Affordable Safe & Secure Mobility Evolution

Advanced Concurrent Static Analysis Toolkit Description

Deliverable D5.2

Deliverable Information			
Nature	Document	**Dissemination Level**	Public
Project	ASSUME	**Project Number**	14014
Deliverable ID	D5.2	**Date**	31.08.2018
Status	Final	**Version**	v13
Contact Person	Reinhold Heckmann	**Organisation**	AbsInt GmbH
Phone	++49-681-38360-25	**E-Mail**	heckmann@absint.com

Author Table

Name	Partner	Email
Reinhold Heckmann	AbsInt	heckmann@absint.com
Arturo Tejada Ruiz	TNO	arturo.tejadaruiz@tno.nl
Mladen Skelin	TU/e	m.skelin@tue.nl
Anja Stoll	MES	anja.stoll@model-engineers.com
Sebastian Ottlik	FZI	ottlik@fzi.de
Philipp Sieweck	UKiel	psi@informatik.uni-kiel.de
Philipp Reinkemeier	OFFIS	philipp.reinkemeier@offis.de
Ralf Vogler	TUM	ralf.vogler@tum.de
Antoine Miné	Sorbonne University	Antoine.Mine@ens.fr
Helmut Seidl	TUM	seidl@in.tum.de
Hassan Salehe Matar	Koç University	hmatar@ku.edu.tr
Anton Paule	FZI	paule@fzi.de
Jens-Peter Rohrlack	MES	jens-peter.rohrlack@model-engineers.com
Rifat Atar	Ericsson	rifat.atar@ericsson.com

Change and Revision History

Version	Date	Reason for Change	Affected sections
v01	23.05.2018	Initial version	All
v02	15.06.2018	Update of SDF section	2.3
v03	22.06.2018	Update of PLAATO section	2.2
v04	25.06.2018	Update of EmbedSanitizer section	5.1
v05	25.06.2018	New section on DRDCheck Hybrid	5.4
v06	25.06.2018	Update of C-SAPP section	4
v07	27.06.2018	Update of Goblint section	3.2
v08	28.06.2018	Update of Gropius section	3.1
v09	11.07.2018	Update of RTANA2 section	2.4
v10	16.07.2018	Update of Astrée section	3.3
v11	20.07.2018	Update of M-XRAY section	2.1
v12	23.07.2018	Finalization for internal review	All
v13	31.08.2018	Finalization for delivery	All

Table of Contents

AUTHOR TABLE **2**

CHANGE AND REVISION HISTORY **3**

TABLE OF CONTENTS **4**

LIST OF FIGURES **7**

LIST OF ABBREVIATIONS **9**

1. EXECUTIVE SUMMARY **11**

2. ANALYSES ON MODEL LEVEL **12**

2.1. M-XRAY: SIMULINK MODEL REFACTORING SUPPORT 13

2.1.1. Introduction *13*
2.1.2. Methodology *13*
2.1.3. Work done in the ASSUME project *13*
2.1.3.1. Cohesion for Simulink models 13
2.1.3.2. Metrics for coupling 14
2.1.3.3. Effective interface metric 15

2.2. PLAATO: PLATFORM ARCHITECTURE ANALYSIS TOOLS 16

2.2.1. Introduction *16*
2.2.2. Methodology *17*
2.2.2.1. Inputs 17
2.2.2.2. Processing 18
2.2.2.3. Output 19
2.2.3. Current status and future development *19*
2.2.4. Use Cases and KPIs *20*

2.3. SDF FOR FREE (SDF^3): PERFORMANCE ANALYSIS OF DATAFLOW MODELS 21

2.3.1. Introduction *21*
2.3.2. Related work *22*
2.3.3. Preliminaries *23*
2.3.3.1. (max,+) Algebra 23
2.3.3.2. Synchronous dataflow 23
2.3.3.3. (max,+) Semantics of Synchronous Dataflow 24
2.3.3.4. Hierarchy in SDF Graphs 26
2.3.3.5. FSM-based Scenario-Aware Dataflow (FSM-SADF) 27
2.3.4. Throughput Analysis of Hierarchical SDF Models *27*
2.3.4.1. Our algorithm 28
2.3.4.2. Symbolic Simulation 28
2.3.4.3. Example 30
2.3.5. Compositionality in Synchronous Dataflow: Modular Performance Analysis from Hierarchical SDF Graphs *33*
2.3.6. Analysis flow *36*
2.3.7. Conclusion *37*
2.3.8. Current status *37*
2.3.9. Use case *37*

2.4. RTANA2: SYSTEM-LEVEL TIMING ANALYSIS OF REAL-TIME SYSTEM MODELS 38

2.4.1. Introduction *38*

2.4.2. Input ... 38
2.4.3. Method description ... 39
2.4.4. Interface to concurrency defect analyses ... 41
2.4.5. Current status and future development ... 42
2.4.5.1. Verification based on Difference Bound Matrices ... 42
2.4.5.2. Verification based on Future Difference Bound Matrices: ... 43
2.4.5.3. Compositional Analysis Strategy Combining State-based and Analytical Methods ... 46

3. STATIC ANALYSIS OF C CODE FOR CONCURRENCY ERRORS ... 50

3.1. GROPIUS ... 50
3.1.1. Input ... 51
3.1.2. Output ... 51
3.1.3. Method description ... 51
3.1.4. Ranking heuristic ... 53
3.1.5. What changed between D5.0 and D5.1 ... 54
3.1.6. What changed since D5.1 ... 54
3.1.7. Use-Cases ... 54
3.1.8. KPI Status ... 54

3.2. GOBLINT ... 55
3.2.1. Input and Output ... 55
3.2.2. Configuration ... 56
3.2.3. Method description ... 57
3.2.4. Analysis of Asynchronous Programs ... 57
3.2.5. Scalability ... 57
3.2.6. Improvements since D5.0 and KPI Status ... 58

3.3. ASTRÉE ... 58
3.3.1. Input and Output ... 58
3.3.2. Features of Asynchronous Programs ... 59
3.3.2.1. Processes ... 59
3.3.2.2. Synchronization Objects ... 60
3.3.3. Analysis of Asynchronous Programs ... 61
3.3.4. Execution Model ... 62
3.3.5. Abstraction of Concrete Behaviour ... 64
3.3.6. Error Reporting ... 64
3.3.7. Improvements during the Project and KPI Status ... 65

4. ANALYSIS OF HARDWARE-DEPENDENT SOFTWARE ... 67

4.1. ANALYSIS OF ASYNCHRONOUS AND CONCURRENT HW DEPENDENT SW ... 67
4.1.1. Input ... 68
4.1.2. Invocation ... 71
4.1.3. Method description ... 71
4.1.4. Output ... 71
4.1.5. Changes compared to D5.0 and future development ... 71
4.1.6. KPI status ... 72

4.2. DEADLOCK ANALYSES FOR REAL-TIME EMBEDDED SYSTEMS ... 72
4.2.1. Method description ... 73
4.2.2. Algorithm ... 74

5. RACE DETECTION BY INSTRUMENTATION 77
5.1. EMBEDSANITIZER: RACE DETECTION FOR ARMV7 POSIX-THREAD APPLICATIONS 77
5.1.1. Introduction 77
5.1.2. Method 77
5.1.2.1. Architecture and Workflow 78
5.1.2.2. Installation 79
5.1.3. Evaluation 79
5.1.3.1. Tool Precision Evaluation 80
5.1.3.2. Tool Performance Evaluation 80
5.1.4. Uses cases and requirements 81
5.1.5. Status w.r.t. KPIs 81
5.2. GOLDILOCKS 82
5.3. FASTTRACK 82
5.4. DRDCHECK HYBRID 83
5.4.1. Introduction 83
5.4.2. Method 83
5.4.3. Evaluation 83
6. CONCLUSIONS AND DISCUSSION 84
REFERENCES 85

List of Figures

Figure 1: M-XRAY workflow 13

Figure 2: Example of subsystem with two inherent components highlighted with coloured rectangles 14

Figure 3: Refactoring example using incoherence metric on subsystem where dependencies are hidden by Goto+From blocks 14

Figure 4: Excerpt from M-XRAY Excel report with quality metrics for number of in- and outports 15

Figure 5: Example of effective interface computation 15

Figure 6: PLAATO Interface example 16

Figure 7: PLAATO’s current structure 17

Figure 8: Example of PLAATO's Matlab GUI 18

Figure 9: Overview of TNO's contribution to the ASSUME project. See Deliverable 1.4 for details 20

Figure 10: Example of an SDF graph. 24

Figure 11: Example of a hierarchical SDF graph. 26

Figure 12: An FSM-SADF graph 27

Figure 13: Compositionality in SDF 33

Figure 14: DSSF profiles 34

Figure 15: FSM-SADF performance model for DSSF 35

Figure 16: SDF^3 analysis flow 36

Figure 17: Task network example 39

Figure 18: State representation for event model (left) and resources (right) 40

Figure 19: System state representation 41

Figure 20: Flow Diagram of compositional combined real-time analysis 47

Figure 21: Case-Study: Exterior Light Management System 47

Figure 22: Analysis steps in Gropius 52

Figure 23: Structure of Goblint (see [13]) 55

Figure 24: Goblint's HTML output 56

Figure 25: Analyses of Hardware-dependent Software (HdS) workflow 68

Figure 26: Specification of the HW/SW interaction 69

Figure 27: Excerpt from the SW/HW Interface Metamodel 70

Figure 28: Platform model for the FZI_UC01 demonstrator 70

Figure 29: Two scenarios with interleavings between two tasks 75

Figure 30: Calculated timing intervals for the two scenarios 76

FIGURE 31: HIGH LEVEL ABSTRACTION OF THREADSANITIZER AND EMBEDSANITIZER IN LLVM/CLANG. (A) THREADSANITIZER: ESSENTIAL LLVM MODULES FOR RACE DETECTION. (B) EMBEDSANITIZER: SAME MODULES MODIFIED TO INSTRUMENT AND DETECT RACES FOR 32-BIT ARM 78

FIGURE 32: SHOWING THE AUTOMATED PROCESS FOR THE INITIAL BUILD OF EMBEDSANITIZER 79

FIGURE 33: SLOWDOWN COMPARISON OF RACE DETECTION ON ARMV7 VS ON QEMU-ARM................ 81

FIGURE 34: ALTERNATIVE OVERVIEW OF EMBEDSANITIZER WHEN USED TO DETECT DATA RACES IN COLLABORATOR'S 32-BIT ARMV7 SMART TV SOFTWARE FOR USE CASE ARC_UC02........................ 81

List of Abbreviations

ADAS	Advanced Driver Assistance Systems
API	Application Programming Interface
AST	Abstract Syntax Tree
BCET	Best-Case Execution-Time
BSF	Breadth-First Search
CFG	Control Flow Graph
CIL	C Intermediate Language
CIVL	Concurrency Intermediate Verification Language
CPU	Central Processing Unit
CSDF	Cycle-Static DataFlow
DMA	Direct Memory Access
DSSF	Deterministic SDF with Shared FIFOs
EA	Enterprise Architect
ECU	Electronic Control Unit
ENS	École Normale Supérieure (Paris)
FCFS	First-Come, First-Served
FIFO	First In, First Out (queue)
FSM	Finite-State Machine
FSM-SADF	Finite-State Machine-based Scenario-Aware DataFlow
GUI	Graphical User Interface
HdS	Hardware-dependent Software
HSDF	Homogeneous SDF
HTML	HyperText Markup Language
HW	Hardware
HWA	Hardware Accelerator
IDE	Integrated Development Environment
I/O	Input/Output
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LCG	Linear Constraint Graph
LLBMC	Low-Level Bounded Model Checker
LLVM	(collection of modular and reusable compiler and toolchain technologies)
LTL	Linear Temporal Logic
MMIO	Memory Mapped Input/Output
MoC	Model of Computation
Mutex	Mutual exclusive lock
M-XRAY	(static analysis tool for Simulink models)
OIL	OSEK Implementation Language

OS	Operating System
PLAATO	Platform Architecture Analysis Tool
RTANA2	(tool for system-level timing analysis of real-time system models)
SDF	Synchronous DataFlow
SDF^3	(tool for performance analysis of dataflow models)
SME	Small or Medium Enterprise
SW	Software
TUM	Technische Universität München
TU/e	Technische Universiteit Eindhoven
UPMC	Université Pierre et Marie Curie
WCET	Worst-Case Execution-Time
WP	Work Package
XML	Extensible Markup Language

1. Executive Summary

This deliverable provides a description of the analysis tools proposed or used in WP5 of the ASSUME project. The common aim of these tools is the analysis of concurrent behaviour. Their descriptions are grouped according to what they analyse and how they do this.

