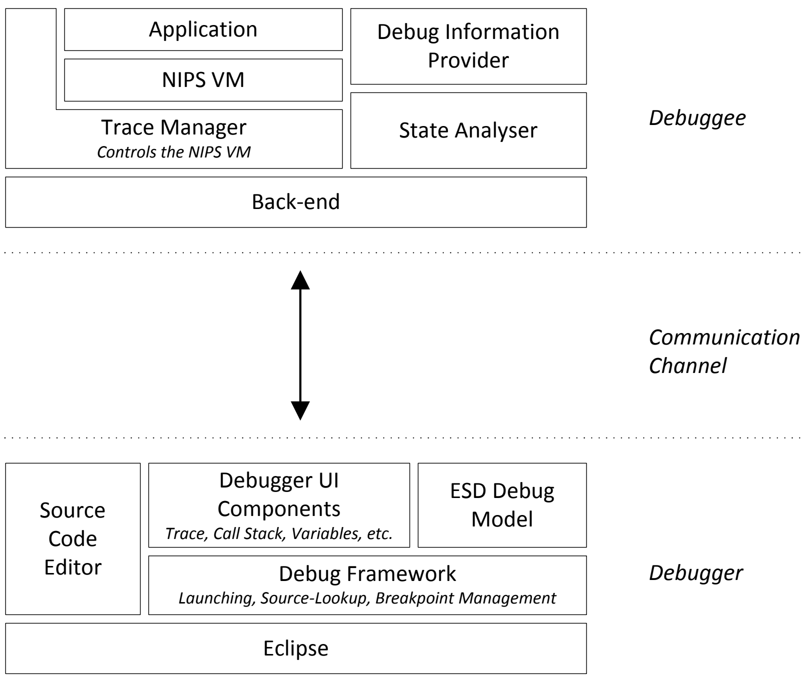


Figure 4.1: Embedded Systems Debugger Architecture

debuggee to inform the debugger about state changes. Events are sent in an asynchronous fashion. A complete list of existing commands and events and an example communication scenario is given in Section 7.2.

Debugger

The debugger is a handle to the application under debug and is integrated within the Eclipse platform. The debugger is not responsible for executing the application the user wishes to debug but delegates this task to the debuggee component. The launch, that is, the creation of the debuggee is therefore one of its first responsibilities. The debugger provides various user interface components that allow to control the debugging process and provide program context information such as the function call stack, variable and register values or the micro-controller's memory state. The user interface components rely on the ESD Debug Model for the provision of all necessary information with respect to the program under debug.

ESD Debug Model

The ESD Debug model is an implementation of the abstract debug model as part of the platform's debug framework and defines a view of the program under debug. Our debug model furthermore extends the abstract model to support the retrieval and display of disassembly information and to cater for debug targets that allow to record the application's execution during debugging. The most important elements of our debug model are depicted in Figure 4.2. The

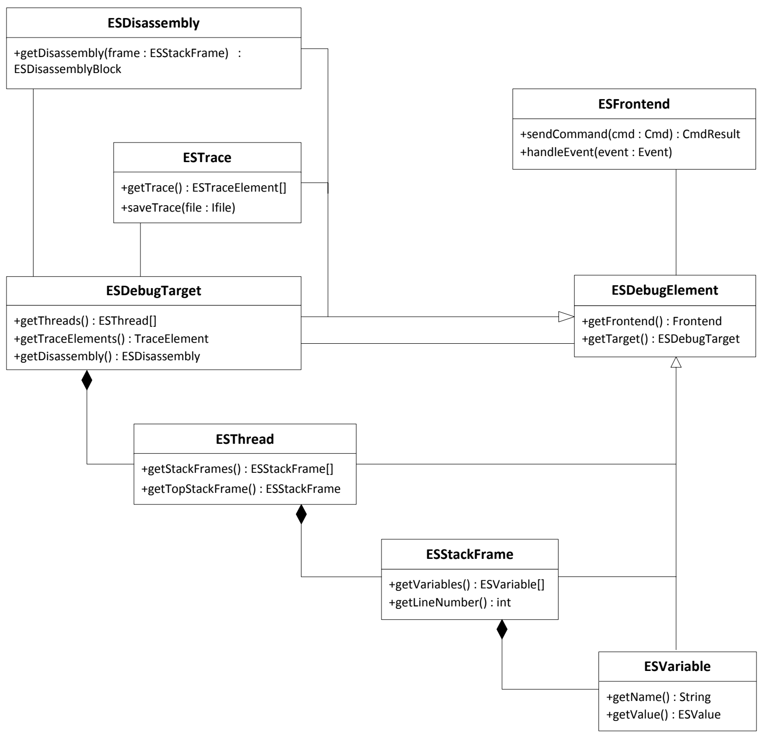

ESDisassembly
+getDisassembly(frame : ESStackFrame) : ESDisassemblyBlock
ESFrontend
+sendCommand(cmd : Cmd) : CmdResult
+handleEvent(event : Event)
ESTrace
+getTrace() : ESTraceElement[]
+saveTrace(file : Ifile)
ESDebugTarget
+getThreads() : ESThread[]
+getTraceElements() : TraceElement
+getDisassembly() : ESDisassembly
ESDebugElement
+getFrontend() : Frontend
+getTarget() : ESDebugTarget
ESThread
+getStackFrames() : ESStackFrame[]
+getTopStackFrame() : ESStackFrame
ESStackFrame
+getVariables() : ESVariable[]
+getLineNumber() : int
ESVariable
+getName() : String
+getValue() : ESValue

Figure 4.2: Excerpt from the ESD Debug Model

central element is the *ESDebugTarget* which represents the target application being executed in debug mode. The *ESDebugTarget* consists of a set of *ESThread* objects, which are representations for parallel executions contexts. Single-threaded target architectures, such as the ATmega16 are represented by a debug target which contains a single thread object only. *ESThread* objects have a name attribute and contain a number of stack frames, more precisely one stack frame for every currently active function or interrupt handler. Stack frames are represented by *ESStackFrame* objects and feature a name, for example a function name including argument names or the name of an interrupt handler. Associated with each stack frame object is also a set of variables, that is *ESVariable* objects, which are visible within the particular stack frame. The mentioned elements are complemented by the *ESDisassembly* and *ESTrace* element: Both elements are associated with a debug target object and whereas the first element is responsible for the retrieval of disassembly information, the second element provides support for targets that allow to record the application's execution.

4.2 Controlling Execution

A fundamental requirement of a debugger is to control the execution of the application being debugged. This section gives an overview of the functionality provided by the ESD debugger regarding this aspect.

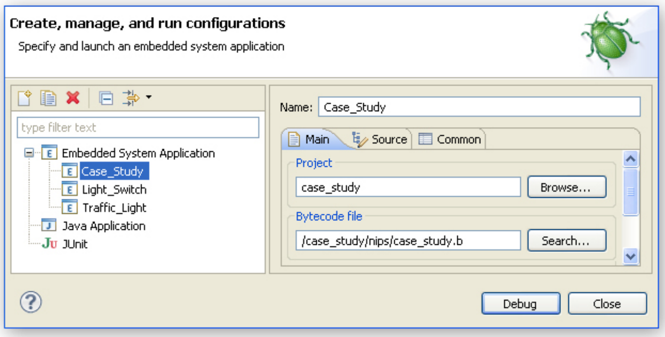


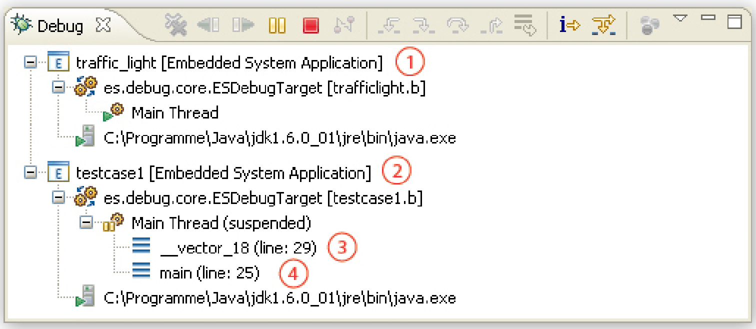
Figure 4.3: Launch Dialog

4.2.1 Launching

A prerequisite for execution control is the instantiation of the debuggee component. The debuggee is not only responsible for controlling the simulator, hence the executed program, but makes execution possible in the first place. One of the first tasks of the debuggee is to initialize the virtual machine and load the bytecode program derived from the original source code to prepare for the upcoming execution. A further prerequisite consist of creating the debugger component, more precisely the instantiation of an *ESDebugTarget* object, in order to represent the started debuggee within the Eclipse user interface. The mentioned activities characterise the start process of our debugger. In the following we would like to examine this process more closely, both from the user and from the technical perspective.

Eclipse refers to the activity of starting a new process by the term *launching* and provides a collection of APIs, internally known as the launch framework, to support implementors with the user interface as well as the behavioral aspects of this task. Although the framework is part of the platform debug project [Ecl10a] the provided launch capabilities are not restricted to debugging but apply to regular execution or profiling as well. A certain program type, such as a Java program or an embedded systems program, that should be launchable within the platform is described by a *LaunchConfigurationType*. A launch configuration type determines what domain-specific information is necessary for launching and also specifies the behavioral aspects of launching by reference to a launch delegate. The domain-specific attributes necessary for performing a specific launch are actually not explicitly described by a launch configuration type, but are instead assigned to individual launch configurations.

Before we continue with the behavioral aspects of launching, let us firstly examine the launching-related UI. Most important in this context is the launch dialog, that is provided by the framework and allows users to create, manage, launch and delete launch configurations of any type. The launch dialog, a screenshot is depicted in Figure 4.3, is horizontally divided into two parts. Its left side consists of a tree showing all currently defined launch configurations, grouped by configuration type. For a better identification of the different configuration types it is possible to assign an icon by registering with the *launchConfigurationTypeImages* extension point. The dialog's right side contains a tabbed folder which allows to insert or edit



```
traffic_light [Embedded System Application]
  es.debug.core.ESDebugTarget [trafficlight.b]
    Main Thread
  C:\Programme\Java\jdk1.6.0_01\jre\bin\java.exe
testcase1 [Embedded System Application]
  es.debug.core.ESDebugTarget [testcase1.b]
    Main Thread (suspended)
      __vector_18 (line: 29)
      main (line: 25)
  C:\Programme\Java\jdk1.6.0_01\jre\bin\java.exe
```

Figure 4.4: Debug View

the domain-specific information of the currently selected configuration, that is, the configuration selected on the left. We contributed to the *launchConfigurationTabGroups* extension point to specify the content of the tabbed folder when an embedded system application is selected.

The behavioural aspects of our contributed launch configuration are specified within the *ESLaunchDelegate* class which implements the *ILaunchConfigurationDelegate* interface and overrides the required *launch()* method. The job of the launch method is to extract the domain-specific attributes from the passed launch configuration and then act upon this information. In our case, the most important information to extract from the passed configuration object is the path and name of the NIPS bytecode file which is input to the simulator. In order to employ replay debugging (see Section 4.2.3 for details) further information may include name and path of a previously saved trace file. Acting upon the information contained within the launch configuration generally results in the creation of system processes and the additional creation of debug targets for those applications to be launched in debug mode. A subsequent objective of the launch method is to register the created processes and/or debug targets with an *ILaunch* object, which is passed as further argument to the launch method. The launch object represents the launched session and is used as handle on the created artifacts. The user's handle to the launched sessions is established by a user interface component referred to as Debug View. A screenshot of the debug view showing two launched embedded systems applications (highlighted as 1 and 2) is given in Figure 4.4. Items 3 and 4 are referenced in a later section of this chapter (see Section 4.3.2).

Once the launching process is completed, the debuggee is executing the considered program. At this point we shall consider what functionality exists to actually control the program execution.

4.2.2 Breakpoint Handling

Breakpoints are the basic mechanism used to control the execution of the target program and provide a way to suspend program execution at a user-specifiable location or upon a specified condition. Being able to suspend program execution at user-specifiable conditions is a prerequisite for inspecting the program context. Breakpoints exists in various forms, including Line breakpoints, Run-to-Line breakpoints and Watchpoints. Line breakpoints are capable of suspending execution when a specified source code line is reached. They can only be set on the executable lines of code, whereas comments, declarations of variables and methods, and empty lines are not valid locations. Run-to-Line is a breakpoint concept which is similar to line breakpoints as it allows to advance execution to a specified line of code. But in contrast to a permanent line breakpoint, which requires an explicit setting and unsetting by the user, a Run-to-Line breakpoint is temporary. Once execution is suspended temporary breakpoints are removed. Watchpoints, or sometimes called data breakpoints, can be used to suspend execution whenever a specific location or region in memory is modified, without having to predict a particular place where this may happen.

The first step toward understanding how the ESD debugger handles breakpoints is to explore how breakpoints are supported and represented by the Eclipse platform. The actual breakpoint capabilities are not provided by the platform, but instead by the underlying debug architecture. For a complete overview it is therefore necessary to discover how breakpoints are implemented within the debuggee component.

To utilize the platform's breakpoint facilities the developer needs to contribute an extension to the *org.eclipse.debug.core.breakpoint* extension point for each breakpoint type supported by the underlying debug architecture. Our provided extension to support line breakpoints is shown as an example in Appendix A.1.2. Breakpoints of the contributed type are represented as instances of a class implementing *IBreakpoint*. The full name of the particular class is a mandatory attribute of the extension, namely the *class* attribute. The task of the specified class is to store the information required to install a breakpoint in a specific architecture. From a technical standpoint this task is realized by having each breakpoint associate a resource marker. Resource markers allow to associate information about a resource in form of named attributes. By this means, the breakpoint implementation can make use of all the existing marker functions, such as persistence and display in editors. Knowledge of this implementation detail is also necessary at the specification level. When contributing a new breakpoint type the developer is required to specify an associate marker type by means of the *markerType* attribute. The associated marker type has to be contributed by another extension to the *org.eclipse.core.resources.markers* extension point. Again, the extension of the marker type associated with our breakpoint type contribution is depicted in Appendix A.1.3. At this point we have discovered the platform's mechanism for breakpoint representation. In the following we will investigate the question of how breakpoints are installed in the debuggee process.

