DWARF-driven Equivalence Checking of UML Statecharts and Software Components

Patrick Heckeler, Jörg Behrend, Jürgen Ruf, Thomas Kropf, Wolfgang Rosenstiel
Eberhard Karls University, Computer Engeneering Department,Tübingen, Germany
{heckeler,behrend,ruf,kropf,rosenstiel}@informatik.uni-tuebingen.de

Roland Weiss
ABB Corprorate Research, Industrial Software Systems, Ladenburg, Germany
{roland.weiss}@de.abb.com

1 Abstract

This article presents an instrumentation-free runtime verification methodology built upon an external observer which uses DWARF[1]-statements to monitor system behavior. The observer delivers information about variable values used for state-encoding and method calls representing transitions. These information are passed to an engine which parses the system specification in terms of a UML statechart. It is transformed into an executable automaton which acts as a *golden reference* for equivalence checking. The presented approach makes it possible to perform verification directly on the target architecture and keeps up test significance by avoiding modification of the executable caused by injected monitors or other code probes.

2 Introduction

For years hardware-dominated embedded systems have been the preferred choice to build safety-critical systems. But many fields of application, for example the area of industrial automation, have been entered by software-dominated systems [5]. The main reasons for that are cost reduction and flexibility enhancement. Due to immense complexity of those systems, formal verification has reached its limits [6]. Therefore, runtime verification (RV) approaches [7] often come into action.

Chen et al. [2] introduce MOP, a framework following the concept of *monitoring-oriented programming* which focuses the checking on Java programs: For a *software under test* (SUT), in that case a Java class, a corresponding monitor is generated (in Java as well) and automatically compiled and linked into the whole program. Monitors, a kind of a code probe, are synthesized from formal properties and are integrated into the program under test. These monitors verify the correct behavior of the SUT during runtime but thereby the SUT is modified. Pintér and Majzik [10] perform a concurrent monitoring of applications specified by UML statecharts. They propose the integration of a *watchdog processor* which is able to detect deviations from the control flow throughout the entire runtime. This includes initialization, event processing and firing correct transitions of a UML statechart specification. Their approach relies on manual high-level source-code instrumentation to gather the mentioned data needed for RV. A major drawback of both approaches is the necessary instrumentation of the executables. While Chen et al. perform the instrumentation automatically, Pintér and Majziks' approach even relies on manual high-level instrumentation which is a time-intensive and error-prone task. In both approaches the SUT is modified and therefore, program execution is influenced. For example the call stack differs when injected verification routines are executed. This avoids applying techniques like *call stack coverage* for test-suite reduction [9]. Also common coverage techniques like line and branch coverage [12] are distorted by the injected code. Furthermore, monitors or other code probes can affect program behavior or can even mask software defects by changing memory consumption or register usage. But for industrial software development units it is crucial to perform test and verification approaches as realistically as possible.

That's why this article presents an instrumentation-free methodology build upon an external observer which uses DWARF-statements to monitor system behavior. The observer delivers information about variable values used for state-encoding and method calls representing transitions. These information are passed to an engine which reads in the system specification in terms of a UML statechart (we use statecharts since they are the most popular way to specify software component behavior). The statechart is transformed into an executable automaton (EA) which acts as a *golden reference* for equivalence checking. The presented approach allows easy combination with all kinds of coverage methods required by many safety standards like IEC 61511 for process industry systems [1].

3 Method

Section 2 shows, that lots of different approaches exist to check software components using RV. But always automatic or even manual source-code instrumentation must be carried out which can affect program behavior. The following section presents our instrumentation-free observer approach which focuses on equivalence checking of a UML statechart and the corresponding implementation without instrumentation. To keep up significance of the test results the presented methodology avoids modification of the SUT.