The first group consists of four tools performing analysis on the model level (Section 2). It comprises M-XRAY providing Simulink model refactoring support (Section 2.1), PLAATO performing platform architecture analysis (Section 2.2), SDF^3 for performance analysis of dataflow models (Section 2.3), and RTANA2 for system-level timing analysis of real-time system models (Section 2.4).

The second group consists of three tools for static analysis of C code for concurrency error (Section 3). It comprises Gropius (Section 3.1), Goblint (Section 3.2), and Astrée (Section 3.3). Static analysis means that the code is analysed without executing it. Its results are valid for all executions with all inputs.

The third group deals with analysis of hardware-dependent software (Section 4). It contains the C-SAPP tool for analysis of asynchronous and concurrent hardware-dependent software (Section 4.1) and a related method for deadlock analysis of real-time embedded systems (Section 4.2).

The last group (Section 5) comprises several related approaches for race detection by instrumentation. All these approaches produce a binary executable augmented by instrumentation code that detects concurrency errors while the instrumented program is running and produces a report about its findings.

This deliverable D5.2 is the third in a series of deliverables. Its predecessors D5.0 and D5.1 presented the status of the WP5 tools after the first and second year of the project, respectively. D5.2 presents the status at the end of the project, i.e. after three years of work in ASSUME.

2. Analyses on Model Level

This section presents four tools with the common property that they perform analysis on model level.

The M-XRAY tool by MES is a static analysis tool for Simulink models, which is extended in ASSUME to support the refactoring of models towards multi-core concurrency by computing and representing dependencies (Section 2.1).

The Platform Architecture Analysis Tool PLAATO is developed by TNO with the goal to provide researchers and designers with the ability of assessing a given HW/SW architecture using objective criteria such as failure probabilities, mean time to failure, etc. (Section 2.2).

The SDF^3 tool for performance analysis of dataflow models has been developed by TU/e. In ASSUME, the analysis of finite-state machine-based scenario-aware dataflow is extended to improve precision in the case where the selection of the scenario to be activated next depends on time (Section 2.3).

The RTANA2 tool performs system-level timing analysis of real-time system models. It is being extended by OFFIS so that it can compute properties of real-time models such as response times per task, end-to-end latencies of functional chains of tasks, maximum number of pending activations per task, and pre-emption relationships between tasks for each resource (Section 2.4).

2.1. M-XRAY: Simulink Model Refactoring Support

2.1.1. Introduction

During the lifetime of the ASSUME project, MES extended M-XRAY – its static analysis tool for Simulink models – to support the refactoring of models towards multi-core concurrency. First coupling and cohesion were defined for the dedicated domain of Simulink models in order to support industrial applications. Second, we designed an algorithm to compute dependencies between subsystems. Third, an appropriate representation of the dependencies was introduced to support the planning of refactoring and the assessment of design options.

2.1.2. Methodology

M-XRAY analyzes Simulink models at the level of individual subsystems as well as aggregated results for the complete model. M-XRAY generates different metrics to evaluate different quality aspects of the model, like the complexity of individual subsystems.

To evaluate the aspect of coupling and cohesion and visualize dependencies, M-XRAY computes specific metrics highly correlated to this aspect. These metrics are used by M-XRAY for two purposes: Firstly, the metrics can be used to guide the model developer to parts of the model that are violating specified quality standards. Secondly, the metrics are used for appropriate representations that can give further information of how to start or conduct a refactoring in terms of multi-core concurrency.

As shown in Figure 1 below, M-XRAY conducts the structural analysis of the given Simulink models and presents the described results in an HTML report.

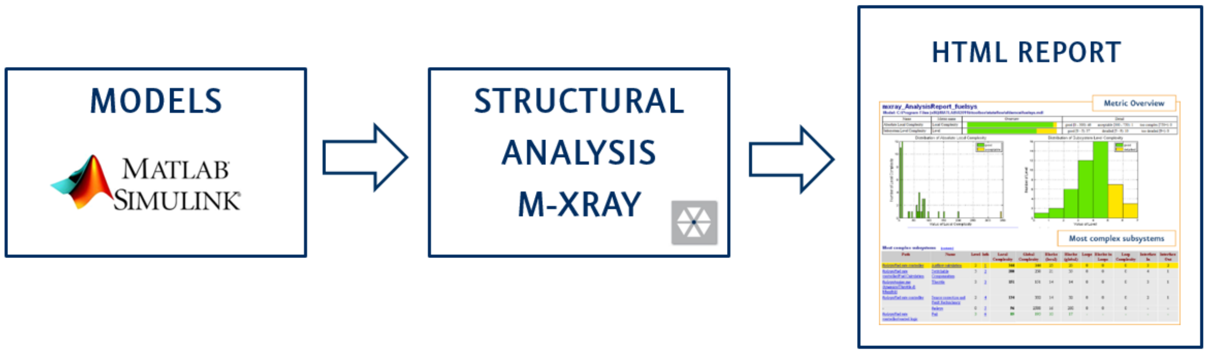


Figure 1: M-XRAY workflow

2.1.3. Work done in the ASSUME project

2.1.3.1. Cohesion for Simulink models

Different metrics have been evaluated by MES that are related to the quality aspect of cohesion. As part of the ASSUME project an *incoherence* metric was introduced in M-XRAY 3.1. This metric is based on the *tight cohesion* introduced in Mäurer et al. [1]. The goal is to identify subsystems that have a low cohesion and, thus, may be split up into multiple subsystems. The new subsystems make the different data flow components obvious, which is a useful information for refactoring, in particular in terms of multi-core concurrency. Imagine the subsystem from Figure 2 which has a low cohesion because, actually, two parallel components are mixed up in a single system (green and orange rectangles).

The *incoherence* metric of M-XRAY helps to detect this kind of subsystems. The incoherence is computed as the inverse of the tight cohesion from Mäurer et al. [1]. Thus, if all blocks of a subsystem are somehow connected by signal paths, the subsystem has a low incoherence (= high cohesion). If there are separate groups of blocks that are not connected by signal paths, the subsystem has a high incoherence (= low cohesion).

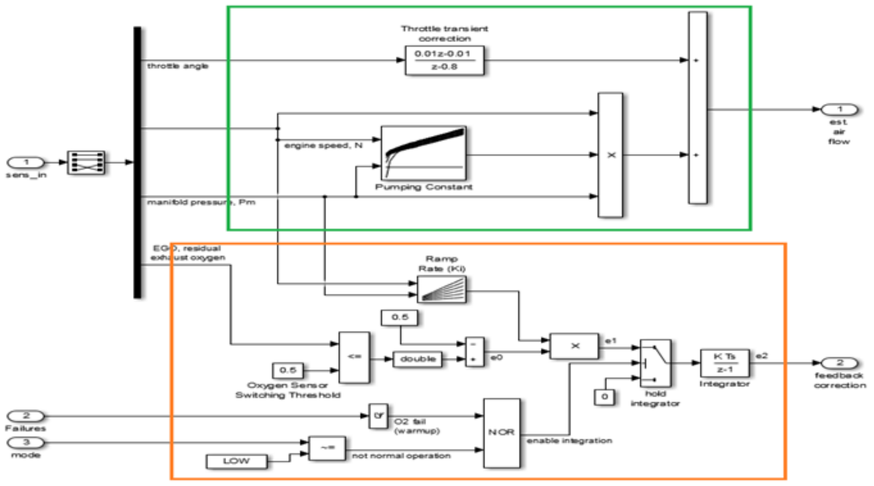


Figure 2: Example of subsystem with two inherent components highlighted with coloured rectangles

The incoherence metric gives a rough estimate of the parallel divisibility of a subsystem. For the example subsystem from Figure 2, the incoherence metric is approximately 2, which gives a good estimate of the parallel divisibility. Figure 3 gives another nice application scenario in terms of identification of parallel components with subsequent refactoring. Due to the extensive use of *Goto+From* blocks, the dependencies between the individual blocks are completely obscured. For this subsystem M-XRAY's incoherence metric is approx. 3, so there should be roughly 3 inherent parallel components. The incoherence metric also helps to identify these components. After refactoring in the model editor, the data flow and separated components are made completely obvious (right part of Figure 3). Thus, the incoherence metric can especially guide the refactoring for cases where the separability of a subsystem is very hard to estimate manually in advance.

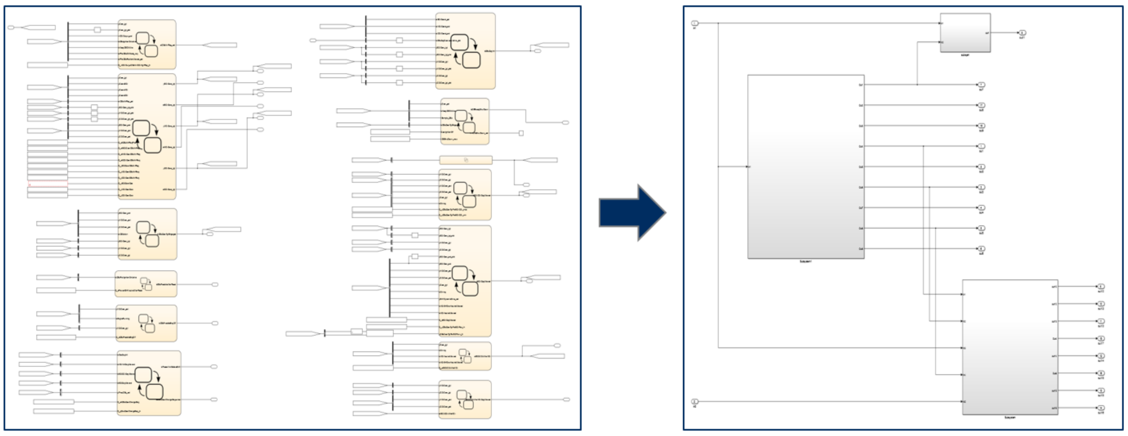
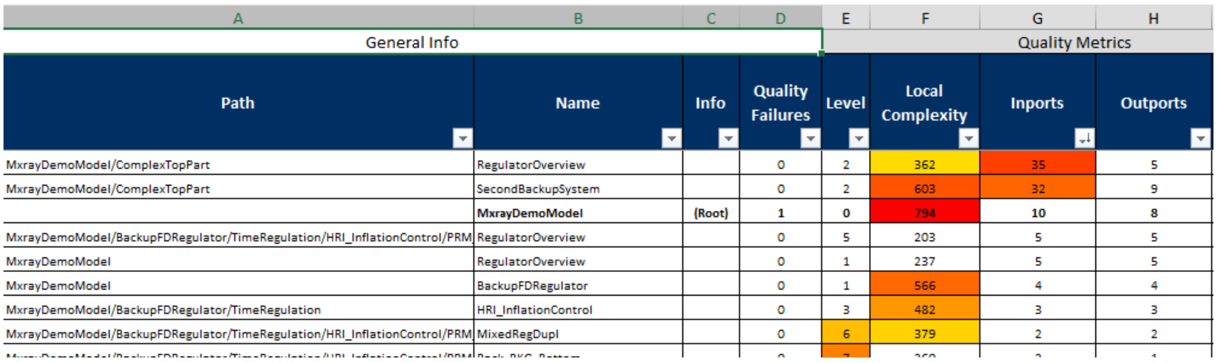