The debug target, being the root element of the debug element hierarchy, is responsible for breakpoint installment. For this purpose the debug target works in close collaboration with the breakpoint manager, which is the central authority over all breakpoints within the workspace. The breakpoint manager is informed when breakpoints are added, removed or changed in the workspace and passes this information to registered listeners, such as the debug target. When the debug target receives change notification from the breakpoint manager it has to decide whether it is responsible for the breakpoint and possibly carry out necessary actions. Besides reacting to change notification, the debug target has to take care of all relevant breakpoints existing prior to launch time. For this reason, the debug target suspends the debuggee process right after startup before it processes any instruction. It then queries the breakpoint manager for all relevant breakpoints and resumes the debuggee process only after these initial (deferred) breakpoints have been installed.

In our case, the logic for breakpoint installment is not kept in the debug target but rather within each *IBreakpoint* implementation representing a breakpoint type. When a breakpoint is set, the corresponding *IBreakpoint* instance registers as debug event listener with the frontend and issues a set-breakpoint command to the debuggee process. The breakpoint attributes necessary for installment are retrieved from the associated resource marker and passed as part of the set-breakpoint command.

The way the debuggee process or the backend implements breakpoints depends on the breakpoint type. For line breakpoints the backend first of all utilizes the debug information component to resolve the instruction address corresponding to the file name and line number of the breakpoint. The backend manages line breakpoints by a map-like data structure and after resolution a new entry is created. The key of this entry consists of the instruction address, whereas the value is represented by an object wrapping the breakpoint's attributes. When the debuggee process is resumed, the backend is able to determine if a breakpoint was hit by checking whether the current instruction address exists in the map.

When a breakpoint is hit the debuggee backend notifies the debugger, or to be more precise the frontend, by sending a corresponding breakpoint event. The frontend passes the received information to the registered debug event listeners, including the installed breakpoints. A particular breakpoint instance will react to such a notification by validating if its marker's attribute values correspond to the attributes values found in the notification. If this is the case, the respective breakpoint will update its hit count and notify the thread.

4.2.3 Resuming Execution

Knowing how to suspend program execution under user-defined conditions via breakpoints is important due to the possibility to inspect program context information during suspension. Nevertheless, chances are slim that the information retrieved at the first suspension is sufficient to discover the causes of a failure or to gain a better understanding of the software system. The ability to resume execution is therefore equally important as suspending because it is a necessity to inspect context information at different moments in time. The ESD debugger features different possibilities to resume execution including functionality for forward but also reverse continuation. User access to this functionality is provided by the debug view's toolbar which is depicted in Figure 4.5.

Concerning forward execution, the user is able to carry out an unconditional resume which continues program execution until either another breakpoint is hit or the program terminates.

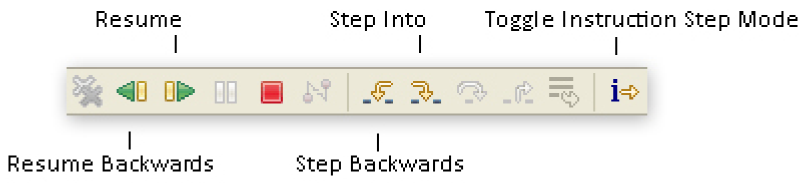


Figure 4.5: The debug view's toolbar for execution control

Another possibility is provided by a source-level step which allows to resume program execution until the next source line is reached. Our implemented step functionality obeys function calls found within the current source line. In this case the debugger steps into the function and suspends execution at the first executable source line found. Sometimes the granularity provided by source-level stepping is too coarse and the user wishes to advance program execution at the level of single machine instructions. We have provided a toggle button, which is part of the mentioned toolbar, to switch between source- and instruction-level stepping. When instruction-stepping is enabled the step action will advance program execution not until the next source-line but instead until the next machine-level instruction is reached. Whereas machine-level steps are atomic and cannot be interrupted, this fact does not hold for source-level steps. In case of a source-level step, execution may be suspended sooner due to an instruction-level breakpoint.

Reverse execution

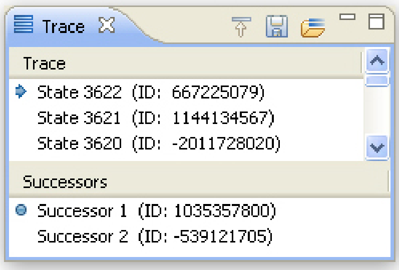
Reverse execution complements the mentioned possibilities to resume execution by allowing to run backwards and return to previous states in the program's execution history. Many software defects are hard to detect as they do not cause failures immediately but show their effect later in a program execution. As a consequence software developer often miss the occurrence of defects during a debugging session and let execution advance to a point where the failure is visible but its cause cannot be inferred anymore. Developers commonly react to this problem by restarting the program and then try to suspend execution earlier to find out what causes the failure. The approach is problematic as difficult to find defects often arise from timing problems or due to special conditions in the microcontroller's environment and the failure is typically difficult to reproduce. Even if it is possible to reproduce the failure in following executions it may still take several restarts to locate its cause. In either case a severe impact on developing time results as consequence. The ability to continue execution backwards is an often requested feature in debuggers as it provides a better solution to the problem. Instead of restarting the debuggee process and being forced to re-execute non relevant parts of the program, the user is able to advance execution backwards to examine program states before the detected fault. Similar to the initial forward execution, users might overlook defects when executing backwards. By alternating between reverse and forward execution the user is able to quickly pinpoint the cause of a failure.

The reverse execution functionality provided by the ESD debugger consists of an unconditional resume, which runs execution backwards until either a breakpoint is hit or the program's beginning is reached. This is supplemented by backwards stepping, either on source- or instruction-level. Just like the forward counterpart, backwards stepping will consider function calls and bring execution either to the function's last executable source line or to its return instruction.

4.2.4 Trace Management

The trace management facilities provided by the ESD debugger play an important role concerning the execution control of a debuggee process. In contrast to the two former sections, where execution control centered on the ability to suspend and resume program execution, the focus is now on choosing between possible paths of program continuation. The ability to control process evolvement in this spirit is seldom found in debug architectures and stems from the requirement to resolve non-determinism introduced by possible environment abstractions. Furthermore, the trace management is responsible for recording the microcontroller's execution states while debugging a program.

The trace management functionality is provided by two separate components. The trace manager is implemented as part of the debuggee component (see Figure 4.1) and has a direct handle to the simulator, whereas the trace element is part of of the debug element hierarchy (see Figure 4.2). Both work in close collaboration to establish the required functionality. The user's handle to the trace management facilities is provided by a third component, namely the trace view, and is depicted in Figure 4.6.



State 3622 (ID: 667225079)
State 3621 (ID: 1144134567)
State 3620 (ID: -2011728020)

Successor 1 (ID: 1035357800)
Successor 2 (ID: -539121705)

Figure 4.6: Trace View

The trace view consists of two vertically stacked components, namely a trace and a successor component. Both components are implemented as JFace table viewers and rely on the trace element to control and provide the underlying data model. When the debug target is suspended, the upper trace component gives an overview of the microcontroller's execution history by showing a list of states, whereas each state corresponds to a particular microcontroller state. At any time, a single state will be selected and marked by a special selection pointer (➧). The selected state is responsible for the program context shown in all other debug views, for example the registers view or the memory view (see Section 4.3 for further information concerning context information).

The successor component below complements these context information by displaying all possible successor states for the selected trace state. The presence of multiple successor states for the selected trace state indicates the existence of different ways of program continuation and as mentioned is caused by environment abstractions which introduce non-determinism. A simple example for this situation can be described by a program which contains an interrupt handler for an external interrupt. The programmer decided to abstract from the external device which triggers the interrupt and only provided an abstraction. As consequence the occurrence of the external interrupt is non-deterministic. In all program states where both the microcontroller's interrupt system and the particular interrupt are enabled, the debugger has to resolve the question whether the interrupt has occurred or not. A positive answer requires execution to continue with the interrupt handler, whereas a negative answer lets execution continue normal. Trace states where the interrupt system and the particular external interrupt are both enabled therefore contain at least two successors, one for each possibility. The basic purpose of the successor component is to provide means to let the programmer explore all possible ways of continuation. The user can iteratively select successor states to analyse the particular effect on the microcontroller's state, for example the program counter, register and variable values, and more. Again, the selection pointer is used to signal that the program context shown in all other debug views then depends on the selected successor state.

At some point the programmer has sufficiently analysed the current program state and possible ways of continuation and wants to resume execution. Forward execution techniques such as an unconditional resume or stepping acknowledge the chosen successor state for process evolution. The execution of a single machine instruction corresponds to appending another state to the current trace list. In case an instruction-level step is issued, the chosen successor becomes the new tail element and execution is suspended again. However, an unconditional resume or source-level step usually requires many machine instructions to be executed until execution is suspended again. In those cases a multitude of states have to be appended to the trace list. As some of these states might allow multiple paths of continuation the question arises how the debugger resolves non-determinism when execution has passed beyond the chosen successor. The default approach to handle the occurrence of a program state with multiple successors is to suspend execution and pass responsibility for making the choice to the user. In many cases tough, the necessary user interaction in this default mode will be too high to advance execution at a suitable pace. For this reason the trace view's toolbar provides the possibility to select a successor selection strategies. Currently implemented selection strategies include two random strategies. The *CompletlyRandomStrategy* selects a successor state by random choice over all successors whereas the *AvoidInterruptsStrategy* restricts the eligible successors to those that avoid interrupt handlers and then randomly chooses one of them.

To summarize, when debugging an application program with the ESD debugger an execution trace is recorded. In case the program's execution is deterministic the trace is build without any further intervention by the debugger. Alternatively a specific trace is created by either employing manual user choice, a successor strategy or a combination of the former two options. Independent of the fact whether program execution is characterised by an execution tree or a single path the ability to replay a recorded trace or parts thereof is beneficial for a better understanding of the program and the detection of possible anomalies. For this reason, the above mentioned trace component allows the unrestricted selection of states and the user can thus choose a previous state in the history list. When execution is then resumed, the debugger will work in replay mode and rerun the already existing trace by default. The user can employ the same set of breakpoints to suspend execution at identical locations. In most cases though the usage of a different set of breakpoints and stepping behaviour is advantageous to analyse specific aspects of the recorded execution. It is therefore possible to deactivate the replay mode by utilizing the corresponding toggle button in the trace view's toolbar. When this is done and execution is resumed from a previous state in the trace list the trace future is open again.

In many cases the detection of software failures does not go hand in hand with the defect localization process. Typically, there is a time difference between both activities. In large software projects there is often also a difference in the personell responsible for each activity. Bug tracking systems, such as Redmine, Bugzilla or Mantis highly support the software development process by providing a platform to keep track of software failures. Keeping failure information of software applications within such a system helps to coordinate the debugging process. An important information which should be provided by a failure report is the description of how the failure can be reconstructed. In our case this information has implicitly been captured by the trace management via recording the visited program states. For later reuse, for example as attachment to a failure report, the trace view allows to save the trace to a user-definable file. Truncating the trace before the save operation is carried out is possible by employing the truncate trace button. The saved trace can be loaded at a later time and replayed by the ESD debugger.

4.3 Inspecting Program Context

In the former sections we explained different means for controlling program execution. However, execution control is not the primary goal of a debugger but rather a necessity to provide the most important information the user needs during debugging: program context information. Program context can include several types of information such as the current source code line, a function call stack, variable and register information, a memory dump, and more. These information can be used answer various questions, including 'Where is the program?', 'How did the program get here?' or 'What are the values of variables?'. Being able to answer these questions results in an in-depth understanding of the program and lets programmer reason about possible failure causes. Before we continue with a detailed description of the information that in total make up the program context, and how these information are presented by our debugger it is first of all necessary to shed more light on the toolchain for executing an embedded system application within the NIPS VM.

4.3.1 Debugging Information

The toolchain as depicted in Figure 2.3 is responsible for the translation of a high-level C source code program to a NIPS bytecode file. The process is quite complex but essentially involves recasting the high-level source into simpler and simpler forms until eventually the result is a sequence of operations that the NIPS VM understands. Information which is not necessary for execution, such as source line information, variable names and types, function names, etc. is discarded during these transformation steps. However, these information are a prerequisite for source-level debugging and are required for the provision of important program context information. The preservation of this data, henceforth debugging information, is an absolute necessity.

Various debugging data formats, such as stabs, DWARF, COFF, IEEE-695 and Program Database have been developed in the past as storage format for debugging information collected by a compiler. For the first four formats the debugging information is stored within the resulting object files or executable file. Whereas COFF and IEEE-695 have been designed for a specific object file format, stabs and DWARF support different object file formats. In case the stabs or DWARF format is used, the concrete representation of the debugging information also depends on the particular object file format. This is contrasted by the Program Database debug format where debugging information is stored in separate files. The format is used by compilers shipped with the Microsoft Visual Studio development environment.

As described in Section 2.3 the first element within our toolchain consists of an open-source C compiler, namely the GNU C compiler. The mentioned compiler supports multiple debugging data formats, including stabs and DWARF. Although the output of debugging information is not enabled by default, additional parameters can be set to include debugging information in the resulting executable file. By this means the provided debugging information create a connection between the high-level program and the generated microcontroller machine code. However, the final result of our toolchain is not the binary executable file but instead a NIPS bytecode file which is created in a further transformation step (see Section 2.3). The debugging information required by our debugger has to create a link between elements of the high-level source language and the resulting NIPS bytecode file. In order to support the retrieval of disassembly context information (see Section 4.3.4 for further details) a connection between the NIPS bytecode file and the disassembly information extracted from the binary executable is additionally required.