[1]Abbr.: Debugging With Attributed Record Formats

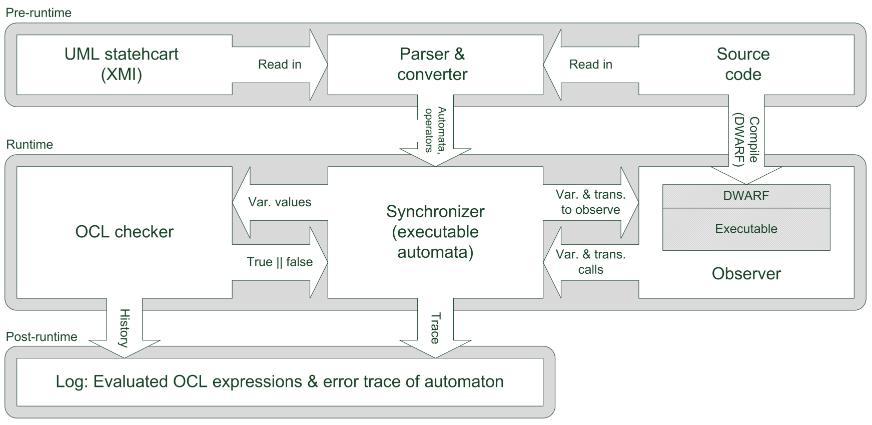


Abbildung 1: Overview of presented approach

We monitor execution behavior from outside by using DWARF-statements (DS) [4] created by the compiler. DWARF is a wide-spread debugging format supported by most compilers and could be generally injected into every program (details are presented in Section 3.3). Figure 1 shows an overview of our approach including all used components and the corresponding directions of communication. Our approach is able to cover all stages of software development starting with unit tests up to integration and system tests. To track behavior of software components our methodology relies on state encoding variables, e.g. in the form of an enumeration (any other variable of basic or complex datatype would also be suitable).

3.1 Pre-Runtime

Before simulation starts, the UML statechart is read by a *parser*. The gathered information about states, transitions and guards are then passed on to the synchronizer (SYNCH) which stores them and then transforms them into an EA (see Section 3.4). The parser also reads in the source-code of the SUT to identify overloaded operators corresponding to user-defined types. Those information are also sent to SYNCH. This is necessary because user-defined types can be used in OCL as operands of comparative operators (see Section 3.5).

3.2 Runtime

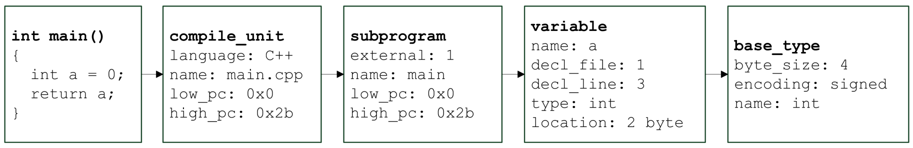
3.3 Observer.

The observer's job is to monitor behavior of the SUT. Therefore, it exploits features of source-level debuggers (for our feasibility study we have used the GNU Debugger 7.1 (GDB)) to read DS stored in an executable. GDB can easily be wrapped by the observer using its machine interface for communication (Graphical debugging front-ends encapsulate GDB in the same way).

DS are created during compile-time (debug mode must be turned on) and enable debuggers to access variable values, monitor function or method calls and read their return values during debugging process. Thereby, *Debugging Information Entries* (DIE) are injected into the SUT. DIE are not affecting program behavior because they are stored in the *Executable and Linking Format* (ELF) and therefore doesn't change the assembler code. For example a DIE specifies the location of a variable in a hierarchical way. At first the compilation unit (the source-code file) is addressed by a DIE. Afterwards a child DIE locates the subprogram (function or method) in which the variable is valid (inluding program counters which points at the loaction where the subprogram is stored) and points to a DIE which provides the information about memory address or register number where the value is stored. Finally a further child DIE defines the type of the variable. Figure 2 presents an example of the integer variable *a* located in a *main*-function. By accessing the DWARF DIE every necessary information is provided during runtime to monitor calls of methods and to access values stored in class attributes and variables. Depending on the current state, the observer receives items to monitor from the synchronizer including all state-encoding variables, transition triggers and all items needed for an OCL evaluation. In reverse, the observer returns the corresponding values to the synchronizer for further processing.