Figure 3: Refactoring example using incoherence metric on subsystem where dependencies are hidden by Goto+From blocks

2.1.3.2. Metrics for coupling

As a first supplement to the incoherence metric introduced in Release 3.1, quality metrics for the most direct and obvious forms of subsystem coupling were added in M-XRAY 3.2: Count of subsystem inports and subsystem outports. These simple but important quality metrics help to assure quality standards like ISO 26262 and their demand for a 'Restricted size of interfaces'. A small interface in terms of inports and outports is a prerequisite for a fast refactoring in terms of multi-core concurrency. Therefore M-XRAY predefines quality metrics for the number of inports and outports. For quality metrics, thresholds can be set and surveyed easily from the M-XRAY reports. Figure 4 shows an excerpt from an M-XRAY Excel report where you can find quality metrics columns for the number of in- and outports of a subsystem together with a color shading for subsystem values above a quality threshold.



A	B	C	D	E	F	G	H
General Info				Quality Metrics			
Path	Name	Info	Quality Failures	Level	Local Complexity	Inports	Outports
MxrayDemoModel/ComplexTopPart	RegulatorOverview		0	2	362	35	5
MxrayDemoModel/ComplexTopPart	SecondBackupSystem		0	2	603	32	9
	MxrayDemoModel	**(Root)**	**1**	**0**	**794**	**10**	**8**
MxrayDemoModel/BackupFDRegulator/TimeRegulation/HRI_InflationControl/PRM	RegulatorOverview		0	5	203	5	5
MxrayDemoModel	RegulatorOverview		0	1	237	5	5
MxrayDemoModel	BackupFDRegulator		0	1	566	4	4
MxrayDemoModel/BackupFDRegulator/TimeRegulation	HRI_InflationControl		0	3	482	3	3
MxrayDemoModel/BackupFDRegulator/TimeRegulation/HRI_InflationControl/PRM	MixedRegDupl		0	6	379	2	2

Figure 4: Excerpt from M-XRAY Excel report with quality metrics for number of in- and outports

2.1.3.3. Effective interface metric

To extend the incoherence and interface count metrics already introduced in M-XRAY 3.1 and 3.2, M-XRAY has been extended as part of the ASSUME project to measure the 'effective interface' of Simulink subsystems. As Simulink subsystems can have input and output signals in the form of buses and as signals can be directly forwarded to subordinate subsystems, a key to understand the real coupling of a subsystem is to detect which input and output signals are effectively used or defined in a subsystem including its subordinate subsystems. This is called the 'effective interface' of the subsystem. To give a simple example, the subsystem from Figure 5 has an effective input interface size of two as only two of six input bus signals are effectively used in the subsystem.

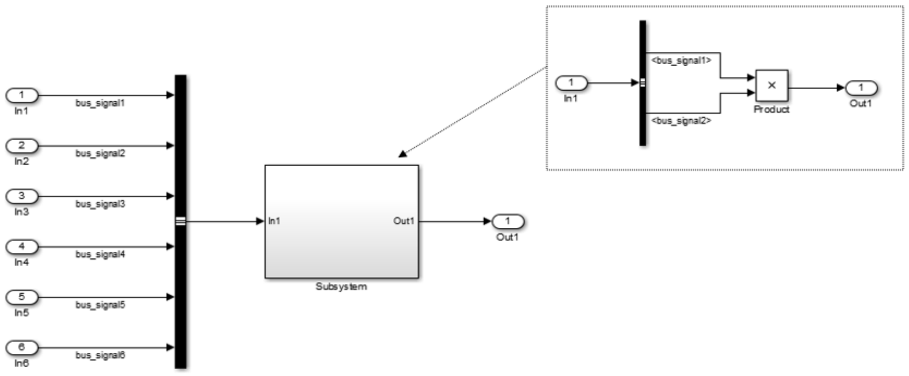


Figure 5: Example of effective interface computation

2.2. PLAATO: Platform Architecture Analysis Tools

2.2.1. Introduction

In safety-critical domains such as automotive, railway, and avionics, even a small failure of a system might cause injury to or death of people. A number of international safety standards are introduced as guidelines for system suppliers to keep the risk of systems at an acceptable level, such as IEC 61508 (multiple domains), ISO 26262 (automotive domain), DO 254 (avionic domain), CENELEC railway standards (railway domain). In the automotive domain, currently the ISO 26262 standard, which is a goal-oriented standard for safety-critical systems within the domain of road vehicles, is the state of the art. This is, of course, the applicable standard for the Dutch (VDL) use case, which is the driver for TNO's developments in the ASSUME project.

After its introduction in 2011, ISO 26262 has attracted more and more attention in the automotive domain. There are more than 120 work products generated throughout the safety lifecycle suggested by this standard. This makes managing traceability and consistency of the information an absolute necessity for assuring safety and compliance. In order to be able to maintain the above-mentioned traceability when designing a vehicular system, it is important to have a well-structured process in place. This is already partly ensured when ISO 26262 is followed, however the norm still leaves a lot open for interpretation and in itself cannot guarantee the quality of a design process.

To aid in this process, TNO has developed a tool called Platform Architecture & Analysis Tool (PLAATO) using Enterprise Architect as front end to input hardware and software models (and eventually link them to requirements using an ISO 26262 template). An example of that interface is shown in Figure 6.

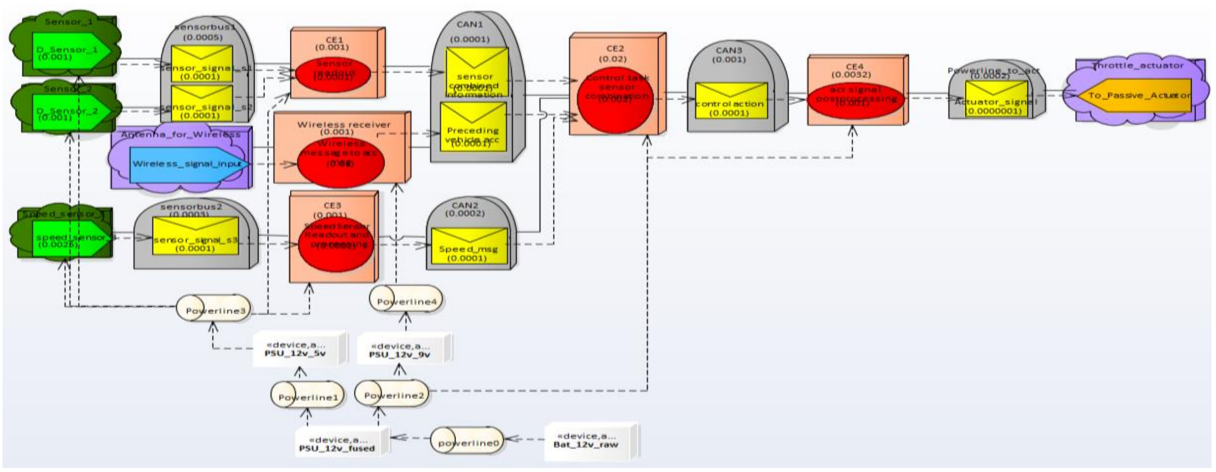


Figure 6: PLAATO Interface example

Based on this information, PLAATO is able to automatically generate fault trees and to perform reliability analysis on them (the analysis is performed in Matlab). The tool can also provide support advising the designer on where to possibly improve the architecture to increase its reliability (i.e. resilience to faults) in accordance with safety requirements.

2.2.2. Methodology

TNO uses an existing commercial tool called Enterprise Architect (EA) to perform system design and analysis in a structured way. To achieve this, a specific “way of working” was set up by customizing the EA user interface and by connecting it with several other software tools (e.g., Excel, Matlab, etc.). Enterprise Architect supports design and architecture of a system at several levels of abstraction. It also supports traceability to documentation, code simulation and centralisation of the design.

By making use of different diagrams that contain specific information for specific people, a natural layered and ordered representation of a system design can be presented to the user. This together with EA’s tools for maintaining traceability ensure that documentation about the design, the design decisions, and other project information is kept very close to the actual system development.

PLAATO’s current structure is shown in Figure 7.

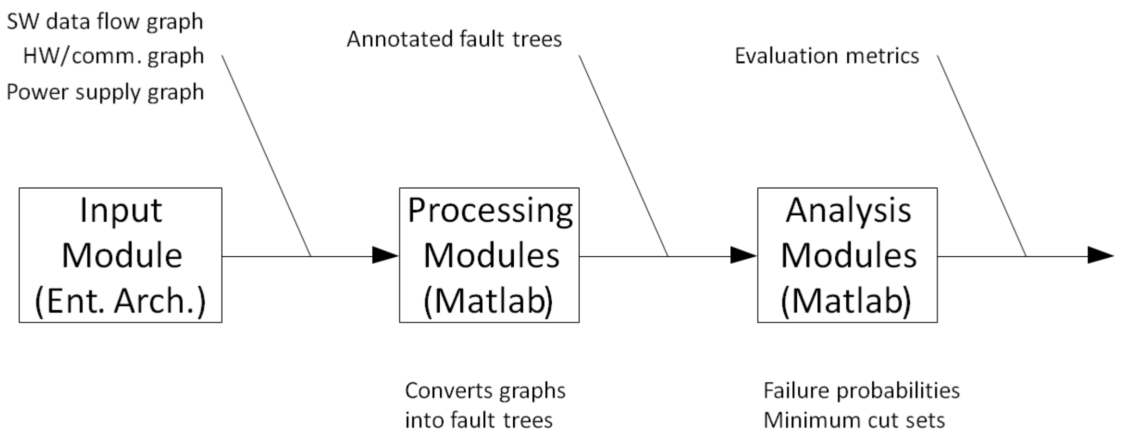


Figure 7: PLAATO’s current structure

The above structure is complete. PLAATO is able to take an ADAS architecture and provide insight on its reliability, the main points of attention (e.g., which component failures are most critical), etc.