In the following we will discuss the debugging information capabilities of the NIPS bytecode format, that can be employed to create the above mentioned connections between source code, machine instructions and bytecode. Independent of these yet unknown capabilities, the usage of the debugging information provided by the C compiler will be necessary in any case. This paragraph is therefore succeeded by an overview of the debugging data format of our choice. We decided to focus on the stabs debugging data format as it foremost provides full support for C programming language elements and furthermore allows for a resource-efficient information processing implementation due to its simple structure.

Stabs

The Stabs format has initially been developed by Kessler to provide debugging information support for the Pascal programming language [MKM93, SM04]. The support has later been extended to include other languages such as Cobol or Fortran 77. However, it is the comprehensive support for C which makes stab interesting for this project. Stabs stands for *symbol* *table entries*, referring to the fact that debugging information was originally placed in the symbol table of an object file. But the starting point for this overview should not be an object but instead a source file. The first step towards an executable file is carried out by the compiler which transforms the source file into an assembly language representation. The result is used by the assembler to produce an object file, which the linker combines with possible other object files and libraries to create an executable file. So far we have described the basic process for machine-code compilation. However, the resulting executable file will not contain debugging information. For this purpose, compilers that support stabs can use specific assembly directives to place debugging information into the generated assembly file. The information contained within these directives will be processed by the assembler and stored in the object file. In the last step the linker has to make sure that stabs information is passed to the resulting binary executable.

The usage of stabs assembly directives allows compilers to preserve information necessary for source-level debugging. A further analysis of these instructions is necessary to understand the capabilities of the stabs debug format. The stabs format defines two assembly directives, namely a `.stabs` (string) and a `.stabn` (number) directive with the following format:

```
.stabs "string", type, other, desc, value

.stabn type, other, desc, value
```

The difference between both instructions manifests itself in the possibilty to attach an variable-length string to the `.stabs` instruction. The `type` attribute is a numeric value and specifies what type of information the stab represents. Stabs types exists to describe source-line information, local and global symbols, type information, function descriptions, lexical blocks and more (cf. Table 4.1). The interpretation and possible values for the remaining attributes, that is, `other`, `desc` and `value` depends on the particular stabs type.

Source line information, for example, can be described by the `N_SLINE` stab type. The `desc` attribute contains the line number and the `value` field contains the instruction address for the start of that source line. The instruction address is not an absolute value but is relative to the function in which the `N_SLINE` symbol occurs. The specification of source-line information is supported by the `.stabn` directive as there is no need for a variable-length string value. This situation changes when we consider function descriptions - they make use of the variable-length string value provided by the `.stabs` directive to store the function name and references to previously defined type information for parameter and return value. Whereas the `value` field contains an absolute instruction address to describe the beginning of the function's translation, the source line is provided in the next `N_SLINE` stab.

As indicated by the mentioned examples the placement of stabs directives can not take place in random fashion but must conform to a specific organization. The first stab for each source file has to be of type `N_SO` and specifies the name of the source file. This information is proceeded by directives that describe the types of variables, parameters or function return values. Since the stabs format does not include any predefined types, the compiler has to

Stab type (symbolic constant)	**Description**
`N_SO`	Source code filename
`N_SLINE`	Source code line number
`N_FUN`	Function name
`N_GSYM`	Global symbol
`N_LSYM`	Stack variable or type information
`N_LBRAC`	Beginning of a lexical block
`N_RBRAC`	End of a lexical block

Table 4.1: Several common stab types (excerpt from [MKM93])

describe every type used in the program by means of the `N_LSYM` stabs type. The string value is used in this case to store the type name and further information such as the size, maximum value and minimum value. The following directive for example defines the char type as allowing values from 0 to 255:

```
.stabs "char:t(0,2)=r(0,2);0;255;", N_LSYM, 0, 0, 0
```

Besides function return values and parameters, the defined type information is also necessary for variable information. Global variable descriptions are for example supported by the `N_GSYM` stab type. Whereas the variable name and type can be inferred from the `"string"` field, the address is not part of the stabs directive and has to be retrieved from the external symbol for the variable. For a complete list of stab types and their usage we refer to [MKM93] and [SM04].

By concentrating on the stabs assembly directives we have introduced the stabs debug format from the compilers point of view. We shall now focus on the debugger's perspective and consider the question, where debugging information is stored within the resulting executable file. There is actually no straight answer to this question as the storage location is dependent on the executable file format used. For the a.out format, the first executable file format to support stabs, debugging information is stored in the executable file's symbol table. Each stabs directive contained in the assembly file results in a fixed-length entry in the symbol table. The variable-length character sequence contained in a `.stabs` directive is not stored in a symbol table entry but is saved in the executable file's string table. Symbol table entries corresponding to a `.stabs` directive refer to their `string` attribute by an offset value targeting the string table. Executable file formats with support for custom section, such as ELF, SOM or COFF, do not store stabs directives in the symbol or string table. Instead the stabs debugging information is stored in two special sections, namely a *.stabs* and a *.stabstr* section. Similar to the concept above, the *.stabs* section contains a fixed length structure for each stab directive, whereas the *.stabstr* section contains all the variable length strings that are references by stabs in the .stab section.

At this point we would like to conclude our discussion on the stabs debug format and continue with an introduction of the debugging information capabilities provided by the NIPS bytecode format.

Debugging Information for NIPS

The NIPS instruction-set features two annotations which can be used to store additional information within a NIPS bytecode file:

```
!srcloc line column

!strinf tag keyword name
```

The `!srcloc` annotation allows to mark a sequence of bytecode instructions with the corresponding source line from which the code sequence has been created. The `!strinf` annotation can be employed to save structure information and consists of three parameters. The keyword and name attribute may contain arbitrary string values, whereas the tag parameter is either *begin* or *end*. Structure information can be used to save further information about code sequences. For example, that the currently executed code corresponds to the translation of an if-construct in the source language or at what locations the translation of functions and processes begin or end. However, the described abilities to store information about the source program within a NIPS bytecode file have not been designed to specifically cater for the needs of source-level debugging but as stated in [WS05] rather provide means to support subsequent bytecode optimizations. The featured debugging information capabilities are not sufficient to allow source-level debugging as, for example, the storage and later extraction of type, variable or register information is not supported. It is therefore necessary to discuss possibilities to cope with the limited debugging information functionality.

The approach we would like to present first, is based on the fact that although the provided annotations are not able to capture the connection between high-level source and NIPS bytecode, the situation is different with respect to a possible connection between microcontroller machine code and NIPS bytecode. On the machine code level the original source code has taken a greatly simplified form and the annotations described above allow to establish a connection between machine code instructions and NIPS instructions. During the execution of NIPS process steps, it is then possible to infer from which microcontroller instructions the NIPS instructions originated. The bytecode file then supports the retrieval of disassembly information, however, elements of the source file, such as function names or type information, can not be accessed directly. This ability can be provided by keeping the binary executable and the stabs debug information stored within. The resulting dependency is a small disadvantage but is compensated by low development costs. More serious is the fact that the storage location within a NIPS state for elements such as global variables is not made explicit. The information can be retrieved by reverse engineering the compiler which is responsible for the machine code to bytecode translation, however, this contradicts the idea of debugging information.

The limited support of the NIPS bytecode format regarding the description of debugging information has been topic of the work done by van Roozen [vR07]. The objective of this work was to create the possibility to describe the information contained within the NIPS byte-buffer states to allow source-level debugging. A result of this work has been the development of the Static Debugging Information (SDI) language which allows to extract source-level artifacts, such as variables, names and types from the states generated during execution. SDI comes with a textual as well as graphical representation. The author points out that SDI is designed to describe debugging information for modeling languages such as Promela. But currently, SDI does not provide support for programming languages. We have therefore decided to continue with the previously described approach.

4.3.2 Program Call Stack

At this point we shall continue our discussion on program context information and focus on the individual elements it consists of. We would like to start out with an element that helps to answer the question "How did the program get here?" when execution is suspended. Before we introduce the program call stack, or call stack for short, let us point out that we already presented a component that provides information to reason about the above mentioned question. As explained in Section 4.2.4, the trace manager and its corresponding user interface, namely the trace view, not only allow to control program execution but also provide information to figure out how the current program location was reached. The trace management records every microcontroller state seen during execution and by looking at the instruction address contained within each state, the user gets a precise understanding of the program flow. The information contributed by the call stack to reason about the program flow is less accurate, however, it is sufficient to analyse a single state.

The call stack is a data structure kept in the microcontroller's memory that stores information about the currently active functions and interrupt handlers. Active in this context means that the function or interrupt handler has been called but has not yet returned. The existence of such a data structure arises with the need to store the instruction addresses at which execution should continue once a called function has finished execution. These so-called return addresses are an integral part of the call stack, however, there is usually more information, such as the values of local variables and function arguments, stored on the call stack. All data associated with an active function is stored as contiguous block, a so-called stack frame. A stack frame, or frame for short, is created on the stack once a function is entered and destroyed on exit. From a high-level view, the call stack can be seen as a list of frames, whereas the top frame corresponds to the currently active function. Access to the stack is often realized by a designated register, namely the stack pointer, which points to the current top of the stack. On most architectures, access to the frame structure is simplified by a separate register, often termed frame pointer, which points to some fixed location within the top frame such as first local variable or the location of the return address. A section of an example call stack is sketched in Figure 4.7 to exemplify the described content. The example stack grows from the highest to the lowest memory address and the selected section shows the last inserted frames including the current frame and stack pointer. A new frame is currently build for a forthcoming function call and parameters have already been stored on the stack. The return address

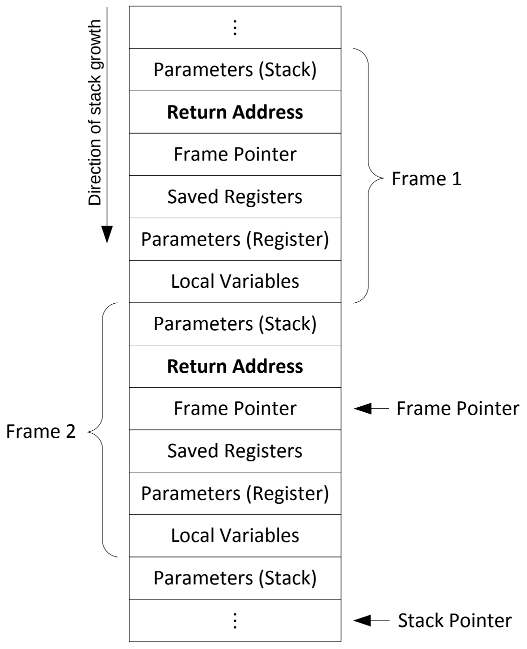


Figure 4.7: Program call stack

will be written on the stack once the corresponding `Call` instruction has been issued.

The focus of this section is on the question, how information can be extracted from the call stack to let users know about the active functions and interrupt handlers. The standard approach to this problem is to start out with the top stack frame and then continue analysis backwards until the bottom of the stack is reached. The greatest difficulty with this task, also known as stack unwinding, is due to the fact that there is no uniform stack layout. The stack layout rather depends on the architecture and compiler in question. Information about the employed procedure calling conventions, that is, the rules which dictate how the stack is build and where the various pieces of data, such as the return address, lie within each frame, are in most cases only informally described in the reference manual for the targeted architecture or employed compiler. Due to the described problems we decided to first of all concentrate our development effort on the ability to retrieve the most important contained on the stack, namely the return addresses. The existence of a frame pointer decreases the development effort in some cases. Relevant is not just the availability of a frame pointer register, but more important is whether and how the frame pointer is used by the machine code generated by the compiler. The example stack given in Figure 4.7 describes a scenario where the present conditions simplify the desired stack unwinding algorithm. Whenever a new frame is created the value of the frame pointer register is saved on the stack. This is done after the return address is written but before the frame pointer register is modified as consequence of the new frame's initialization. The storage location is at a fixed offset from the return address. The frame pointer register itself points to the location of the stored frame pointer within the last created frame, that is, the top frame. By this means, it is possible to unwind the stack by simply following the chain of frame pointers. For the retrieval of the desired return addresses it is then only necessary to add a fixed-size offset to the respective frame pointer locations.

Although the targeted ATmega16 microcontroller does not feature a designated frame pointer [Atm05], the AVR GCC compiler employs the Y-register for this purpose. Similar to the situation above the value of the Y-register is stored on the stack whenever a new frame is created. However, the register value does not point to the location of the stored frame pointer, but instead, to the first local variable. This means that there is a variable size offset between the location referred to by the Y-register and the stored frame pointer or the return address. Knowing about the frame pointer has only little value in this case as an implementation based on the frame pointer still has to know about the procedure calling conventions to determine the size of this offset for each frame.

Independent of the question whether the stack unwinding algorithm can be based on the frame pointer or not, it is generally not a good idea to do so. This is because changing the compiler or targeting a different architecture will most likely break the algorithm. A better solution to this problem is to stay with the generated machine code and let the algorithm find out for itself how the stack was build. Although the initial development effort of this approach is quite high and also requires assembly programming experience it proves to be robust with respect to compiler changes and requires only minor changes to support different architectures. But again, due to the complexity involved we have decided to limit our development effort on the retrieval of the return addresses to manage in time.

Our algorithm is based on the fact that immediately after a `Call` instruction is executed the last value written on the stack is the return address. Thereby, the stack pointer is decremented and points to the next free address, that is the address right after the return address. If we keep track of modifications to the stack pointer between the function call and its return instruction we are able to specify a numeric offset for each instruction address in this range. The offset should reverse all stack pointer modifications made by instructions following the `CALL` instruction until the address under consideration. Adding this offset to the current stack pointer then gives the address of the return address. Regarding possible modifications of the stack pointer we can distinguish between two cases, namely implicit and explicit modifications. Implicit modifications are characterized by the fact that they occur automatically as side effect of certain instructions, such as the `PUSH` or `POP` instruction. The `PUSH` instruction for example stores the content of the parameter register on the stack and decrements the stack pointer by 1 afterwards.