3.4 Synchronizer.

SYNCH orchestrates and synchronizes the whole verification process during simulation-time. It passes a list of all items to be monitored to the observer which returns the corresponding values. For example, the observer communicates a method call of a trigger function for a state change (a transition is fired), SYNCH triggers the encapsulated EA (which is based on Boost's statechart library [3]) and is able to perform the behavioral equivalence check. When a deviation is detected, SYNCH writes a full trace into the log file (see Section 3.1), raises a notification and stops simulation. When an OCL constraint guards a transition, SYNCH requests all corresponding values from the observer and passes them in combination with the



```
int main()
{
  int a = 0;
  return a;
}
```

```
compile_unit
language: C++
name: main.cpp
low_pc: 0x0
high_pc: 0x2b
```

```
subprogram
external: 1
name: main
low_pc: 0x0
high_pc: 0x2b
```

```
variable
name: a
decl_file: 1
decl_line: 3
type: int
location: 2 byte
```

```
base_type
byte_size: 4
encoding: signed
name: int
```

Abbildung 2: Example program and corresponding DWARF data

OCL expression to the OCL solver (OCLS). OCLS returns a logical value which indicates whether the transition was allowed to be fired or not. Finally, after a state change, SYNCH requests the value(s) of the state-encoding variable(s) from the observer and compares them with values of its EA. If the received values are consistent with the ones of the EA, it is ensured, that a valid target state is reached.

3.5 OCL Solver.

OCLS is a stand-alone module which is able to solve OCL expressions [11]. SYNCH passes the expression and corresponding values (obtained by the observer) to the solver. OCLS evaluates the expression and returns a logical value. An OCL expression consists of contexts, pre- and post-conditions. Right now, our approach only regards UML guards and change events. Guards are expressions which must be evaluated to *true* in order to let a transition be fired. To evaluate such an expression the synchronizer must pass all attribute values to the OCL Solver. The synchronizer has all necessary information received by the parser: SYNCH knows after which event he has to request the attribute values to be evaluated in the OCL statement.

3.6 Post-Runtime

Revealed behavioral deviations between implementation and the statechart are stored in an error log file. After runtime, this log holds a complete trace record of fired transitions, evaluated OCL statements and reached states. The log can be used for measuring state and transition coverage as well as for error tracking.

4 Conclusion and Future Work

In this article we have presented an approach to perform runtime verification without source-code instrumentation to keep up test significance during integration and system tests. An external observer monitors program execution and delivers necessary information about state-encoding variables and called methods (fired transitions). The gathered information is processed further by an equivalence checker (SYNCH) which holds an executable model (built up from a UML statechart) as a *golden reference*. Included OCL expressions are evaluated by a stand-alone OCL solver. The next steps are to measure the emerging runtime overhead occurred by our approach and to verify the behavior of the subscription service of the OPC Unified Architecture [8] implemented in terms of a C++ library.

Literatur

[1] Josef Börcsök. *Functional Safety.* Hüthig, 2007.

[2] Feng Chen and Grigore Rosu. Mop: an efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588, 2007.

[3] Andreas Huber Dönni. The boost statechart library. `http://www.boost.org/doc/libs/1_34_1/libs/statechart/doc/index.html`, 05 2011.

[4] Michael J. Eager. Introduction to the DWARF debugging format. Technical report, Eager Consulting, 2007.

[5] Carlos Eduardo Pereira and Luigi Carro. *Distributed real-time embedded systems: Recent advances, future trends and their impact on manufacturing plant control*, volume 31. Elsevier Science Ltd, 2006.

[6] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, October 2009.

[7] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[8] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture.* Springer Publishing Company, Incorporated, 2009.

[9] Scott McMaster and Atif Memon. Call-stack coverage for GUI test suite reduction. *IEEE Trans. Softw. Eng.*, 34:99–115, January 2008.

[10] Gergely Pintér and István Majzik. Runtime verification of statechart implementations. In *WADS*, pages 148–172, 2004.

[11] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[12] Qian Yang, J. Jenny Li, and David Weiss. A survey of coverage based testing tools. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 99–103, New York, NY, USA, 2006. ACM.