2.2.2.1. Inputs

PLAATO enables the user to deliver design input through a graphical representation of the system. To this end, three diagram templates have been added to those that are already available in Enterprise Architect:

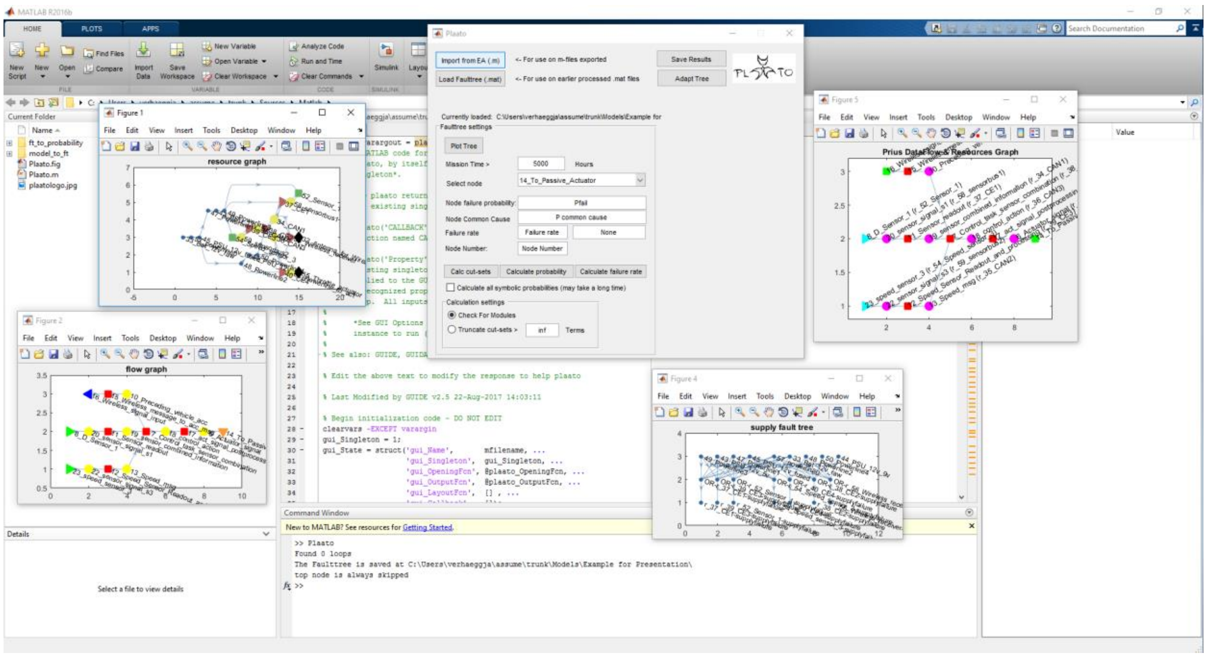
- The first diagram is a function description diagram which shows which functions are present in the system. It captures information about what these functions do and the specific details about the information that is exchanged between these functions.
- The second diagram is a hardware description diagram, which captures details on how hardware is interconnected and what properties the hardware has. It also defines the interfaces that are used to communicate between hardware components, and properties that are required for fault tree generation (e.g. failure probability or failure rate).

- The third diagram is the deployment diagram. It contains information on how functions are deployed or mapped to the hardware. In this way an engineer can experiment with deploying multiple functions on one or more Electronic Control Units (ECUs) or shift tasks between ECUs.

2.2.2.2. Processing

When all information has been input correctly, it can be exported to Matlab for further processing. This is done by a plug-in to EA developed by TNO. The plug-in exports an ".m" file that can be interpreted by Matlab. This file is then executed, yielding a representation in Matlab of the system deployment.

This representation allows the user to perform system analysis using a custom-made Matlab graphical user interface (GUI) (see Figure 8). This GUI can be used to generate a fault tree from the information that was entered in the model diagrams. When information is missing the tool will try to guide the user to enter all required information in EA.



Currently loaded: C:\Users\verhaeggja\assume\trunk\Models\Example for
Faulttree settings

Mission Time >	5000	Hours
Select node	14_To_Passive_Actuator	
Node failure probability:	Pfail	
Node Common Cause	P common cause	
Failure rate	Failure rate	None
Node Number:	Node Number	

☐ Calculate all symbolic probabilities (may take a long time)

Calculation settings
(●) Check For Modules
( ) Truncate cut-sets > inf Terms

```
% Begin initialization code - DO NOT EDIT
clearvars -EXCEPT varargin
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @plaato_OpeningFcn, ...
                   'gui_OutputFcn',  @plaato_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
```

Command Window

New to MATLAB? See resources for Getting Started.

```
>> Plaato
Found 0 loops
The Faulttree is saved at C:\Users\verhaeggja\assume\trunk\Models\Example for Presentation\
top node is always skipped
>>
```

Figure 8: Example of PLAATO's Matlab GUI

Once everything is complete, the fault tree can be analysed structurally as well as quantitatively. The structural analysis can be performed from a very early stage in the project as numbers are not strictly required – only the structure of the system will suffice. The quantitative analysis requires actual numbers such as failure rates or importance metrics of specific elements in the system. The GUI offers the user several options for both forms of analysis, however specific knowledge of the analysis methods is required to be able to interpret the results. A Matlab script subsequently processes these results to generate a static fault tree for a specific architectural component failure. (A fault tree describes the logical chain of component/subsystem faults that may trigger the considered architectural component failure [3]).

2.2.2.3. Output

PLAATO provides support to the system designer on deciding where to possibly improve the system's architecture. To this end, PLAATO offers a number of structural and quantitative analysis tools (running over the fault trees described above).

For structural analysis PLAATO offers the ability to look at:

- Modules in the tree. These are independent regions which will show that a certain part of the tree is independent of other parts of the tree. This can indicate containment regions, in which failures will not propagate to other branches of the tree. These modular regions can be emphasised by the tool.
- Minimal cut-sets. These are the smallest terms of the logical formula (Boolean expression) that corresponds with a tree. The expression describes how the Basic Events contribute to failure of the top node of the tree. Basic events are considered to be the smallest elements that can fail in the system. By looking at the expressions one gets insight in which events may result in system failures. If there are only few (or only one), this means that few faults (or only one) are required to trigger a system failure.
- The actual tree and its structure can be seen as a graphical representation. This will allow the analyst to visually inspect the tree.

For quantitative analysis, PLAATO offers the ability to look at:

- Failure probabilities of the basic events and how these are distributed with respect to each other.
- Failure probabilities of intermediate events and top event. These probabilities have to be calculated by finding the probability expression from the logical formula. The tool can perform these calculations.
- Importance metrics. The tool offers the ability to calculate importance metrics that will give information about the relative importance of the basic events in the tree. The tool supports the Birnbaum [55] and the Fussel-Vesely [56] importance measures.

The Matlab tool allows the user to store results in ".mat" files to be used for later analysis. Node properties can be adapted with the tool once the tree is generated. Additionally, one can build fault-trees directly in EA. A tree can then be export to an ".m" file and also be analysed in the tool. This way the fault-tree generation step can be skipped.

2.2.3. Current status and future development

Figure 9 shows the main contributions of TNO to the ASSUME project, including PLAATO development (see Deliverable 1.4). At this point, the development of PLAATO has reached the stage of a mature prototype.

As shown in Figure 9, PLAATO mostly covers the left-hand side of the safety engineering V-cycle. The rest of the left side of the V-cycle (from Item Definition to System Design) will be covered by a second tool called MBaSSy: Model Based System Safety Analysis (MBaSSy). This tool is currently under development at TNO as part of a PhD project (partially supported by ASSUME). Once completed, MBaSSy will replace the PLAATO interface shown in Figure 6 with a SysML-compliant one, while keeping PLAATO's fault-tree analysis engine. It is expected that MBaSSy will cover the full V-cycle once completed, thus merging the three main contributions of TNO to this project (see Figure 9 and Deliverable 1.4 for details). For more information on MBaSSy, see Deliverable 3.4.

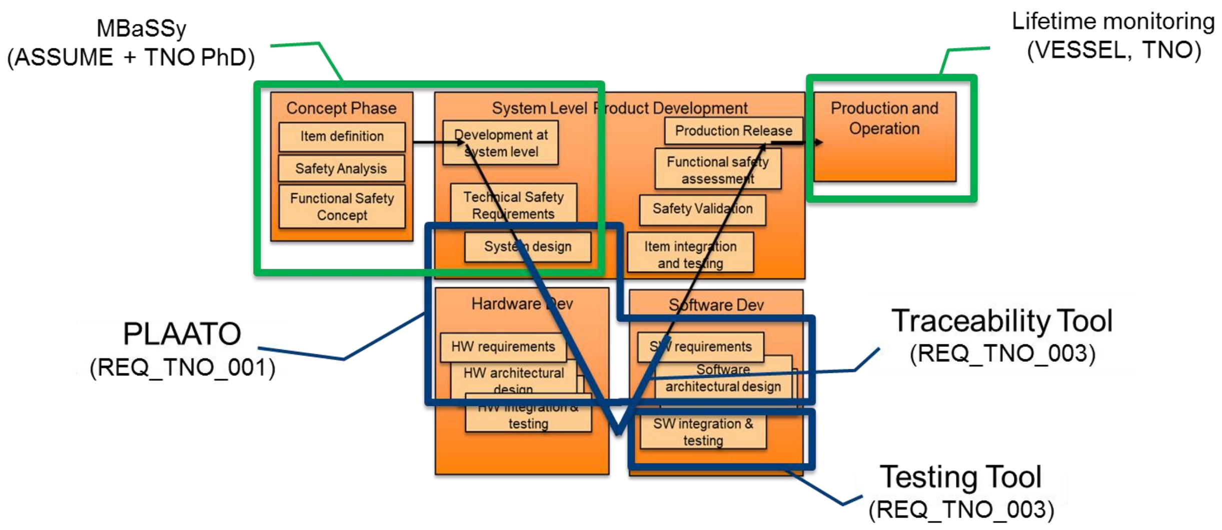


Figure 9: Overview of TNO's contribution to the ASSUME project. See Deliverable 1.4 for details

2.2.4. Use Cases and KPIs

All the activities described above are part of VDL's use case (TNO_UC_01) and support KPI1.1 (reduce the effort required to set up and employ an analysis tool).

2.3. SDF for free (SDF^3): Performance Analysis of Dataflow Models

2.3.1. Introduction

Dataflow models of computation are widely used to represent streaming systems. This is thanks to their simple graphical representation, compactness and the ability to express parallelism inherent to many streaming systems. In dataflow, a system is represented by a directed graph where nodes are called *actors* and edges are called *channels*. Actors represent computational kernels while channels typically capture data, control and resource dependencies between actors. The quanta of information exchanged across channels are called *tokens*. Actors involve themselves into communication with other actors by *firing*. The firing represents the quantum of computation during which actors consume tokens from their input channels and produce tokens in their output channels. Preconditions for firing are given by firing rules [32]. The numbers of tokens produced and consumed are called *rates*. In the timed versions of dataflow that we are investigating in this project, actor firings have duration that we call the actor *firing delay*.

There exist quite a number of dataflow models. They can be roughly divided into decidable [30] and dynamic dataflow models [18]. Decidable dataflow models can be considered versions of dataflow with restricted semantics so that the model can be scheduled at design-time as well as analysed for boundedness, deadlock and timing properties. Examples of decidable dataflow formalisms are synchronous dataflow (SDF) [5], cyclo-static dataflow [20] and scalable SDF [35]. Dynamic dataflow models offer more expressive power in exchange for a decrease in analysability and implementation efficiency [37]. Well-known examples are Boolean dataflow and dynamic dataflow [21].