Explicit modifications are described by those instructions whose primary target is to change the stack pointer value, either by direct or indirect addressing. The `OUT` instruction for example uses direct addressing and is able to modify the stack pointer because it writes the value of a given register to a specified I/O location[1]. Handling implicit modifications requires a small development effort only and a simple forward scan can be employed for the task at hand. Explicit modifications in contrary require a substantial larger amount of work to be

[1]The stack pointer of the ATmega16 is implemented as IOR_{61} and IOR_{62} in I/O space

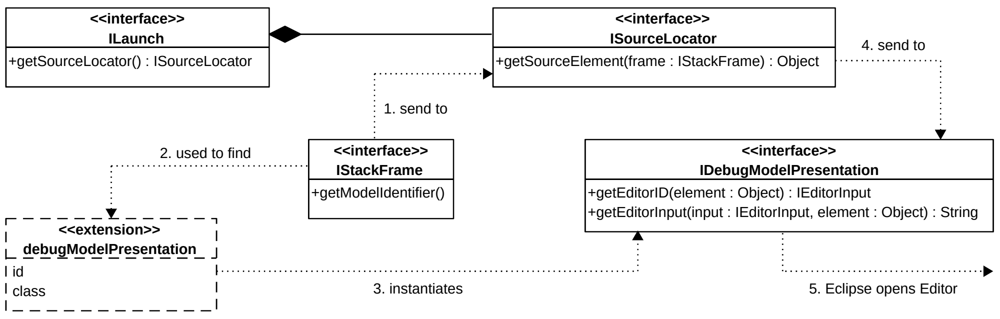


Figure 4.8: Eclipse Source Lookup Model [Wri04]

taken care of. The job includes detecting whether the current instruction writes to the stack pointer, retrieving the value to be written and also verifying that no arbitrary value is written but that just an offset is added or subtracted. The forward search has been extended to cater for detecting the corresponding instructions, whereas an additional backward search has been introduced to take care of the last mentioned tasks.

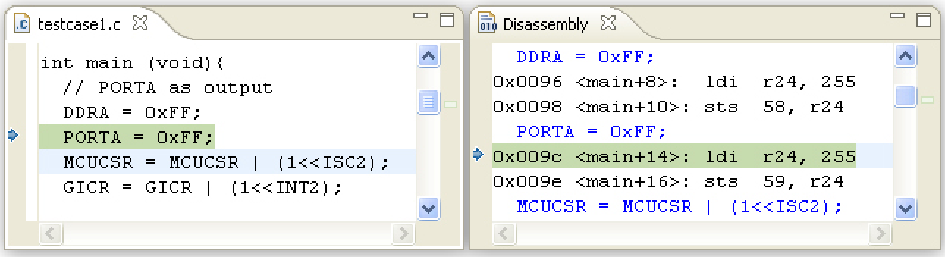
The results of our stack unwinding effort can be noticed in the debug view as shown in Figure 4.4 on page 32. The second application under debug is currently suspended and items 3 and 4 display the active functions which have been found on stack.

4.3.3 Source-Level View

We have explained in previous sections how programmers can answer the question “How did the program get here?” by using context information provided by the recorded trace or the program call stack. This ability is very important and allows programmers to confirm or refute their intuition about program execution. However, interest in the question “Where is the program” almost always comes first when execution is suspended. “Users think in terms of their source code” is a statement made by Heisenberg [Ros96] and indicates which information programmers are interested in to answer the last mentioned question. When execution is suspended, a source-level debugger is at least required to highlight the current source line or statement. However, the ability to analyse code prior or after the current source line is effectively mandatory for modern debuggers. In the following we will describe how the provision of source code, henceforth source code lookup, is realized by the ESD debugger.

We based our implementation on the source lookup framework which is available since Eclipse 3.0. Particular beneficial about this framework is the fact that the context for source lookup does not manifest itself in a suspended debug target but rather in the debug target’s selected stack frame. Therefore it is not only possible to analyse source code corresponding to the current instruction address as provided by the top stack frame but also view source code for function calls or triggered interrupt handler routines in stack frames below.

Before we continue our discussion of the source lookup functionality we would like to refer to Figure 4.8 as a first overview. Each launch object has a source locator object associated



```
int main (void){
  // PORTA as output
  DDRA = 0xFF;
  PORTA = 0xFF;
  MCUCSR = MCUCSR | (1<<ISC2);
  GICR = GICR | (1<<INT2);
```

```
  DDRA = 0xFF;
0x0096 <main+8>:  ldi  r24, 255
0x0098 <main+10>: sts  58, r24
  PORTA = 0xFF;
0x009c <main+14>: ldi  r24, 255
0x009e <main+16>: sts  59, r24
  MCUCSR = MCUCSR | (1<<ISC2);
```

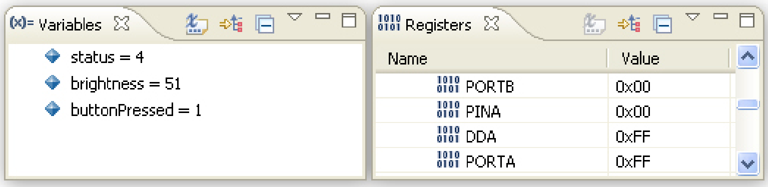
Figure 4.9: Source code and corresponding disassembly

with it which is responsible for locating the source element for a specific stack frame. The source locator can either be manually instantiated by the launch delegate or is automatically installed by the framework by contributing to the source locator extension point as employed for the ESD debugger. The first step carried out by source locator is to map the stack frame to a filename and then search for the corresponding file in a given source path. The source path thereby consists of a set of source containers, such as workspace folders, projects, file system directories or archives. By means of a source lookup tab which is added to the launch configuration tab group, the programmer is able to configure and modify the source lookup path. Alternatively a programmatically specified or default lookup path is used when the user has no choice or makes none.

Once the source element for a stack frame has been retrieved it is the job of the debug model presentation object to map the source element to an editor input and editor id. This information is used by the framework to open the corresponding editor, display the source code and position the editor to the line specified by the stack frame. Furthermore, two instruction pointers are employed for highlighting; either ➧ to denote the source line corresponding to the current instruction address or ⇨ in case the source information context is determined by a stack frame selection below the top frame. The above stated requirements concerning source context information are therefore completely fulfilled by our debugger.

4.3.4 Disassembly

For certain difficult to find bugs – a concrete example is provided in our case study in Chapter 6 – the high-level source will be too imprecise to determine failure causes. In these cases, the defect localization requires to analyse execution at a more fine-grained level, that is, to examine executed or future machine instructions. The requirement has been addressed by other state-of-the-art debuggers as well: The GNU debugger, for example, provides means to display machine instructions by means of the *disassemble* command [gdb09]. The current version of the platform debug model provides no support for disassembly retrieval and display. Thus we integrated a custom disassembly framework within the ESD debug model. Identical to source lookup, the context for disassembly retrieval is characterised by the debug target's currently selected stack frame. In contrast to the source lookup, the necessary information for further processing is not already contained in the stack frame and has to be



status = 4
brightness = 51
buttonPressed = 1

Name	Value
PORTB	0x00
PINA	0x00
DDA	0xFF
PORTA	0xFF

Figure 4.10: Variables and register view

retrieved from the backend by means of a corresponding command. Given filename and first line number of the stack frame's function, the backend is asked to retrieve the machine instructions that implement the function. The programmer is enabled to access the retrieved information by means of the disassembly view, which is depicted in Figure 4.9 on the right side next to the source view.

The NIPS VM does not actually execute machine instructions of the target processor but rather simulates their effect by translation to a NIPS bytecode program. Once this compilation has finished it is not possible anymore to infer the microcontroller machine instructions from the established NIPS instructions. The retrieval of disassembly therefore either relies on debug information provided by the NIPS bytecode file or otherwise employs the disassembly extracted from the binary executable which has been used for the bytecode translation in the first place.

4.3.5 Variables and Registers

Being able to inspect the values of variables used in the program's source code is an important feature of every source-level debugger. For embedded systems the ability to inspect register values is equally important. Figure 4.10 shows the variables and registers view. They are provided by the debug project and operate on the ESD debug model to display the existing variables and register values.

4.3.6 Memory

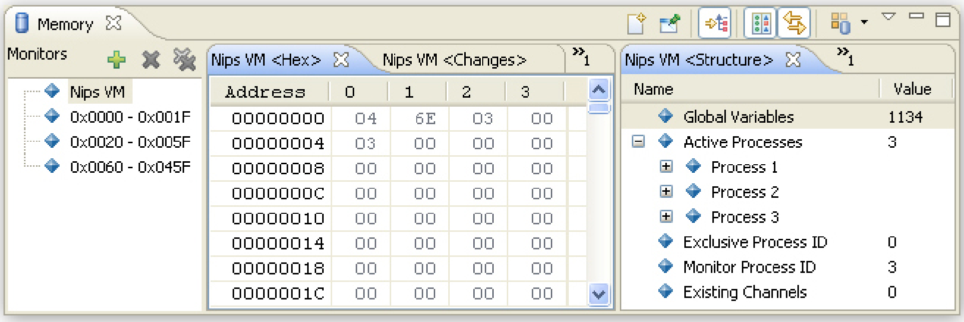
An in-depth view of the current microcontroller state can be obtained by inspecting the microcontroller's memories. Inspecting the microcontroller state on such a low level allows, for example, to determine the precise layout of stack frames or to retrieve values of saved registers or local variables stored within these frames. Although many microcontrollers face severe restrictions concerning memory size, the complete memory content is usually still too large for programmers to handle as a whole. For effective memory examination the debugger is therefore required to allow programmers to concentrate on certain sections of the memory.

So far, we focused our motivation for memory examination entirely on the targeted microcontroller. In fact, most users of our debugger will show no interest in the underlying NIPS VM which simulates the microcontroller and runs the microcontroller application. The ability to debug their program is all they are looking for. For these users the motivation for memory examination does not go beyond what is mentioned already. However, for certain users, such as developers of the NIPS VM or users interested in virtualization technology, the underlying NIPS VM does matter. In this case the user will benefit from and therefore also require the ability to inspect the NIPS VM's memory state during program execution. As discussed in Section 2.1 a NIPS state obeys to a specific structure featuring a global variable space, processes and channels. Providing the capabilities to analyse the NIPS memory state in a raw byte fashion is benefitial, but being able to reflect the mentioned structure in the memory representation adds additional value. Before we continue with the description of the memory exploration capabilities of the ESD debugger let us first summarize the three most important requirements:

1. Ability to inspect both microcontroller- and NIPS VM memory state
2. Skill to focus on certain sections (either by address or name)
3. Provision of different representation formats

The support for memory retrieval and examination as provided by the ESD debugger is based on the debug facilities supplied by the debug project, namely the memory framework and the memory view. The central concept of these facilities is referred to by the term memory block and represents a section of memory. A memory block constitutes an element of the debug element hierarchy and in its simplest form is determined by a start address (also know as base address) and a length value. More complex debuggers, including our own implementation, furthermore allow to specify memory blocks by custom expressions. Although the debug target element is the authority when it comes to the creation of new memory block instances, it is the memory block element itself that is responsible for the retrieval of memory information from the debuggee process. With respect to the Model-View-Controller architectural pattern [BMR+96], the memory block element takes the role of the model. Consequently, a memory block does not make any statement about how the contained information is represented to the user. Instead, the task of representing memory information falls to an entity known as memory rendering which is responsible for displaying a memory block in a specified data format. The platform provides four default renderings which all display memory in a table layout but differ in the actual byte representation: It is either Hex, ASCII, unsigned integer or signed integer. However, as demonstrated by our implementation custom memory renderings can be contributed that display information in a completely different format.

Programmer's access to the memory functionality, for example adding a new memory block or inspecting the information contained within, is provided by the memory view. The user interface component does not actually use the term memory block but refers to them as memory monitors instead. The memory view is depicted in Figure 4.11 and consists of three panes: The *Monitors* pane on the right lists all memory monitors that are currently installed in the current debug session and provides means to add or remove memory monitors. The two panes to the left are called rendering panes and display the memory information contained within the selected memory monitor from the *Monitors* pane. When the programmer wants to add another memory monitor he is requested to insert necessary information via a corresponding dialog. As the underlying NIPS VM is of interest for a restricted number of users only, our



Monitors:

- Nips VM
- 0x0000 - 0x001F
- 0x0020 - 0x005F
- 0x0060 - 0x045F

Address	0	1	2	3
00000000	04	6E	03	00
00000004	03	00	00	00
00000008	00	00	00	00
0000000C	00	00	00	00
00000010	00	00	00	00
00000014	00	00	00	00
00000018	00	00	00	00
0000001C	00	00	00	00

Name	Value
Global Variables	1134
Active Processes	3
Process 1	
Process 2	
Process 3	
Exclusive Process ID	0
Monitor Process ID	3
Existing Channels	0

Figure 4.11: Memory view displaying the current NIPS VM memory state

primary support targets the microcontroller. Providing a start address and an optional length value therefore creates a memory monitor for the specified section of the microcontroller's memory. If a length value is not provided the resulting monitor will cover the complete memory from the given start address to the highest possible address. Alternatively the dialog allows to choose the predefined "NIPS VM Memory State" entry in order to create a memory monitor for those users interested in the NIPS VM memory.

5 LTL Model Checking

Validation approaches such as testing or simulation are able to show the existence of failures, however, they are not able to prove their absence. As embedded systems are oftentimes employed in environments where failures are unacceptable the need for verification strategies to prove or disprove the system's correctness is obvious. One approach to this verification problem is described by a method called model checking. Its introduction and the connection between debugging and model checking is the topic of this chapter.