All in all, in terms of support for design and analysis of timing-predictable and repeatable systems (and most predictable systems are at first *real-time systems*), among dataflow models, decidable dataflow models still play a more pronounced role than the echelons of emerging dynamic dataflow models. This in particular refers to SDF as the most stable and mature flavour of decidable dataflow that is characterized by its predictability, strong formal properties and amenability to powerful optimization techniques [18]. In SDF rates are fixed and known at compilation time. The firing rules of SDF are conjunctive [32] in the sense that for an actor to fire, every of its inbound channels must contain the number of tokens prescribed by the port rate defined by the actor and the inbound channel in consideration. Furthermore, they are distributive [32] in the sense that when the actor fires all outbound channels receive tokens in the quantity prescribed by the corresponding port rates. As we will further elaborate below, SDF graphs evolve in iterations. An iteration is a set of actor firings that have no net effect on the token distribution of the graph. The number of firings of a particular actor in an iteration is given in the so-called repetition vector of the graph. In this report, we consider the so-called self-timed execution of SDF graphs, which means that actors must fire as soon as they are enabled.

Several examples of use of SDF in design and analysis of predictable and repeatable systems can be found in [16][33][38]. If we study these papers, we see the SDF formalism is not only useful for reasoning about the functional behaviour and correctness of systems, but also, in its timed version [36], can be used when one needs to derive or prove worst-case performance guarantees, in particular throughput that is a vital performance indicator in real-time streaming systems, which is defined as the long-term average number of completed iterations per time-unit.

Many authors [9][26][28][29][36] have dealt with the problem of performance analysis of SDF models. To make these techniques applicable in everyday engineering practice, it is important that they are available in tools that can be utilized in fully or semi-automated design flows. The SDF^3 tool [4] developed by TU/e is such a tool. In particular, it implements various performance analysis algorithms for dataflow MoCs such as synchronous dataflow (SDF) [5], cycle-static dataflow (CSDF) [6], and finite-state machine-based scenario-aware dataflow (FSM-SADF) [7].

The common characteristic of all of the algorithms is that they are in terms of performance adversely affected by the increase of repetition vector entries of the graph. In particular, the performance will scale at least linearly with the sum of the repetition vector entries [7].

However, monolithic SDF models are inconvenient for capturing large designs. Therefore, allowing for compositional modelling is a necessity in the design of large systems as it enforces good engineering practices such as modularity and design reuse, and improves readability. Hierarchy has been introduced to SDF [39][17][34][24]. There is a technique for directly analyzing hierarchical SDF structures [23] but it is not exact, which means that it can only give a conservative throughput estimate but not the exact value. The *exact* throughput analysis algorithms existing so far can only be applied to hierarchical dataflow models after flattening them.

In this report, we propose a new exact modular technique for throughput analysis of a subclass of hierarchical SDF graphs with an arbitrary number of hierarchy levels that removes the need for flattening the graph. This is achieved by using (max,+)-based state-space representations of hierarchical actors instead of flattening in the context of existing throughput analysis techniques based on symbolic simulation. Furthermore, as our technique is able to take advantage of the hierarchical semantics of SDF, we argue that our technique helps to mitigate the adverse effect of an increase in the repetition vector entries on the performance of existing performance analysis techniques. This is due to the fact that no matter how many times a hierarchical actor is scheduled in the composition, we do not need to replicate the firings of all the actors embodied in the hierarchical actor as the existing techniques do, but only use its more compact state-space representation to capture the effects of its firing on the rest of the composition.

In addition (novelty compared to D5.1), we use our newly developed modular technique for throughput analysis of hierarchical SDF graphs to address the compositionality problem in SDF in the context of modular performance analysis and Deterministic SDF with Shared FIFOs.

In particular, hierarchical SDF models are not compositional: a composite SDF actor cannot be represented as an atomic SDF actor without loss of information that can lead to rate inconsistency or deadlock [57]. To remedy the former while working in the context of code generation, the authors of [57] propose a compositional abstraction of composite SDF actors called DSSF (Deterministic SDF with Shared FIFOs) that can be used for modular compilation. Nevertheless, the DSSF profiles have no accompanying performance models. In this work, we show how to incrementally build performance models for DSSF profiles using FSM-based scenario-aware dataflow (FSM-SADF) [60] across arbitrary levels of hierarchy that eventually brings us to the overall system performance model from which a performance metric can be derived using the usual (max,+)-based techniques. We illustrate our approach on a simple hierarchical SDF graph borrowed from [57].

The remainder of Section 2.3 is organized as follows. Section 2.3.2 discusses related work, Section 2.3.3 covers the basic concepts used, Section 2.3.4 presents our throughput analysis technique, Section 2.3.5 compositionality, and Section 2.3.6 the analysis flow.

2.3.2. Related work

Roughly, state-of-the-art techniques for throughput analysis of SDF graphs can be divided in two groups.

The first group of approaches is based on the conversion of SDF graphs to equivalent homogeneous SDF (HSDF) graphs. HSDF is a special kind of SDF where all rates equal 1. The basic algorithm for the conversion is described in [36]. The drawback of these approaches is that the size of the graph may expand exponentially [26]. However, advances have been made by the authors of [29] wherein the size of the expansion can be significantly reduced by constructing a so-called linear constraint graph (LCG) from the original SDF graph. With LCG in particular, the compaction is achieved by taking advantage of its redundancy and regularity. Still, some graphs as reported in [29] cannot be represented compactly by the LCG.

The second group of approaches are the simulation-based approaches. The seminal work of [28] performs explicit state-space exploration of the operational semantics of SDF. Despite its high worst-case complexity, the method works well in practice, while the techniques based on the conversion of [36] often fail. The symbolic simulation-based approach described in [26][9] uses (max,+) algebra to capture the self-timed execution of SDF graphs. In particular, the graph's evolution is sublimed into a simple recursive (max,+) linear matrix equation. The matrix in the equation is derived by symbolic simulation of one iteration of the SDF graph. This matrix can be considered as the incidence matrix of a weighted digraph, the maximum cycle mean of which is equal to the inverse of the graph's throughput.

All the exact techniques mentioned above have the common characteristic that the increase of the repetition vector entries in the graph will adversely affect their performance. In addition, the technique of [28] is also sensitive to the length of the graph's transient (self-timed execution of an SDF graph consists of a periodic phase preceded by a so-called transient phase). Furthermore, none of the exact techniques are directly applicable to hierarchical SDF structures, i.e., the hierarchical model should be flattened before.

There exists a technique that targets hierarchical SDF structures [23] but it is not exact, which means that it can only give a conservative throughput estimate but not the exact value.

2.3.3. Preliminaries

This section recaps the (max,+) algebra, the basic SDF concepts and the (max,+) linear system theoretic aspects of SDF that are used in this report.

2.3.3.1. (max,+) Algebra

Let $\mathbb{R}_{max} = \mathbb{R} \cup \{-\infty\}$ where $\mathbb{R}$ is the set of real numbers. For elements $a, b \in \mathbb{R}_{max}$, we define operations $\oplus$ and $\otimes$ with max as addition ($a \oplus b \overset{\text{def}}{=} \max(a, b)$) and + as product ($a \otimes b \overset{\text{def}}{=} a + b$). The set $\mathbb{R}_{max}$ together with operations $\oplus$ and $\otimes$ extended to matrices and vectors in the same way as in conventional linear algebra is called (max,+) algebra. The set of n-dimensional (max,+) vectors is denoted $\mathbb{R}^n_{max}$, while $\mathbb{R}^{n\times n}_{max}$ denotes the set of $n \times n$ (max,+) matrices. The (sup-) sum of matrices $A, B \in \mathbb{R}^{n\times n}_{max}$ is defined by $[A \oplus B]_{i,j} = [A]_{i,j} \oplus [B]_{i,j}$ where $[A]_{i,j}$ and $[B]_{i,j}$ are entries of matrices A and B with indices i and j. The matrix product $A \otimes B$ is defined by $[A \otimes B]_{i,j} = \bigoplus_{k=1}^{n} [A]_{j,k} \otimes [B]_{k,j}$.

2.3.3.2. Synchronous dataflow

Figure 10 shows an SDF graph.

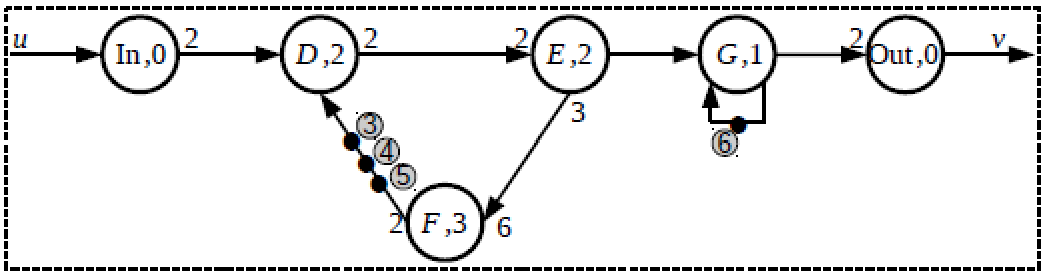


Figure 10: Example of an SDF graph.

The graph has six actors, *In*, *D*, *E*, *F*, *G* and *Out*. Actor firing delays are denoted next to actor names. Rates are denoted next to channel ends with a convention that the omission of a rate value implies the value of 1. Notice that the graph in the figure has two feedback loops, one going from actor *E* to *D* across *F*, and a so-called self-edge from actor *G* back to itself. Such feedback loops can cause the graph to deadlock because actors in the loops depend on each other for tokens. Therefore, feedback loops must include a certain number of *initial tokens* that specify the initial condition from which the execution starts. In the figure, these are depicted using grey circles and are marked as 3, 4, 5, 6.

SDF graphs evolve in iterations. An iteration is a set of actor firings that have no net effect on the token distribution in the graph. The numbers of firings are stored in the repetition vector of the graph Γ. For the graph in Figure 10, $\Gamma(In, D, E, F, G, Out) = [1, 2, 2, 1, 2, 1]^T$. This vector can be obtained by solving the so-called set of balance equations for an SDF graph [5]. Notice that iterations can overlap in time, i.e. they can be pipelined. An SDF graph can be closed or open [39] depending on whether all input ports are connected or not, respectively. The graph in Figure 10 is open as not all its input ports are connected.

2.3.3.3. (max,+) Semantics of Synchronous Dataflow

We use (max,+) algebra [15] to model timed SDF graphs. It is a natural choice as it has two operations that determine the self-timed execution of SDF graphs: synchronization and delay. Synchronization manifests itself when an actor waits for all its input tokens to become available ($\oplus$ i.e. max in (max,+)) before firing. The delay manifests itself in the fact that tokens will be produced only after an amount of time corresponding to the actor firing delay after the firing has begun ($\otimes$ i.e. + in (max,+)). We mentioned that SDF graphs evolve in iterations that restore the graph back to its initial state. The initial state is determined by the distribution of initial tokens over the channels of the graph. Thus, in terms of time, the evolution of an SDF graph can be represented as a sequence of vectors $\boldsymbol{x}(k)$ where each entry of the vector stores the availability time of a token produced in place of a particular initial token after the k-th iteration of the graph. Geilen [9] shows that this sequence (for closed SDF graphs) can be determined by a (max,+) linear recursive equation

$$\mathbf{x}(k+1) = M \otimes \mathbf{x}(k), \tag{1.1.1}$$

where M is the (max,+) matrix of the graph that defines its state-space representation. For open SDF graphs, whose inputs are fed by the token sequence $\boldsymbol{u}(k)$, and that produce tokens the timestamps of which are stored in sequence $\boldsymbol{v}(k)$, (1.1.1) can be generalized to the form

$$\begin{bmatrix} x(k+1) \\ v(k) \end{bmatrix} = \begin{bmatrix} M^A & M^B \\ M^C & M^D \end{bmatrix} \otimes \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \tag{1.1.2}$$

where M^A is the state matrix, M^B is the input matrix, M^C is the output matrix and M^D is the feed-through matrix [27].

These matrices (as in conventional linear system theory) encode mutual dependencies between inputs, outputs and internal state. They can be derived via symbolic simulation of one iteration of the graph as described in [9]. We illustrate how to do this on the example SDF graph in Figure 10. To establish the relationship between the timestamps of tokens contained in (1.1.2) we need to express the timestamps of tokens produced in positions of initial tokens after the (k+1)-st iteration and tokens produced at the outputs of the graph as (max,+) linear combinations of the timestamps of the same tokens after the k-th graph iteration and the input tokens. For the graph of Figure 10,

$t_3 = [0 \quad -\infty \quad -\infty \quad -\infty \quad -\infty] \otimes [x(k) \quad u(k)]^T$, $t_4 = [-\infty \quad 0 \quad -\infty \quad -\infty \quad -\infty] \otimes [x(k) \quad u(k)]^T$

all the way up to $t_u = [-\infty \quad -\infty \quad -\infty \quad -\infty \quad 0] \otimes [x(k) \quad u(k)]^T$. We call these timestamps symbolic timestamps. We now perform symbolic simulation. The iteration is given by the schedule $In\, D^2 E^2 F G^2 Out$ where powers represent actor repetition counts. The iteration starts by actor In firing. This firing consumes the input token u and produces two tokens in channel (In, D) carrying the symbolic timestamp $t_u \otimes 0 = [-\infty \quad -\infty \quad -\infty \quad -\infty \quad 0]$. These tokens along with initial tokens in channel (F, D) fuel two firings of actor D as follows. The firings produce two tokens each. The first two have the symbolic timestamp

$([-\infty \quad -\infty \quad -\infty \quad -\infty \quad 0] \oplus t_3) \otimes 2 = [2 \quad -\infty \quad -\infty \quad -\infty \quad 2]$.

The remaining are of the following symbolic timestamp

$([-\infty \quad -\infty \quad -\infty \quad -\infty \quad 0] \oplus t_4) \otimes 2 = [-\infty \quad 2 \quad -\infty \quad -\infty \quad 2]$.

Then we proceed with actor E the first firing of which is initialized by the tokens produced by the first firing of actor D. The firing results in production of three tokens in channel (E, F) and one token in channel (E, G) with the timestamp $[2 \quad -\infty \quad -\infty \quad -\infty \quad 2] \otimes 2 = [4 \quad -\infty \quad -\infty \quad -\infty \quad 4]$. The tokens produced by the second firing are available at $[-\infty \quad 2 \quad -\infty \quad -\infty \quad 2] \otimes 2 = [-\infty \quad 4 \quad -\infty \quad -\infty \quad 4]$. This enables actor F to fire and restore tokens in position 4 and 5 with the symbolic timestamps $t_4' = t_5' = [7 \quad 7 \quad -\infty \quad -\infty \quad 7]$.

Note that token 5 was not consumed in the current iteration but was shifted in position of token 3. Thus, $t_3' = t_5 = [-\infty \quad -\infty \quad 0 \quad -\infty \quad -\infty]$. Similarly, actor G fires and its second firing results in restoring the token in position 6 that ends up with the timestamp $t_6' = [6 \quad 5 \quad -\infty \quad 2 \quad 6]$. This is also the timestamp of the token produced on the output, i.e. $t_v = t_6'$. If we gather the symbolic timestamps t_3', t_4', t_5', t_6' and t_v' row-by-row into a matrix, we obtain

$$\begin{bmatrix} x(k+1) \\ v(k) \end{bmatrix} = \left[\begin{array}{cccc|c} -\infty & -\infty & 0 & -\infty & -\infty \\ 7 & 7 & -\infty & -\infty & 7 \\ 7 & 7 & -\infty & -\infty & 7 \\ 6 & 5 & -\infty & 2 & 6 \\ \hline 6 & 5 & -\infty & 2 & 6 \end{array}\right] \otimes \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \qquad (1.1.3)$$

where $x(k+1) = \begin{bmatrix} t_3' & t_4' & t_5' & t_6' \end{bmatrix}^T$, $v(k) = [t_v]^T$, $x(k) = [t_3 \quad t_4 \quad t_5 \quad t_6]^T$ and $u(k) = [t_u]^T$.

2.3.3.4. Hierarchy in SDF Graphs

In this report, following the terminology of [39], when we talk about hierarchical SDF graphs, we mean graphs that contain hierarchical actors. Unlike atomic actors, a hierarchical actor encapsulates an SDF graph. Hierarchical actors can then be connected to other SDF actors, either atomic or hierarchical to form hierarchies of arbitrary depths.

An example of a hierarchical SDF graph is shown in Figure 11.

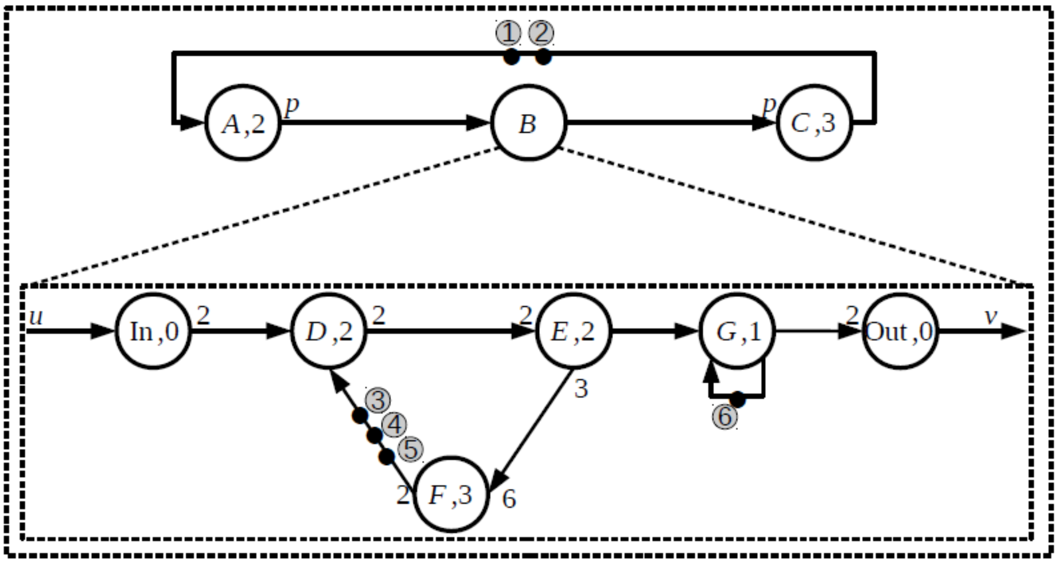


Figure 11: Example of a hierarchical SDF graph.

In the figure, the top-level graph is composed out of three actors *A*, *B* and *C*. Actors *A* and *C* are atomic, while actor *B* is a hierarchical actor that encapsulates the SDF graph of Figure 10. In this particular example, the output port of actor *A* is connected to the input port of actor *In*, while the input port of actor *C* is connected to the output port of actor *Out* of the encapsulated graph of hierarchical actor *B*.

Although hierarchical SDF models are widely used (e.g. in the well-known Ptolemy II framework [25]), care must be taken as there is one complication. In general, hierarchical SDF models are not compositional. In particular, a hierarchical SDF actor cannot be represented by an atomic SDF actor without loss of information that can lead to inconsistency and deadlock [39]. In this report, we assume that only valid aggregations are specified.

2.3.3.5. FSM-based Scenario-Aware Dataflow (FSM-SADF)

FSM-SADF is a dynamic dataflow model that preserves as much as possible of the determinacy of dataflow behaviour while introducing the possibility of non-deterministic variations in the form of scenarios individually represented as SDF graphs. Operationally, an FSM-SADF evolves in (possibly) partial iterations of its non-deterministically selected scenarios captured by SDF graphs. The scenario FSM defines the scenario occurrence patterns.

Figure 12 shows an example of an FSM-SADF graph.

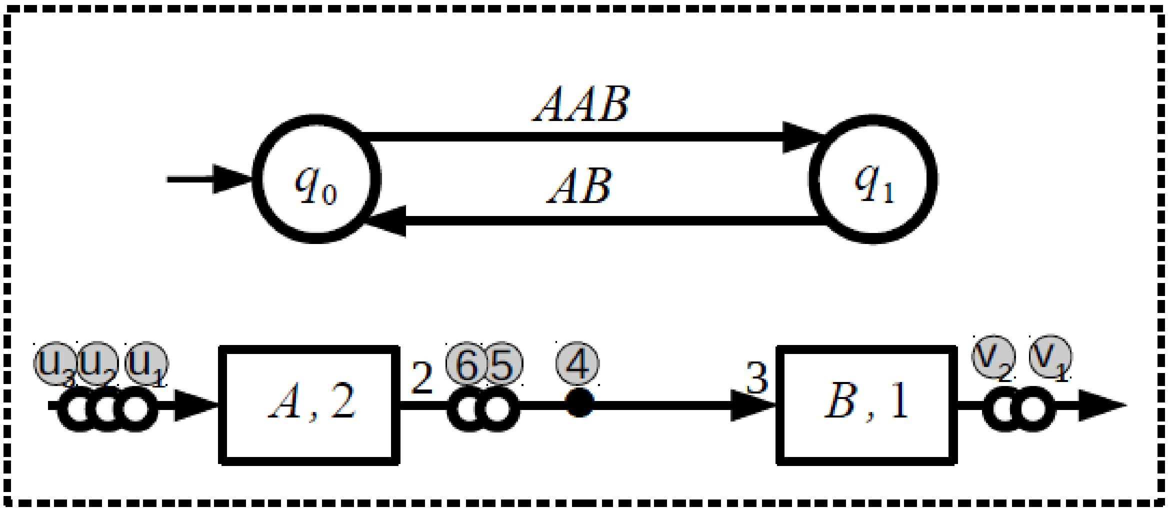


Figure 12: An FSM-SADF graph

The scenario FSM is shown in the upper part of the figure while the associated scenario graphs are shown in the lower part of the figure. The FSM-SADF has two scenarios: "*AAB*" and "*AB*" specifying the orderings of actor firings in the associated scenario graph. With the FSM as shown, the scenario traces generated by the model are as follows: ("*AAB*" "*AB*")*. In the general case, from one scenario to the other, the set of firing actors, their port rates and firing delays will differ. The entailing (max,+) scenario representations and the control specification (the FSM of the FSM-SADF) are used to perform the worst-case performance analysis of the model based on the (max,+)-linear switched system semantics of FSM-SADF involving a structure called the (max,+)-automaton [58]. For details we refer to [59] and [60].