Model checking operates on a model of the considered system and is not employed on the system directly. For a given system model and a specification the model checking problem consists of the decision whether the model satisfies the specification. The model checking algorithm solves this decision problem by exhaustive search of the model's state space, and under assumption of sufficient memory and time resources always terminates with a yes or no answer. In either case, that is a positive or negative answer, the result of this analysis concerns the model and not yet the actual system. The implication that from the model's correctness one can infer the system's correctness is only true in case the model is a so-called system *abstraction*. This means that all execution paths defined by the system are captured by the model as well. A weakness of the model checking technique is found in the circumstance, that a formal proof of this property is not possible in general. Advantageous about model checking is the fact that the underlying process is automatic and requires no user interaction with exception of the result analysis. In case a given specification is not satisfied by the model a counterexample is generated which gives insight about possible system design errors. It is necessary to distinguish between feasible and infeasible counterexample; a feasible counterexamples describes an erroneous behaviour that also exists in the original system, whereas the failure captured by an infeasible counterexample exists in the model only.

In the following, we would like to focus on the three elements of model checking, namely, modeling, specification and the actual verification algorithm. We will introduce Kripke structures as formal and mathematical precise representation of system models and analyse the Linear Temporal Logic (LTL) as specification language. Regarding the verification algorithm we will discuss the automata-theoretic approach to model checking and introduce the *Nested Depth First Search.*

5.1 Modeling

The correctness of an embedded system depends on its behaviour over time. The system model, which is a prerequisite for model checking, must provide means to capture the system behaviour in a precise and unambiguous way. At the same time, the model must omit all information that is not necessary to verify the considered specifications. Kripke structures have proven to be a suitable formalism for the modeling job at hand, and provide an intuitive way to capture the behaviour of systems. A Kripke structure consists of a set of states, a set of transition between states, and a function that labels each state with a set of atomic propositions. States contain information about the system at a certain moment of its behaviour, whereas transitions describe how the system evolves from on state into another. Atomic propositions express known facts about the states of the system and by this means abstract from the complete information captured within a state.

Definition 5.1 Kripke structure

Let AP be a set of atomic propositions. A Kripke structure M over AP is a four tuple $M = (S, s_I, R, L)$ where

- S is a finite set of states,
- $s_I \in S$ is the initial state,
- $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $(s, s') \in R$,
- $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

Whereas the behaviour of the original system is defined by its possible executions, the equivalent in the model consists of paths through the Kripke structure.

Definition 5.2 Path

A *path* in the Kripke structure M is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s_I$ and $(s_i, s_{i+1}) \in R$ holds for all $i \geq 0$.

The specification language we are interested in does not directly refer to the states of a path, but instead considers the atomic propositions that are observed. The following definition of a trace simplifies the introduction of the specification language.

Definition 5.3 Trace

Let M be Kripke structure and π a path in M. The trace of π is defined as $trace(\pi) = L(s_0)L(s_1)L(s_2)\ldots$. For $\sigma = trace(\pi)$ we define $\sigma[j\ldots] = L(s_j)L(s_{j+1})L(s_{s+2})\ldots$ to be the suffix of σ starting with $L(s_j)$.

5.2 Specification

The Linear Temporal Logic (LTL) extends classical propositional logic with a set of temporal operators and allows to make statements about system executions. Despite the adjective *temporal*, LTL does not introduce time explicitly but instead specifies the relative order of events.

Definition 5.4 Syntax of LTL

LTL formulae over the set $AP = \{p_1, \ldots, p_n\}$ of atomic propositions are inductively defined as follows:

- p_i is formula
- If φ and ψ are formula, then so are $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$.
- If φ and ψ are formula, then so are $X\varphi$, $F\varphi$, $G\varphi$, $\varphi U \psi$

Definition 5.5 Sematics of LTL over Paths and Traces

Let M be a Kripke structure and φ be a LTL-formula over AP. A path π of M satisfies φ, denoted $\pi \models \varphi$, if and only if the corresponding trace $\sigma = trace(\pi)$, satisfies φ. The satisfaction relation for trace σ is thereby inductively defined as follows:

- $\sigma \models p \quad \text{iff} \ \ p \in L(s_0)$
- $\sigma \models \neg\varphi \quad \text{iff} \ \ not\ \sigma \models \varphi$, similarly for $\wedge$ and $\vee$
- $\sigma \models X\varphi \quad \text{iff} \ \ \sigma[1\ldots] = L(s_1)L(s_2)L(s_3)\ldots \models \varphi$
- $\sigma \models F\varphi \quad \text{iff} \ \ \exists j \geq 0\text{: } \sigma[j\ldots] \models \varphi$
- $\sigma \models G\varphi \quad \text{iff} \ \ \forall j \geq 0\text{: } \sigma[j\ldots] \models \varphi$
- $\sigma \models \varphi_1\, U\, \varphi_2 \quad \text{iff} \ \ \exists j \geq 0\text{: } \sigma[j\ldots] \models \varphi_2$ and $\sigma[i\ldots] \models \varphi_1$, for all $0 \leq i < j$

Definition 5.6 Satisfaction of LTL formulae by Kripke structures

A Kripke structure M satisfies a LTL formula φ over AP, denoted $M \models \varphi$, if all paths of M satisfy φ.

By using Kripke structures as formalization for system models and LTL as specification language the informal description of the model checking problem can be reformulated:

Definition 5.7 LTL Model Checking Problem
Given a Kripke structure M and a LTL formula φ over AP decide whether $M \models \varphi$.

5.3 Verification algorithm

The algorithm which is presented here to solve the model checking problem is based on the automata-theoretic approach suggested by Vardi and Wolper [VW86]. The key idea of this approach is that, given an LTL formula φ, it is possible to construct a Büchi automaton $\mathcal{A}_\varphi$ that accepts precisely all computations that satisfy φ. The language of infinite words accepted by $\mathcal{A}_\varphi$ is denoted by $\mathcal{L}(\mathcal{A}_\varphi)$. The Kripke structure M which represents the system model can also be given as Büchi automaton $\mathcal{A}_M$ and the corresponding language is denoted by $\mathcal{L}(\mathcal{A}_M)$. Then, the model M satisfies the specification φ iff

$$\mathcal{L}(\mathcal{A}_M) \subseteq \mathcal{L}(\mathcal{A}_\varphi) \tag{5.1}$$

that is, each infinite word accepted by the model automaton is also accepted by the specification automaton. If $\overline{\mathcal{L}(\mathcal{A}_\varphi)} = \Sigma^\omega - \mathcal{L}(\mathcal{A}_\varphi)$ denotes the complement language of $\mathcal{L}(A_\varphi)$, equation (5.1) can be rewritten as

$$\mathcal{L}(\mathcal{A}_M) \cap \overline{\mathcal{L}(\mathcal{A}_\varphi)} = \emptyset \tag{5.2}$$

As Büchi automata are closed under complementation it is possible to obtain $\overline{\mathcal{L}(\mathcal{A}_\varphi)}$ by complementing $\mathcal{A}_\varphi$. In practise, however, it is simpler to complement the specification and use the negated formula $\neg\varphi$ as starting point for the automata construction. Thus, the model checking problem for Kripke structures and LTL can be reduced to checking the emptiness of Büchi automata.

Nested Depth First Search

The algorithm we would like to introduce to solve the emptiness problem of Büchi automata employs two interleaved depth-first searchs. The *Nested DFS* algorithm was originally proposed by Courcoubetis et al. [CVWY92]. The basic idea is to perform a first depth-first search (*blue search*) to encounter all accepting states. This is complemented by employing a nested, second depth-first search (*red search*), to search for a cycle once an accepting state has been fully expanded by the blue search. A modification of the algorithm suggested by Holzmann et al. [HPY96] improves the algorithm's memory efficiency and is used in the SPIN model checker. Gastin et al. proposed a further variant in order to find counterexamples faster [GMZ04]. Schoon and Esparza [SE05] developed yet another version which outperforms the former variants concerning both time and space. The version we would like to present here has been introduced by Gaiser and Schwoon [GS09] and is known as the currently best per-

```
1  procedure $new\_dfs()$
2    call $dfs\_blue(s_0)$;

3  procedure $dfs\_blue(s)$
4    $allred :=$ true;
5    $s.colour := cyan$;
6    for all $t \in post(s)$ do
7      if $t.colour = cyan$
8          $\wedge (s \in A \vee t \in A)$ then
9        report cycle;
10     else if $t.colour = white$ then
11       call $dfs\_blue(t)$;
12     if $t.colour \neq red$ then
13       $allred :=$ false;
14   if $allred$ then
15     $s.colour := red$
16   else if $s \in A$ then
17     call $dfs\_red(s)$;
18     $s.colour := red$
19   else
20     $s.colour := blue$

21 procedure $dfs\_red(s)$
22   for all $t \in post(s)$ do
23     if $t.colour = cyan$ then
24       report cycle
25     else if $t.colour = blue$ then
26       $t.colour := red$;
27       call $dfs\_red(t)$
```

Listing 5.1: Nested Depth First Search Algorithmus

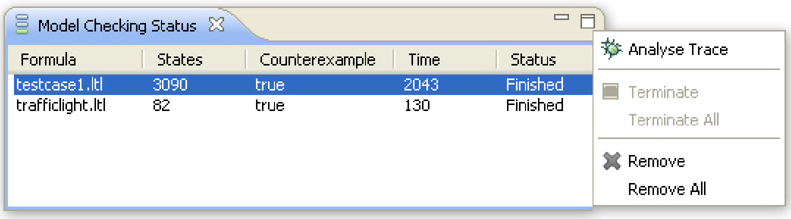
forming *Nested DFS* algorithm. It improves the previous algorithm of Schwoon et al. by avoiding certain unnecessary initiations of the red search.

The algorithm builds on the existing successor function *post(s)* that calculates all successor states of *s*. Furthermore, the algorithm requires two bits for each generated state and by this means assigns one of the following colors to each state:

- *white*: A state that has been created by a call to *post* but has not been considered by the blue search.
- *cyan*: A state whose blue search has not yet terminated. It is on the search stack of the blue search.
- *blue*: A state that has finished its blue search and has not yet been reached in a red search
- *red*: A state that has been considered in both the blue and the red search.

The algorithm from [GS09] is stated in listing 5.1. The algorithm starts the first depth-first search with the initial state s_0. Every state the blue search *dfs_blue* is invoked upon is colored *cyan* to denote that the state is now part of the current search path. The blue search can terminate and report a counterexample in line 9 without entering the red search. This situation occurs in case the considered successor state *t* is on the current search path and either the current state *s* or *t* is an accepting state. If the successor has just been generated, its color is thus white, the blue search continues.

If during backtracking an accepting state is found which has at least a single not red colored successor, the red search is invoked upon this state. A state found during backtracking that has none or only red colored successors is colored red itself to indicate that this state cannot be part of a counterexample and to make sure that this state is not touched by a future red



Formula	States	Counterexample	Time	Status
testcase1.ltl	3090	true	2043	Finished
trafficlight.ltl	82	true	130	Finished

Figure 5.1: Model Checking Status View

search. Alternatively, the current state is colored blue as the blue search has finished with this state. For the first mentioned case the red search then tries to detect a cycle by searching for a cyan colored state. This is either the state the red search has been invoked upon or a state further up the blue search's current search path. In both cases a counterexample can be reported in line 24 once a cyan colored state has been found. The red search considers only blue states when continuing its search and colors them red to make sure that they are not considered during future invocations of the red search.

In case the algorithm finishes in line 9 or 24 an accepting cycle has been found and the counterexample can be generated from the blue search's and red search's search path.

5.4 Debugging Counterexamples

Within the MCESS project, Rohrbach developed a model checker for embedded systems software on top of the NIPS VM [Roh06]. By integrating the developed model checking capabilities into the Eclipse platform we have been able to integrate model checking much closer into the development process. Being able to verify a developed application against user-defined specifications from within Eclipse greatly enhances the methods acceptance as the majority of programmers will not accept the necessity to switch from their development environment to another tool. Despite this advantage our effort has primarily been made to create the foundation for studying the benefits that can be obtained by allowing developers to analyse counterexamples by means of a source-level debugger.

Analysing and understanding counterexamples is a complicated and time-intensive task in practise. Existing model checkers for embedded systems software such as the MCESS model checker only provide insufficient support for this task and require the user to iterate over all states the counterexample consists of. It is not uncommon that counterexamples consists of multiple thousands states and iterating over each state one by one is very tedious. The ability to analyse counterexamples with the ESD debugger, that provides advanced run-control possibilities such as stepping, breakpoints and reverse execution, allows developers to manage this task with greater efficiency and comfort. The Model Checking Status View as depicted in Figure 5.1 is the starting point for launching the ESD debugger on a generated counterexample. The mentioned status view gives information about currently running or finished model checking operations. For those specifications the model checking algorithm terminated with a negative result the developer can start a trace analysis on the generated counterexample. Similar to a user-recorded trace (see Section 4.2.4) the ESD debugger loads the counterexample and allows to advance execution by means of stepping, continuing to the next breakpoint or selecting states in the trace view. In combination with the ability to inspect various program context information (see Section 4.3) our evaluation has shown that the capability to inspect counterexamples by means of a source-level debugger reduces the time to locate the defect by an order of magnitude.

6 Case Study

Although hands-on experience is the best way to become acquainted with the ESD debugger we would like to present a case study to give a first impression at least. The program that we have chosen for this purpose violates a particular safety property.

The example program is shown in listing 6.1 and consists of a main function and an interrupt handler for the external interrupt 2. It is the port pin B2 which serves as interrupt source. The first part of the main function is concerned with initialization tasks, amongst others the activation of the external interrupt. The initialization code is succeeded by the program's main loop. By means of the contained iteration loop a program is simulated which features an information exchange between main program and interrupt handler. The integer variable `i` serves as simple example for information to be exchanged. It is iteratively incremented and also reset in the main program but is not modified by the interrupt handler. Whenever the interrupt handler is entered the measure to be undertaken solely depends on the value of `i`. The interrupt handler distinguishes between three cases, whereas the first two cases, namely $\texttt{i} < 150$ and $150 \leq \texttt{i} \leq 300$ correspond to the specified program behaviour. The third case marks an erroneous system behaviour. On first sight, every possible value of i that can be set in the main program is covered by the first two cases and and the *Failure* in line 27 should be unreachable.