2.3.4. Throughput Analysis of Hierarchical SDF Models

In this section, we discuss throughput analysis for a class of hierarchical SDF models where starting from the bottom level of the hierarchy, the firing of every hierarchical actor implies the execution of one full iteration of the encapsulated graph. In particular, we propose a technique that is an enhancement of the symbolic simulation procedure of [9] that is able to take advantage of the SDF hierarchy semantics.

Our technique combines symbolic simulation and the system-theoretic view on SDF graphs as (max,+) linear systems that recognize the "usual" state-space representation based on the state, input, output and feedthrough matrices (cf. (1.1.2)).

2.3.4.1. Our algorithm

A high-level overview of our algorithm is the following Algorithm 1.

```
Algorithm 1: Throughput analysis for hierarchical SDF models.
  input  : A hierarchical SDF graph $G$
  output: Throughput $Thr$ of $G$
1 $H$ = IsolateHierarchyLevels($G$)
2 foreach hierarchy level $h$ in $H$ do
3     foreach hierarchical actor $a$ in $h$ do
4         $M$ =SymbolicSimulation+($a$, $S[h-1][:]$)
5         $S[h][a] = M$
6     end
7 end
8 $Cg$ = CommunicationGraph($M$)
9 $Thr = 1/$MCM($Cg$)
```

Algorithm 1: Throughput analysis for hierarchical SDF models

The input to the algorithm is a hierarchical SDF graph *G*, while the output of the algorithm is the throughput *Thr* of the graph. We begin the procedure by isolating the hierarchy levels of the graph starting in a bottom-up manner (cf. Line 1). This can be done by employing a suitable variant of the reverse breadth-first search algorithm. Thereafter for each hierarchical actor (cf. Line 3) at the current hierarchy level (cf. Line 4) we perform symbolic simulation in order to obtain the relevant state-space representation of the hierarchical actor (cf. Line 2). The representation (composite matrix that includes the state, input, output and feedthrough submatrices) is stored because later on the symbolic simulation at a higher hierarchy level will need this representation (note that the symbolic simulation in Line 4 is invoked with all the state-space representations belonging to the previous hierarchy level). Finally, when we reach the highest hierarchy level, the symbolic simulation will produce a state-space representation of *G* for which we construct the corresponding communication graph (cf. Line 8). For details on how to construct the communication graph of a (max,+) matrix we refer the interested reader to [31]. The throughput of the graph equals the inverse of the maximum cycle mean of the communication graph (cf. Line 9). Note that the algorithm assumes the existence of a hierarchy in the sense that a graph composed only of atomic actors is assumed to be a hierarchical graph composed of one hierarchical actor that encapsulates the atomic actor composition.

2.3.4.2. Symbolic Simulation

Algorithm 1 as its core part uses symbolic simulation. The symbolic simulation as originally proposed by [9] cannot take advantage of semantics of hierarchical SDF models.

This means that if we are to use the techniques of [9] to compute the throughput of a hierarchical SDF model, we first need to flatten the hierarchy, i.e. to transform the graph to one without it.

As explained in Section 2.3.3.3, symbolic simulation derives the state-space representation of an SDF graph by simulating the graph for exactly one iteration following the iteration schedule. In the schedule, every actor is fired the number of times corresponding to the repetition vector entry for that actor. The procedure requires administration of every token produced and consumed during the iteration. Thus, for graphs with large repetition vector entries, the symbolic simulation can become a bottleneck in the overall throughput analysis flow. In case of hierarchical SDF models (regardless whether the hierarchy is extracted from a flat graph or comes in the specification) we argue that one can avoid the administration of every token. To explain: given a hierarchical actor, the symbolic simulation of [9] would simulate the firing of the actor the number of times given by the encapsulating graph's repetition vector. This implies the firing of every actor (and administration of tokens produced and consumed) of the encapsulation for the same number of times multiplied by the corresponding repetition vector entry in the encapsulation itself and so until the deepest level of the hierarchy. E.g., for the graph of Figure 11, actor *D* would have to be fired $p \cdot 2$ times. It is clear that, across hierarchy levels, depending on the repetition vectors at different levels, we may experience (in the worst-case) an exponential rise in complexity. We argue that we can mitigate the impact that this rise has on the efficiency of symbolic simulation by taking advantage of the hierarchy semantics of SDF. In particular, by using the system theoretic view on SDF sublimed in (1.1.2), we propose a way to avoid administration of all actor firings and token consumptions/productions of the encapsulation by using its state-space representation. In particular, to compute the new state of the hierarchical actor and the timestamps of tokens that are produced at its output interface, it is more beneficial, lightweight and elegant to perform a matrix multiplication (cf. (1.1.2)) than to simulate the encapsulation symbolically. This way we focus only on the tokens that are part of the state (initial ones) and need not to care about others. Furthermore, particular actor firings are compactly encoded in a single matrix. We argue that in case of hierarchical SDF models with large repetition vectors across the hierarchy this approach will mitigate the adverse effect the increase of the graph's repetition vector entries has to the throughput analysis algorithm performance. The modified algorithm for symbolic simulation is outlined as Algorithm 2.

```
Algorithm 2: Compute state-space representation of a hierarchical SDF actor.
    input  : A hierarchical or atomic actor $a$
    input  : State-space representations of actors in lower hierarchy level $S[h][:]$
    output: State space representation $M$ of $a$
 1  $\Sigma$ = SeqSchedule($a$)
 2  $T = \emptyset$
 3  for $i = 1$ to Length($\Sigma$) do
 4      $a = \Sigma[i]$
 5      if $a$ is hierarchical then
 6          $T$ += Fire+($a$, $S[h][a]$)
 7      end
 8      else
 9          $T$ += Fire($a$)
10      end
11  end
12  $M$ = ConstructStateSpaceRepresentation($T$)
```

Algorithm 2: Compute state-space representation of a hierarchical SDF actor.

The algorithm is a modification of Algorithm 1 of [9]. The inputs to the algorithm are the very structure of the hierarchical actor a and the state-space representations ((max,+) matrices) of actors of previous hierarchy levels as the hierarchical actor may as well encapsulate a hierarchical actor from a lower hierarchy level. The output is the state-space representation of the hierarchical actor, i.e. the matrix of (1.1.2). The algorithm first computes the iteration schedule of the actor (cf. Line 1) that gives the ordering of actor firings within the iteration. The schedule is then traversed (cf. Line 3). The firing of each actor is simulated with all the tokens consumed and produced by the firing being administered in container T (cf. Line 2). The crucial difference between Algorithm 2 and Algorithm 1 of [9] is that here, if the actor is hierarchical, we use its state-space representation to compute its new state as well as the symbolic timestamps of tokens it produces in its output channels (cf. Line 6). This way we have avoided the need for flattening the structure and having to administer all the actor firings and tokens consumed and produced by these firings. This has the effect of a compression, as we only focus on initial tokens that are part of the overall state and are carried over to the next hierarchy level. Finally, the tokens can be gathered in the state-space representation for the current actor (cf. Line 12).

2.3.4.3. Example

We demonstrate the operating principles of Algorithm 1 and Algorithm 2 using the running example graph with p = 2. We begin with Algorithm 1. We can isolate two hierarchy levels in the structure. Going bottom up the algorithm encounters the hierarchical actor B for which a state-space representation is derived using Algorithm 2 in the usual manner of [9] as the encapsulation has only atomic actors. The state-space representation is given by (1.1.3). Algorithm 1 now visits the top level of the hierarchy on which it employs Algorithm 2. Consequently, the iteration schedule is computed which has the form AB^2C. The schedule is now simulated. Actor A fires first by consuming token 1. The two tokens produced by its firing carry the symbolic timestamp

$$\begin{bmatrix}0 & -\infty & -\infty & -\infty & -\infty & -\infty\end{bmatrix} \otimes 2 = \begin{bmatrix}2 & -\infty & -\infty & -\infty & -\infty & -\infty\end{bmatrix}. \tag{1.1.4}$$

Note that the symbolic timestamps account for all tokens across all hierarchy levels while the ordering in the vector is given by initial token indices in the figure. After actor A, the schedule dictates that actor B is fired. However, now, to perform this firing, we do not flatten the graph but we use the state-space representation of B in (1.1.3).

The timestamp of the token produced by the first firing of hierarchical actor B can be directly computed from (1.1.2) as follows

$$\begin{aligned}
&\begin{bmatrix}6 & 5 & -\infty & 2\end{bmatrix} \otimes \begin{bmatrix}-\infty & -\infty & 0 & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & 0 & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0\end{bmatrix} \\
&\oplus 6 \otimes \begin{bmatrix}2 & -\infty & -\infty & -\infty & -\infty & -\infty\end{bmatrix} \\
&= \begin{bmatrix}-\infty & -\infty & 6 & 5 & -\infty & 2\end{bmatrix} \oplus \begin{bmatrix}8 & -\infty & -\infty & -\infty & -\infty & -\infty\end{bmatrix} \\
&= \begin{bmatrix}8 & -\infty & 6 & 5 & -\infty & 2\end{bmatrix}
\end{aligned} \tag{1.1.5}$$

According to (1.1.2), the internal state of B advances as follows

$$
\begin{aligned}
&\begin{bmatrix} -\infty & -\infty & 0 & -\infty \\ 7 & 7 & -\infty & -\infty \\ 7 & 7 & -\infty & -\infty \\ 6 & 5 & -\infty & 2 \end{bmatrix} \otimes \begin{bmatrix} -\infty & -\infty & 0 & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & 0 & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0 \end{bmatrix} \\
&\oplus \begin{bmatrix} -\infty \\ 7 \\ 7 \\ 6 \end{bmatrix} \otimes \begin{bmatrix} 2 & -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix} \\
&= \begin{bmatrix} -\infty & -\infty & -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & 7 & 7 & -\infty & -\infty \\ -\infty & -\infty & 7 & 7 & -\infty & -\infty \\ -\infty & -\infty & 6 & 5 & -\infty & 2 \end{bmatrix} \oplus \begin{bmatrix} -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ 9 & -\infty & -\infty & -\infty & -\infty & -\infty \\ 9 & -\infty & -\infty & -\infty & -\infty & -\infty \\ 8 & -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix} \\
&= \begin{bmatrix} -\infty & -\infty & -\infty & -\infty & 0 & -\infty \\ 9 & -\infty & 7 & 7 & -\infty & -\infty \\ 9 & -\infty & 7 & 7 & -\infty & -\infty \\ 8 & -\infty & 6 & 5 & -\infty & 2 \end{bmatrix}
\end{aligned}
\tag{1.1.6}
$$

This leads us to the second firing of *B*. The hierarchical actor is now initialized with the state of (1.1.6) while consuming the second token produced by *A* that carries the symbolic timestamp of (1.1.4). The symbolic timestamp of the second token produced by *B* is therefore calculated as follows

$$
\begin{aligned}
&\begin{bmatrix} 6 & 5 & -\infty & 2 \end{bmatrix} \otimes \begin{bmatrix} -\infty & -\infty & -\infty & -\infty & 0 & -\infty \\ 9 & -\infty & 7 & 7 & -\infty & -\infty \\ 9 & -\infty & 7 & 7 & -\infty & -\infty \\ 8 & -\infty & 6 & 5 & -\infty & 2 \end{bmatrix} \\
&\oplus 6 \otimes \begin{bmatrix} 2 & -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix} \\
&= \begin{bmatrix} 14 & -\infty & 12 & 12 & 6 & 4 \end{bmatrix} \oplus \begin{bmatrix} 8 & -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix} \\
&= \begin{bmatrix} 14 & -\infty & 12 & 12 & 6 & 4 \end{bmatrix}
\end{aligned}
\tag{1.1.7}
$$