However, if we employ the program in an production environment, we will eventually find out that this assumption is wrong and that the failure can occur. Independent of the failure consequence, the developer needs to debug the program to locate and correct the existing defect. A difficult part at the beginning of the debugging process consists of reconstructing the failure. In our example the task is to search for a program execution where the interrupt handler is entered and variable `i` is greater than 300. A starting point might be the test

```
 1  #include <avr/io.h>
 2  #include <avr/interrupt.h>
 3  #include <avr/signal.h>
 4
 5  volatile int i=0;
 6
 7  int main (void){
 8    DDRA = 0xFF;
 9    PORTA = 0xFF;
10    GICR = GICR | (1<<INT2);
11    sei();
12
13    while (1){
14      while (i<300){
15        i++;
16      }
17      i = 0;
18    }
19  }
20
21  SIGNAL (SIG_INTERRUPT2){
22    if (i < 150)
23      PORTA = 0x00;
24    else if (i <= 300)
25      PORTA = 0x55;
26    else
27      FAILURE;
28  }
```

```
 1  #define q ({i} >= 0) && ({i} <= 300)
 2  ![] (interrupt18_entered -> q)
```

Listing 6.1: Source code and specification of the case study

whether variable `i` is reset once a value of 300 is reached. The developer can, for example, set a watchpoint to suspend execution when `i` reaches 299 and then use source-level stepping to advance execution. As expected the variable is reset to 0. A conceivable idea is then to trigger the interrupt and inspect the variable value after the interrupt handler is entered. Again, we would like to carry out this test when the value of `i` is 299.

In this particular program we could make use of the already existing watchpoint and issue a forward resume operation. This would bring us to the same program state that has been observed earlier. While this possibility exists in the simple example program, this is usually not true for real-life programs. There we would have to restart the program and navigate to the desired program state. Fortunately this is not necessary when using the ESD debugger debugger; instead of restarting the program we can navigate backwards by employing reverse stepping or a reverse resume operation. Having reached the desired program state we can make use of the trace view's successor component (see Section 4.2.4) to explicitly trigger the interrupt and inspect the variable value after the interrupt handler has been entered; as expected it is still 299. The program is behaving as it is supposed to and we do not known yet, how the failure can occur. At least, the developer might have realized that the source-code level view is not sufficient to reason about the failure and its causing defect. The machine-level or more particularly the disassembly view, which shows the executed machine instructions, should be in focus of our analysis next.

A good starting point for this investigation is probably the increment instruction in line 15. We can summarize the knowledge the user gains by inspecting the provided disassembly information by stating that the ATmega16 implements a value assignment on an integer variable by using two `sts` instructions. However, the technical details of how the increment operation is realized on the machine instruction-level is more important than just a summary and we need to discuss this aspect more throughly. The assembly code that corresponds to the `i++`

operation is therefore listed below:

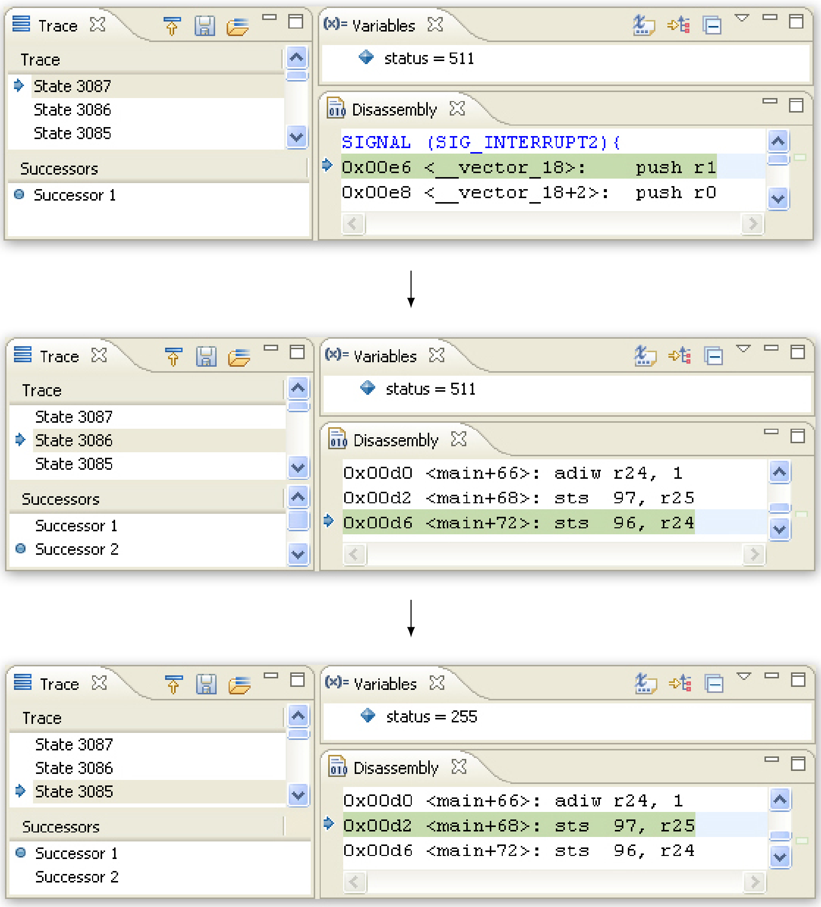
```
c8:     80 91 60 00    lds   r24, 97
cc:     90 91 61 00    lds   r25, 98
d0:     01 96          adiw  r24, 1
d2:     90 93 61 00    sts   98, r25
d6:     80 93 60 00    sts   97, r24
```

The first two instructions load the current value of `i` from SRAM in the register pair R24 and R25. In the following the value contained in the register pair is incremented by one using the `ADIW` instruction. Register R24 and R25 then hold the new value of variable `i`. The last step includes storing this value back to SRAM which is achieved by the mentioned `sts` instructions. The first `sts` instruction saves the high-order byte whereas the second instruction saves the low-order byte.

Although the increment operation appears to be atomic on the source-level, it consists of several operations on the instruction-level. In the given example program an interrupt can occur between any two of these instructions. In case the interrupt occurs after the first `sts` instruction, the high-level byte of the integer variable might have been modified, whereas the low-level byte has not been updated yet. If variable `i` has a value of 299 = 0x012B the situation is not dramatic; if the interrupt is triggered between the `sts` instructions, the interrupt handler is entered, and `i` still has a value of 299 = 0x012B. However, the situation is different when `i` has a value of 255 = 0x00FF. Here the increment instruction must alter the high-order byte to 0x01 and then reset the low-order byte to 0x00, resulting in a value of 256=0x0100. In case the instruction sequence is interrupted after the high-order byte has been written to SRAM, the interrupt handler is entered and then finds a value of 511=0x1FF for variable `i`.

While understanding the failure and its cause is very difficult, the solution is quite simple: It is only necessary to deactivate all interrupts before the increment operation is executed and activate them later one. This can be done by using the `cli()` and `sei()` assembly instructions. We have seen that the ESD debugger greatly supports the debugging process and lets developers concentrate on the information they need. The developer is able to quickly move forward and backwards in an execution trace and also explore alternative path for program continuation. But the success of the debugging activity depends, for the most part, on the experience and endurance of the developer. More problematic is the fact, that the debugging activity is usually started too late, that is, once the failure has been observed in the production environment.

Fortunately, we are able verify the program's correctness by means of the model checking method. The specification from Listing 6.1 states that every time the interrupt handler is entered the value of variable `i` is in the range from 0 to 300. As we already known the example program violates this specification and the model checking algorithm terminates with a counterexample. Understanding the failure and localizing its cause is greatly simplified



```
State 3087
State 3086
State 3085
Successor 1

status = 511

SIGNAL (SIG_INTERRUPT2){
0x00e6 <__vector_18>:    push r1
0x00e8 <__vector_18+2>:  push r0
```

```
State 3087
State 3086
State 3085
Successor 1
Successor 2

status = 511

0x00d0 <main+66>: adiw r24, 1
0x00d2 <main+68>: sts  97, r25
0x00d6 <main+72>: sts  96, r24
```

```
State 3087
State 3086
State 3085
Successor 1
Successor 2

status = 255

0x00d0 <main+66>: adiw r24, 1
0x00d2 <main+68>: sts  97, r25
0x00d6 <main+72>: sts  96, r24
```

Figure 6.1: Using the ESD debugger to locate the failure cause

when failure information exists in form of a generated counterexample. The ESD debugger is able to load the counterexample and the contained execution trace is displayed in the trace view. It is advisable to jump to the last state and from there on navigate backwards to localize the defect. In the example program it is sufficient to employ backwards stepping to locate the defect, as the location of the failure and its cause are quite close to each other. In real-world examples the developer can also consider the use of breakpoint to analyse the counterexample. Figure 6.1 gives an impression of how this analysis is done with the ESD debugger.

The strength of our debugger is to bring the programmer to the point where he understands the failure causes and is able to come up with possible corrections. Equally important is the connection to a verification tool such as the employed model checker, as the debugger is not able to verify program correctness.

7 Technical Realisation

This chapter presents implementation aspects of the embedded systems debugger. The ESD debugger has been developed in the Java^{TM} programming language and consists of a set of plugins for integration into the Eclipse platform. Besides being the target environment for our tool, Eclipse has also been used as environment for plugin development. In particular the Plugin Development Environment (PDE) has been employed to create and edit the configuration files associated with each plugin. In order to achieve a separation between user interface specific aspects and underlying business logic the functionality of the ESD debugger is contributed by two separate plugins, namely the *es.debug.core* and *es.debug.ui* plugin. The complete system consists of 19 interfaces, 198 classes and has a size of more than 10000 lines of code. The exact distribution between both plugins is given in Table 7.1.

A high-level overview of the system architecture is given in Figure 7.1. The diagram is an intermediary between UML component and package diagram and we believe that this form is well suited to allow readers an intuitive perception of our implementation. The functionality within each plugin is divided into smaller units and is kept together as Java packages. The complete package name is made up from the plugin name (e.g *es.debug.core*) and the package identifiers as specified in Figure 7.1. The following paragraphs will analyse the system architecture in more detail. The outline of our presentation is based on the plugins and the

Plug-in	Interfaces	Interfaces %	Classes	Classes %	**LOC**	LOC %
es.debug.core	15	78%	123	62%	**7321**	71%
es.debug.ui	4	22%	75	38%	**2967**	29%

Table 7.1: ESD plugins: Interfaces, Classes and Lines of code

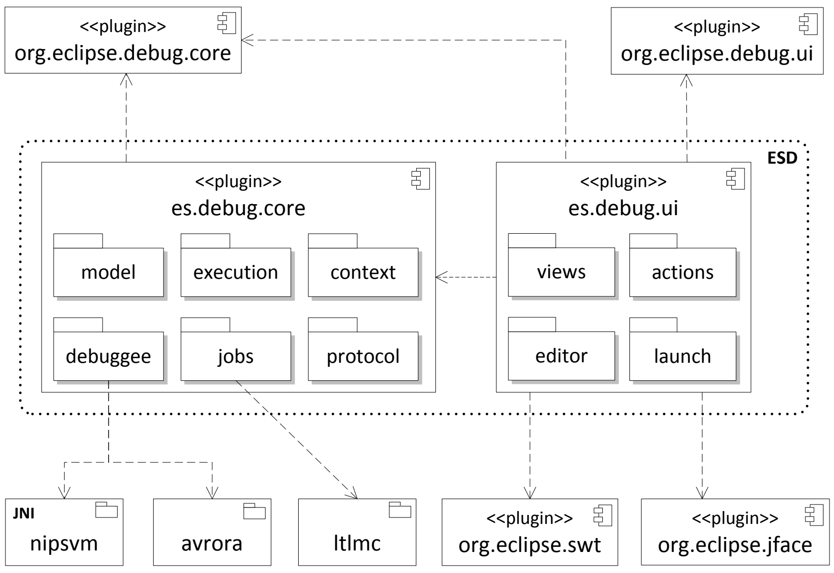


Figure 7.1: ESD - Plugins, Packages and Dependencies (Arrows)

contained packages.

7.1 Core Plugin

Model The ESD Debug Model as described in Section 4.1 is implemented by the classes contained in the *model* package. The debug model elements define a view of the program under debug.

Execution The *execution* package is concerned with implementation aspects focusing on execution control. As most of the respective business logic is provided by debug model elements this package has been designed as gathering place for items that do not fit into the model package. These items include classes that implement breakpoints and also the *ESLaunchDelegate* class which takes care of launching the debuggee.

Context The purpose of the *context* package is to hold all business logic that is affiliated with the provision of program context information but is not covered by debug model elements. More precisely, the context package takes care of the source lookup mechanism and provides a source locator which is employed to map stack frames to file names (see section 4.3.3). The implemented source locator does not take care of the stack frame to file name mapping itself but delegates this task to a so-called lookup participant. The source locator is able to manage multiple lookup participant and by this means provides support for environments that contain different kind of stack frames. The technical details for mapping a stack frame to a file name might differ for each frame type and by this means is encapsulated within a particular lookup participant. The ability to manage multiple frame types is, for example, required by multi-language environments. Section 9.1 will give an outlook why multi-language debugging becomes important in the context of this work.

Debuggee The implementation of the debuggee component is established by the debuggee package. Classes of primary importance are the *Backend*, *TraceManager*, *StateAnalyser* and *DebugInformationProvider* class.

The main task of the *Backend* class is to communicate with the frontend and to make sure that necessary steps are taken whenever a request is received. Requests can be divided into two types and are either concerned with controlling the program's execution (1) or target the retrieval of context information (2). The backend delegates the work necessary to handle requests to either the *TraceManager* or the *StateAnalyser*. Requests of the first type are transmitted to the *TraceManager* whereas the second type is forwarded to the *StateAnalyser*. A minor job of the backend consists of several initialization tasks including the creation of a *TraceManager* and *StateAnalyser* object for later use. Furthermore, the initialisation procedure comprises the generation of the simulator component, that is, the NIPS VM. The NIPS VM is implemented in the C programming language and is contributed to our project as native, that is platform-specific, library. Access to a native library from a Java environment is not possible in an ad-hoc manner but requires the development of a Java Native Interface (JNI) layer [Lia99]. The MCESS project has been required to concentrate on this aspect at an earlier point in time and the results of this effort can be found in the *nipsvm* package. We have been able to reuse the existing JNI layer in our work. This also explains the dependency to the *nipsvm* package.

The *TraceManager* class is the counterpart to the trace element included in the debug model and implements the business logic required for execution control. The trace manager provides complex functionality such as source-level stepping, breakpoints, or reverse execution on top of the NIPS VM. The NIPS VM itself provides very limited capabilities with respect to execution control and only permits the calculation of direct successors for a given input state. For the time being we can ignore the state format used by the virtual machine and refer to these states as microcontroller state representations. From this view point a direct successor corresponds to the microcontroller state obtained from the input state by executing a single microcontroller machine instruction. Implementing high-level execution control in this environment can either be achieved by enhancing the NIPS VM with more complex control capabilities or by simulating the required functionality. Simulation in this case means that successor states are repeatedly generated until a certain stop condition such as hitting a breakpoint is met. This approach infers a severe performance penalty caused by the overhead for running a continuous state analysis. An enhancement of the NIPS VM to support high-level execution control internally is presumably a better choice with respect to the debugger's overall performance. However, the necessary enhancements require substantial modification to the NIPS VM implementation. An initial effort estimation indicated that the required development time would violate the time frame given by this project and we decided to base our implementation on the simulation approach. The *TraceManager* implementation is therefore characterised by making intensive use of the NIPS VM successor function. Generated successor states are examined by means of the *StateAnalyser* to inspect whether the state generation process should be interrupted due to a satisfied condition such as the completion of a source-step operation or hitting a breakpoint. A list data structure is used to keep track of the established execution trace. The list does not contain complete states but rather state identifiers which are calculated up-front. The usage of state identifiers is important with respect to an efficient exchange of trace date between the debuggee and debugger component.

The *StateAnalyser* is responsible for providing access to the internal of a microcontroller state including the current instruction address, values of variables and registers or the program call stack. For this task the *StateAnalyser* requires information about the microcontroller's memory layout and its mapping to a NIPS byte-buffer state. The necessary information is provided by memory layout classes, for example the *ATmega16MemoryLayout* class, contained in the *microcontroller* sub-package. The provided functionality is requested by the *TraceManager* to decide whether further successor states need to be generated. It is also required by the *Backend* for the retrieval of program context information. The most complex operation carried out by the *StateAnalyser* is unwinding the call stack as described in Section 4.3.2. Being a non-trivial task from the start, it has been further complicated by the fact that the semantics of interrupt calls are not completely preserved by the toolchain responsible for generating the NIPS bytecode. In case an interrupt handler is called the stack pointer is correctly decremented, however, the value of the return address is not written on the stack. A first version of our implementation solves this problem by referring to the interrupt stack (see Figure 2.2) as provided by a NIPS state. We are able to determine the missing return addresses by iterating over the interrupt stack and retrieving the current instruction address for the NIPS processes found on the interrupt stack.

The main task of the *DebugInformationProvider* class is to extract and manage the debug information contained in the executable file (see Section 4.3.1 for background information). The *DebugInformationProvider* does not directly work on the executable but instead on the output produced by the tool objdump. The objdump tool is able to display various information about executables and has been used by the MCESS toolchain (see Section 2.3) as disassembler to view the executable in assembly form. Our motivation for using the objdump tool lies in its ability to extract the required debug information sections, namely the *stabs* and *stabstring* section. The parser which has been developed to extract the contained stab instructions is provided by the *stabs* sub-package. During the initialization of the debuggee, the *DebugInformationProvider* iterates over the stab instructions provided by the parser and stores the obtained information into internal data structures. Due to the already existing dependency to the executable or rather the objdump output we decided to extract further information provided by this file, that is disassembly information. The processing of stabs directives that hold information about existing functions therefore includes the retrieval of the associated machine instructions. The instruction are at this point in plain text format which is fine for presenting disassembly information in the user interface. However, analysing the machine code with respect to the stack unwinding preprocessing step is simplified if instructions exists in as Java class representation. The Avrora project provides a Java API for analysing assembly code written for AVR microcontrollers and has been employed for mapping the plain text instructions to Java objects.

Jobs The primary functionality of the jobs package is to provide the ability to launch model checking operations. Responsibility for this task lies with the *ModelCheckingJob* class which integrates model checking algorithms that are supplied by the *ltlmc* package. The *ltlmc* package is part of the MCESS project and implements algorithms such as the discussed NFDS algorithm or Couvreurs algorithm (see section 5.3). As discussed by Clayberg in [EC08] long running jobs should be executed in the background so that the user interface stays responsive. For this reason we have based our implementation on the Eclipse *Jobs API* which presents a guideline for creating, managing and displaying background operations. The jobs package is complemented by the ability to save and load execution traces. This capability allows users to analyse executions at a later time but simultaneously is also a prerequisite to allows our debugger to operate on model checking counterexamples.

7.2 Communication Protocol

The ESD debugger's communication protocol is currently implemented as collection of Java classes which are contained in the *protocol* package of the *core* plugin. As discussed in Section 4.1 our communication protocol distinguishes between commands and events. Commands are used by the debug model elements to control the execution of the debuggee or to retrieve program context information. The necessity to issue commands originates at a different place, namely the user interface. As result of a user interaction, such as selecting a previous state in the trace view or by clicking the resume button in the debug view the user interface interacts with a corresponding debug model element to change the status of the debuggee. Alternatively, upon receiving notice about a status change, the user interface components update and consequently use the debug model elements as intermediary to retrieve the required program context information. Communication by means of commands is carried out in a synchronous manner. This means that after a particular debug model element has issued a command it blocks and waits to receive the debuggee's reply message.

This is fine for all commands asscociated with the retrieval of program context information, as the respective operations take an insignificant amount of time. However, for those commands concerned with execution control, for example, a step or a resume command, the time necessary to complete the operation is unknown. In the worst case the required operation will not finish at all. As consequence, the user interface becomes unresponsive for a certain amount of time or even completely freezes. To solve this problem we have designed the already mentioned event channel. Instead of having the debug model element wait until the particular control operation finishes, commands targeting execution control are acknowledged immediately so that the user interface remains responsive. Events describing the current status of the debuggee are send asynchronously to keep the debug model elements and consequently the user interface components informed.

In the following, we will give a short overview of the existing commands and events that make up our debug protocol. Afterwards we will give an example how the individual elements, that is user interface, debugger and debuggee interact.

7.2.1 Execution control commands

The following commands instruct the debuggee process to change its current execution state. They return immediately and do not wait until the corresponding operation is finished.

Suspend [all | thread id] Suspends execution of the debuggee or a selected thread.

Resume [all | thread id] Resumes execution of the debuggee or a selected thread.

Terminate Instructs the debuggee to terminate.

Step [thread id] [level] [direction] [successor id] Distinguishes between forward and backward stepping as well as source-level and instruction-level stepping. Executes the next source- or machine-instruction or reverses the last source- or machine-instruction.

SetBreakpoint [source file] [line] Sets a breakpoint on the given line.

ClearBreakpoint [source file] [line] Clears any breakpoint on the given line.

InsertTraceState [trace index] [state id] Inserts the given state identifier after the specified index in the execution trace.

SelectTraceState [trace index] Selects the execution state at the specified index within the current execution trace.

7.2.2 Context information retrieval commands

The commands listed below are employed to retrieve specific program context information. For most of these commands a corresponding helper class is exists that can be instantiated to hold the information to be returned. These classes are contained in the *results* package.

Thread Retrieve the name and identifier of all existing threads.

Stack [thread id] Retrieve a list of stack frames for the given thread identifier.

Frame [thread id] [frame id] Get frame information for the given thread and frame identifier.

StackDepth [thread id] Query the debuggee for current frame count.

Trace [trace index] Retrieve the execution trace (list of state identifiers) starting from the given index.

Sucessors [state id] Returns the successor states (their identifiers) for the specified state.

Disassembly [frame id] Retrieves disassembly information for the given frame.

Memory [start] [length] Queries the debuggee for the microcontroller's memory content.

RegisterGroups Asks for existing register groups such as GPR or SPR.

Registers [group id] Returns a list of registers (name and identifier) for the given register group.

Register [register id] Get the value for given register.

Variables [frame id] Retrieve the existing variables for the given frame.

Variable [variable id] Return the value for the specified variable.

7.2.3 Other commands

SaveTrace [file path] Instructs the debuggee to save the current execution trace.

RemoveTraceFuture [trace index] Removes the execution trace starting at the given index.

7.2.4 Events

Started The debuggee started event is generated once the initialization tasks are successfully completed. The start of a new thread is communicated by means of the thread started event.

Resumed Either a single thread or all threads resumed execution.

Suspended A single thread or all threads are suspended.

Terminated The debuggee or a single thread terminated.

TraceChanged The execution trace changed.

7.2.5 Communication Example

The interplay between the user interface, the debug model elements and the debuggee component is best explained by an example. We have choosen the step operation for this purpose. Before we go on, we would like to encourage our readers to have a look at the sequence diagram of the step operation as depicted in figure 7.2.

When the user issues a step action, either by clicking on the corresponding button or by selecting a successor state in the trace view, the user interface retrieves the correct *ESThread* object and informs it about the required step operation. In the following, the *ESThread* object communicates with the *ESTrace* object to retrieve the currently selected trace index and successor state. The *ESThread* object takes further parameters such as the step-level or the step-direction (forward vs. backwards) into account and then issues a particular *Step* command with help of the *Frontend*. The command is immediately acknowledged by the *Backend* to make sure that the user interface stays responsive. A *resume* event is sent shortly after receiving the *Step* command to let the debug elements, particularly the specific *ESThread* object, know that the debuggee has resumed execution. The event is actually not directly sent to the debug model elements but instead to the *Frontend*. The *Frontend* informs all registered listeners, amongst others the particular *ESThread* object. The *ESThread* object will then fire an internal event to inform the user interface components about the debuggee's status change. At some point later the step operation finishes and an *suspended* event is sent. Again, the *ESThread* object receives a message from the *Frontend* and informs the user interface components. Upon notice about the debuggee's suspension, further action is required by the *ESTrace* object: A *Trace* command is send to receive information about all state identifiers that have been added to the execution trace. After the *ESTrace* object has updated its internal data structure a message is send to the trace view to trigger an refresh.

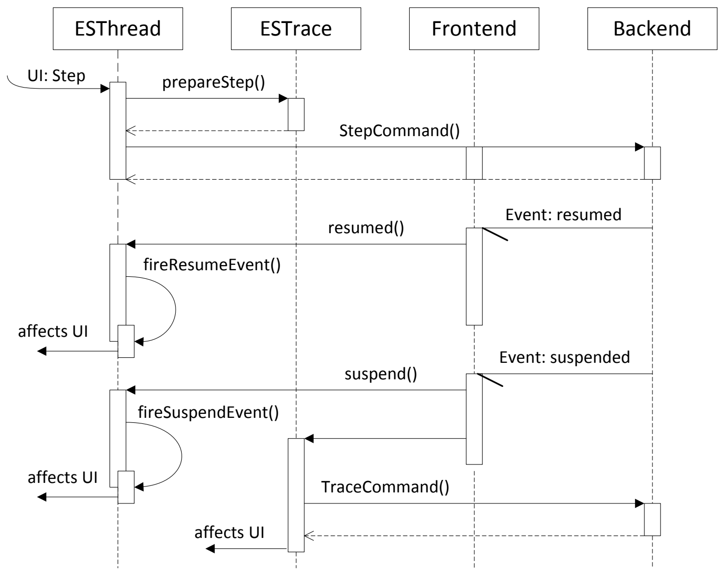


Figure 7.2: Sequence Diagram: Step Operation

7.3 User Interface Plugin

Views The *views* package complements the debug view implementations that are part of the debug project and provides the trace view and disassembly view implementation. Furthermore, it includes the implementation of the model checking status view.

Actions The user interface components rely on the classes contained in the *actions* package to employ functionality provided by the debug model elements.

Editor A C source code editor is included in the *editor* package.

Launch The *launch* package contributes the user interface required for launching embedded systems applications.

8 Related Work

8.1 NanoEsto Debugger

The development of the NanoEsto Debugger by Chun and Lim [CL06] takes place in the context of designing and validating wireless sensor networks. A *wireless sensor network* consists of multiple sensor nodes that cooperatively monitor physical or environmental conditions and as states by Dyer [Dye07] can be regarded as an example of a distributed embedded system. Sensor networking applications are typically developed on top of a special operating systems, whereas TinyOS [HSW^{+}00] is probably the most popular platform being used. Being unsatisfied with the existing platforms the Electronics and Telecommunications Research Institute (ETRI) has developed an OS named NanoQPlus and an Eclipse-based development environment named NanoEsto to support the development of NanoQplus applications. An important aspect of the NanoEsto development environment is the provision of an embedded systems debugger based on the on-chip debugging approach as discussed in Section 1.2.2. The debugger's architecture is depicted in Figure 8.1 and consists of three components, that is the host system, an JTAG emulator and the target system which runs the application to be debugged. The debugger's user interface is based on the CDT project (C/C++ Development Tools) [1] and works against the debugging engine. The debugging engine is build on the GNU debugger and employs the JTAG debugging agent in order to translate GDB commands to JTAG instructions. The resulting instructions are transfered to the JTAG emulator via an USB connection and are converted to corresponding JTAG signals. The generated JTAG signals establish the connection to the target system. The described JTAG emulator is an internal development based on the ATmega16, whereas the target system is a sensor board featuring

[1] http://www.eclipse.org/cdt/

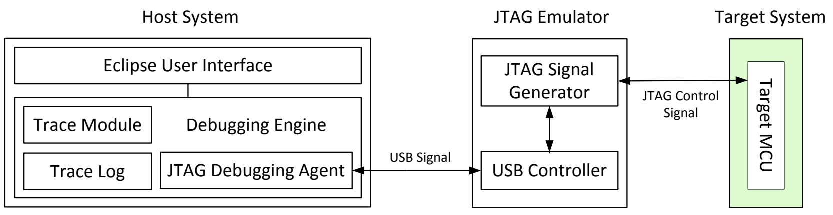


Figure 8.1: Architecture of the NanoEsto Debugger [CL06]

an ATmega128. Unfortunately the authors do not provide a case-study and as most of their research is contributed in Korean we have not been able to follow their work in more detail.

8.2 Model Checking with the GNU debugger

In contrast to our work where an existing model checking framework, namely the NIPS VM, is used to build a source-level debugger the work by Mercer and Jones [MJ05] follows the opposite direction. They present a model checker for embedded systems software which employs the GNU debugger as state-space generator. In principle it does not matter whether the debugger works against the native hardware or an instruction-set simulator, however, the employment of native hardware is generally more difficult as we will see shortly.

The model checking process is started by passing the binary executable that corresponds to the program of interest to the debugger. The debugger is then responsible for executing the program and eventually stops to let the model checker read the current microcontroller state. An advantage of this approach lies in the fact, that the state generation granularity is flexible: The debugger can be told to return new states after every source statement, machine instruction, user-defined breakpoints or a combination of the former possibilities. While reading the current program state with the help of the debugger is quite simple this does hold for writing a microcontroller state. Writing a microcontroller state becomes necessary in case the model checking algorithm starts backtracking or an environment interaction is evaluated. It is the existence of read-only registers (counter values) and clear-on-write registers (interrupt flags) that makes this task an additional burden. In case the debugger works on a simulator the problem can be solved by modifying the simulator in order to make read-only and special control registers writable. This approach is not possible though when the debugger runs on the native hardware. Instead the microcontroller state has to manipulated with great care to achieve the desired result. Setting an interrupt flag, for example, cannot be done directly as it is automatically cleared. Instead the model checker needs to create a state that causes the interrupt to fire, for example, by toggling the external interrupt pin.

By using the GNU debugger as state generator Mercer and Jones have been able to build a model checker that considers the targeted hardware. The implemented verification algorithm is limited to general properties such as stack overflow or read-only violation and does include the verification of temporal logic formulas.

9 Summary and Perspective

In this work we have developed a source-level debugger for embedded systems software based on a virtual machine for state-space generation, namely the NIPS VM. We have used the toolchain created by the MCESS project [Roh06] to translate the source program to a required NIPS bytecode program and also discussed how debug information create a connection between both artifacts. Considering the fact that the toolchain takes a processor model specified in a hardware description language as input our debugger supports different hardware architectures. By integrating the debugger into the Eclipse platform we have achieved a seamless integration into the software development process.

The debugger's functionality concentrates on the ability to control the program's execution and to inspect program context information. Execution control includes means to suspend execution at the occurrence of user-specified conditions and to continue execution step-wise, either on the source- or the instruction-level. Being able to reverse execution to an earlier point in time is a more advanced execution control concept provided by the ESD debugger and is based on the its ability to record the program's execution. Inspecting program context information is possible on two levels, that is, the source- and/or the instruction-level. Possible information includes the current source statement, the current machine instruction, the program call stack, variable and register values and memory content.

We discussed that debugging is concerned with the localization and correction of known failures but does not make a statement about system correctness. Model checking on the other hand is able to prove or disprove system correctness by exhaustive state-space exploration. We have combined an existing model checker for embedded systems software with our debugger to reason about counterexamples. Our experience shows that this is a very powerful combination, especially when both activities, that is, debugging and model checking are integrated closely into an development environment.

9.1 Open Issues and Future Work

Tracing

The ability to record the microcontroller's execution is one of the greatest strength of the ESD debugger and builds the foundation for reverse execution. However, the mechanism for program recording is at an early development stage and requires improvements. Currently, the recording mechanism is set to record the microcontroller state after every machine instruction. Although this fine granularity is beneficial with respect to a successive trace analysis, it results in a performance penalty and also high memory requirements. A first improvement should include the ability to record less information, for example, to record the microcontroller state only on method entry and exit, access to specified I/O or memory locations or other user-specifiable conditions. Additionally, the recording mechanism should be able to remove the oldest recorded program states on demand to manage with existing memory limitations. An in-depth discussion of possibilities to improve program recording functionality can be found in [Lew03]. A large amount of research on program monitoring or recording, respectively, has also been invested by the runtime verification community. A good introduction to this research can be found in the work by Watterson [WH07] or Havelund [Hav08].

At the same time, the recorded trace information are not used to the full extend yet. Further improvements target both trace visualisation and trace navigation. Regarding further trace visualisation capabilities, it should be possible, for example, to employ a time-series analysis for specified variables, memory locations or I/O locations and inspect how their values evolve during program execution. When the developer hovers over a trace state a balloon help could present identifying information such as the respective source line and machine instruction. Furthermore, the balloon help could be complemented by highlighting the differences to the previous state.

Navigating the recorded trace could be further improved by providing a trace query language, which allows to query for interesting events, such as read or write access to a variable or memory location, and lists all program location that apply.

Microcontroller Environment

The behaviour of the program under study depends heavily on the environment the embedded system operates in. In the current setting it is possible to specify additional NIPS bytecode snippets which are executed on access to specified memory locations and by this means simulate the environment (see Section 2.5 for details). However, the development of an environment model is quite tedious in the current setting and improvements are in order. Changing the specification language from the low-level NIPS instructions to a powerful scripting language such as JavaScript or TCL would simply the specification process to a great extend. An inspiration might also be the work by Schlich et al. [BSK08] that investigates the usage of finite automata as possible formalism for environment specification.

Multi-Layer Debugging

So far we have concentrated our development effort on the perspective of an embedded systems application developer and his need for validating and debugging a particular application. We have employed an existing virtual machine, namely the NIPS VM as instruction-set simulator to execute the application under study and to provide debugging as well as model checking functionality. Until now we have implicitly assumed that the NIPS VM as well as the featured toolchain correctly define the program's operational semantics. Although the NIPS VM as well as the toolchain have been extensively tested, chances are that both components still contain defects that might become apparent when debugging an embedded systems program. A first step to support the maintainer of the NIPS VM and the featured toolchain with the task of localizing defects is to provide means to inspect the executing program at a deeper level, that is, the NIPS bytecode level. Displaying the executed NIPS bytecode, highlighting the current NIPS instruction with an instruction pointer and providing a special visualisation of the current NIPS byte-buffer state describes this step in a more explicit way. The visualisation of the current NIPS byte-buffer state has been mentioned in Section 4.3.6 and resulted in the provision of special memory rendering.

Further insight into the virtual machine's operation can be obtained by providing support for stepping on the level of individual NIPS instructions and displaying the temporal operand stack. Both, the inner workings of the virtual machine and it's interface would require major changes to support instruction-level stepping. The functionality described so far will give a better understanding of the executed NIPS bytecode, however, provides no help with defects that manifest themselves at the virtual machine's implementation level or the Java interface layer. Essentially we are looking at the task of debugging a mixed-mode application consisting of Java, C and JNI code. A good starting point is the work by Chauvin [Mar07] or Jancewicz [Nic06].

List of Figures

Bibliography

[AK02] Sanjeev Kumar Aggarwal and M. Sarath Kumar. Debuggers for programming languages. In *The Compiler Design Handbook*, pages 295–328. CRC Press, 2002.

[Atm05] Atmel. *http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf, ATmega16 Complete Datasheet*, 2005.

[Atm09] Atmel. AVR JTAG ICE, 2009. Available from: `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2737` [cited 6th April 2010].

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1 edition, 1996.

[BSK08] Dominique Gückel Bastian Schlich and Stefan Kowalewski. Modeling the environment of microcontrollers to tackle the state-explosion problem in model checking. In *In Proceedings of Symposium Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2008)*, 2008.

[CL06] I. Chun and C. Lim. NanoEsto Debugger: The Tiny Embedded System Debugger. In *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference*, volume 1, 2006.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal methods in system design*, 1(2):275–288, 1992.

[Dye07] Matthias Dyer. *Distributed embedded systems*. PhD thesis, Eidgenössische Technische Hochschule ETH Zürich, 2007.

[EC08] Dan Rubel Eric Clayberg. *Eclipse: building commercial-quality plug-ins*. Addison-Wesley, 3 edition, 2008.

[Ecl10a] Eclipse Foundation. Eclipse Debug Project, 2010. Available from: `http://www.eclipse.org/eclipse/debug/index.php` [cited 6th April 2010].

[Ecl10b] Eclipse Foundation. Eclipse Platform, 2010. Available from: http://www.eclipse.org/ [cited 6th April 2010].

[gdb09] GDB: The GNU Project Debugger, 2009. Available from: http://www.gnu.org/software/gdb/ [cited 6th April 2010].

[GMZ04] P. Gastin, P. Moro, and M. Zeitoun. Minimization of counterexamples in spin. In *Proc. 11th SPIN Workshop, LNCS 2989*, pages 92–108, 2004.

[GS09] Andreas Gaiser and Stefan Schwoon. Comparison of algorithms for checking emptiness on büchi automata. In *Proceedings of the 5th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS)*, pages 69–77, Znojmo, Czechia, November 2009.

[Hav08] K. Havelund. Runtime verification of C programs. *Testing of Software and Communicating Systems*, pages 7–22, 2008.

[HPY96] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32. American Mathematical Society, 1996.

[HSW$^+$00] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[Kow06] Stefan Kowalewski. *Safety and Reliability Engineering (Vorlesung)*, 2006.

[Lew03] Bill Lewis. Debugging backwards in time. In *In 5th Workshop on Automated and Algorithmic Debugging (AADEBUG)*, 2003.

[Lia99] Sheng Liang. *The Java native interface: programmer's guide and specification*. Addison-Wesley, 1999.

[Mar06] Peter Marwedel. *Embedded system design*. Springer, 2006.

[Mar07] Mariot Chauvin. Support seamless debugging between JDT and CDT, 2007. Available from: http://wiki.eclipse.org/Support_seamless_debugging_between_JDT_and_CDT [cited 6th April 2010].

[MJ05] Eric Mercer and Michael Jones. Model checking machine code with the gnu debugger. In *SPIN*, pages 251–265, 2005.

[MKM93] J. Menapace, J. Kingdon, and D. MacKenzie. *The "stabs" debug format*, 1993. Available from: http://gnuarm.org/pdf/stabs.pdf [cited 6th April 2010].

[Nic06] Nick Jancewicz. Java and C/C++ JNI Application Debugging - The GUI Way, 2006. Available from: http://www.kineteksystems.com/white-papers/mixedjavaandc.html [cited 6th April 2010].

[Ora10] Oracle Corporation. Java Platform Debugger Architecture, 2010. Available from: http://java.sun.com/javase/technologies/core/toolsapis/jpda/ [cited 6th April 2010].

[Roh06] Michael Rohrbach. Ein ansatz zum model-checking von software für eingebettete systeme. Master thesis, Embedded Software Laboratory, RWTH Aachen, 2006.

[Ros96] Jonathan B. Rosenberg. *How debuggers work: algorithms, data structures, and architecture*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[RW04] J. des Rivieres and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43, 2004.

[SE05] S. Schwoon and J. Esparza. A Note on On-the-Fly Verification Algorithms. In *Proc. of TACAS'05, LNCS*, pages 174–190. Springer Verlag, 2005.

[SM04] Inc. Sun Microsystems. *Stabs Interface*, 2004. Available from: http://developers.sun.com/sunstudio/documentation/ss11/stabs.pdf [cited 6th April 2010].

[vR07] Riemer van Rozen. A debugging framework for nips. Master thesis, Formal Methods and Tools Group, University of Twente, Netherlands, 2007.

[VW86] M.Y. Vardi and P. Wolper. Automata theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32(2):182–221, 1986.

[WH07] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *Software, IET*, 1(5):172–179, 2007.

[Wri04] Darin Wright. *How to write an Eclipse debugger*, 2004. Available from: http://www.eclipse.org/articles/Article-Debugger/how-to.html [cited 6th April 2010].

[WS05] Michael Weber and Stefan Schürmans. A virtual machine for state space generation. Diploma thesis, Lehrstuhl für Informatik II, RWTH Aachen, Germany, 2005.

A Appendix

A.1 Selected Extensions

A.1.1 Launch configuration type

```
<extension
   point="org.eclipse.debug.core.launchConfigurationTypes">
   <launchConfigurationType name="Embedded System Software"
      id="es.debug.core.ESLaunchType"
      modes="debug"
      delegate="es.debug.core.launcher.ESLaunchDelegate">
   </launchConfigurationType>
</extension>
```

A.1.2 Line Breakpoint

```
<extension point="org.eclipse.debug.core.breakpoints">
  <breakpoint
     class="es.debug.core.breakpoints.ESLineBreakpoint"
     name="ES Line Breakpoints"
     markerType="es.debug.core.markerType.lineBreakpoint"
     id="es.lineBreakpoint"/>
</extension>
```

A.1.3 Marker type

```
<extension point="org.eclipse.core.resources.markers"
      id="markerType.lineBreakpoint"
      name="ES Line Breakpoint Marker">
   <super type="org.eclipse.debug.core.lineBreakpointMarker"/>
   <persistent value="true"/>
</extension>
```