Similarly, as in the case of (1.1.6) we can calculate the new state of the encapsulated actor *B*.

$$
\begin{aligned}
&\begin{bmatrix} -\infty & -\infty & 0 & -\infty \\ 7 & 7 & -\infty & -\infty \\ 7 & 7 & -\infty & -\infty \\ 6 & 5 & -\infty & 2 \end{bmatrix} \otimes \begin{bmatrix} -\infty & -\infty & -\infty & -\infty & 0 & -\infty \\ 9 & -\infty & 7 & 7 & -\infty & -\infty \\ 9 & -\infty & 7 & 7 & -\infty & -\infty \\ 8 & -\infty & 6 & 5 & -\infty & 2 \end{bmatrix} \\
&\oplus \begin{bmatrix} -\infty \\ 7 \\ 7 \\ 6 \end{bmatrix} \otimes \begin{bmatrix} 2 & -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix} \\
&= \begin{bmatrix} 9 & -\infty & 7 & 7 & 0 & -\infty \\ 16 & -\infty & 14 & 14 & 7 & -\infty \\ 16 & -\infty & 14 & 14 & 7 & -\infty \\ 14 & -\infty & 12 & 12 & 6 & 4 \end{bmatrix} \oplus \begin{bmatrix} -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ 9 & -\infty & -\infty & -\infty & -\infty & -\infty \\ 9 & -\infty & -\infty & -\infty & -\infty & -\infty \\ 8 & -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix} \\
&= \begin{bmatrix} 9 & -\infty & 7 & 7 & 0 & -\infty \\ 16 & -\infty & 14 & 14 & 7 & -\infty \\ 16 & -\infty & 14 & 14 & 7 & -\infty \\ 14 & -\infty & 12 & 12 & 6 & 4 \end{bmatrix}
\end{aligned} \tag{1.1.8}
$$

Finally, actor *C* can fire by consuming the tokens produced by *B* (cf. (1.1.5) and (1.1.7)) and producing the token carrying the symbolic timestamp

$$
\begin{aligned}
&\left(\begin{bmatrix} 8 & -\infty & 6 & 5 & -\infty & 2 \end{bmatrix} \oplus \begin{bmatrix} 14 & -\infty & 12 & 12 & 6 & 4 \end{bmatrix}\right) \otimes 3 \\
&= \begin{bmatrix} 14 & -\infty & 12 & 12 & 6 & 4 \end{bmatrix}
\end{aligned} \tag{1.1.9}
$$

The firing of *C* completes the iteration and the tokens produced in positions of initial tokens can be gathered up to compose the state-space representation of the graph. In particular, the tokens produced in places of the initial tokens of the underlying encapsulated graph of *B* (tokens 3, 4, 5, 6) are available in (1.1.8). Token 2 is not consumed in the current iteration as at its end it has moved in position of token 1. Therefore, the token in position 1 after the iteration has the symbolic timestamp

$$
\begin{bmatrix} -\infty & 0 & -\infty & -\infty & -\infty & -\infty \end{bmatrix} \tag{1.1.10}
$$

The token produced in place of initial token 2 is the result of the firing of actor *C* and carries the symbolic timestamp of (1.1.9).

When we arrange these tokens into a matrix, we obtain the desired state-space representation of the running example SDF model

$$
M = \begin{bmatrix} -\infty & 0 & -\infty & -\infty & -\infty & -\infty \\ 14 & -\infty & 12 & 12 & 6 & 4 \\ 9 & -\infty & 7 & 7 & -\infty & -\infty \\ 16 & -\infty & 14 & 14 & 7 & -\infty \\ 16 & -\infty & 14 & 14 & 7 & -\infty \\ 14 & -\infty & 12 & 12 & 6 & 4 \end{bmatrix} \tag{1.1.11}
$$

From the communication graph of this matrix, we can obtain the throughput of the graph by applying a maximum cycle mean algorithm [22]. In this case the throughput of the graph is 1/14 iterations per time-unit. From the performance perspective, by doing symbolic simulation in this manner, for the graph of Figure 11, we have replaced *p* standard symbolic simulations of the encapsulation of *B* with *p* matrix multiplications of (1.1.2). We argue that this is a more performance-friendly way for constructing state-space representations of SDF graphs exposing hierarchy and featuring repetition vectors with large entries. In our example, using the SDF^3 tool suite [4] running on an Intel i7-6500U machine operating at 2.50 GHz, we have observed that even for the simple structure of Figure 11 with *p* = 10,000, symbolic simulation of [9] will take about 20 seconds, while the version introduced in this report will complete in about 0.5 seconds.

2.3.5. Compositionality in Synchronous Dataflow: Modular Performance Analysis from Hierarchical SDF Graphs

We now put our hierarchical analysis in the context of work addressing compositionality of SDF in the context of modular code generation of [57]. This is the novel contribution w.r.t. D5.1.

Let us explain. Hierarchical SDF models are not compositional: a composite SDF actor cannot be represented as an atomic SDF actor without loss of information that can lead to rate inconsistency or deadlock [57]. To remedy the former while working in the context of code generation, the authors of [57] propose a compositional abstraction of composite SDF actors called DSSF (Deterministic SDF with Shared FIFOs) that can be used for modular compilation. Nevertheless, the DSSF profiles have no accompanying performance models. In this work, we show how to incrementally build performance models for DSSF profiles using FSM-SADF across arbitrary levels of hierarchy that eventually brings us to the overall system performance model from which a performance metric can be derived using the usual (max,+)-based techniques. We illustrate our approach on a simple hierarchical SDF graph borrowed from [57].

Take a close look at SDF graphs shown in Figure 13. Consider the graph on the right where composite actor *P* is used. Now consider the graph on the left. It seems that the right graph should be equivalent to the left graph if we consider that actors *A* and *B* represent the internal content of *P*. Note that actor *P* has a rate of 3 at its input port and rate 2 at its output port.

This makes sense, as it corresponds to a full iteration of the internal graph in the leftmost part of the figure, namely, (*A*,*A*,*A*,*B*,*B*). The left graph has no deadlock while the right graph deadlocks. In particular, *P* cannot fire because it needs 3 tokens from channel (*C*, *P*) but only two are available.

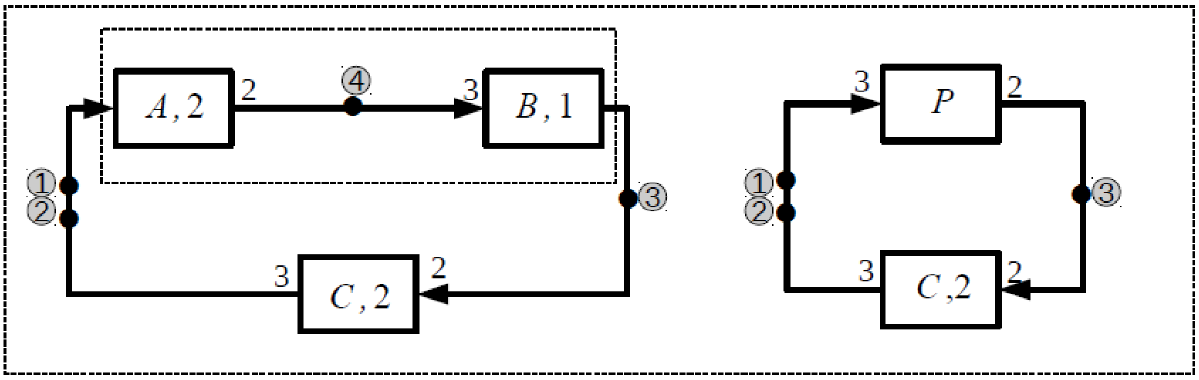


Figure 13: Compositionality in SDF

The example illustrates that composite SDF actors cannot be represented by atomic actors without loss of information that leads to deadlock. The work of [57] deals with the problem by introducing the notion of profiles that characterize a given actor. It is shown that SDF profiles are SDF graphs themselves with an extra ability of associating multiple producers and consumers to a single FIFO queue. Sharing queues gives rise to non-determinism. However, the profiles of [57] are able to guarantee that actors access shared queues in deterministic order. Thus, the profiles are called deterministic SDF with shared FIFOs (DSSF).

Let us consider an example. Take a closer look at Figure 14.

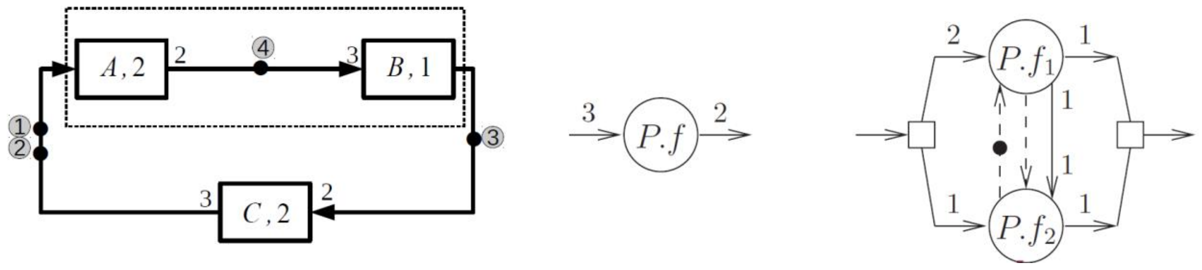


Figure 14: DSSF profiles

The graph shown in the left part of Figure 14 is the same as the left graph of Figure 13 with a composite actor encapsulating atomic actors *A* and *B*. The graph in the middle is a profile with a single actor representing a single firing function corresponding to the internal sequence of firings (*A*,*A*,*A*,*B*,*B*). Such a profile is problematic as described in the discussion related to Figure 13. The right part of Figure 14 is a DSSF with two actors, $P.f_1$ and $P.f_2$. The actors share two FIFO queues depicted as small squares. The accessing of shared queues is made deterministic by introducing dependency edges between the profile actors. This profile is maximally reusable and optimal. This means that no less than two firing functions (actors) can achieve maximal reusability. The firing functions represented by actors $P.f_1$ and $P.f_2$ correspond to firing sequences (*A*,*A*,*B*) and (*A*,*B*), respectively.

To conclude, the concept of DSSF succeeds in forming a compositional abstraction of composite actors that can be used for modular compilation. However, the DSSF concept has no accompanying performance models that can be incrementally built and used in performance (throughput and latency) analysis of the system. The SDF performance models ((max,+) matrices) cannot be used because DSSF is different from SDF in the sense that it involves shared FIFO queues. Therefore, DSSF cannot be captured in SDF. However, FSM-SADF is expressive enough to capture DSSF and serve as a performance model for it.

Let's look again at the right profile in Figure 14. The corresponding FSM-SADF model is shown in Figure 12. Taking the transition labelled with "*AAB*" corresponds to the execution of the firing function $P.f_1$ corresponding to the actor firing sequence (*A*,*A*,*B*), while taking transition "*AB*" corresponds to the execution of the firing function $P.f_2$ corresponding to the firing sequence (*A*,*B*). The static order of accesses to the shared FIFOs is encoded in the FSM.

This shows that FSM-SADF is fit to be used as a performance model for DSSF profiles. As DSSF profiles are built incrementally across arbitrary levels of hierarchy, their performance models must also be built using the hierarchical throughput analysis.

Consider Figure 15.

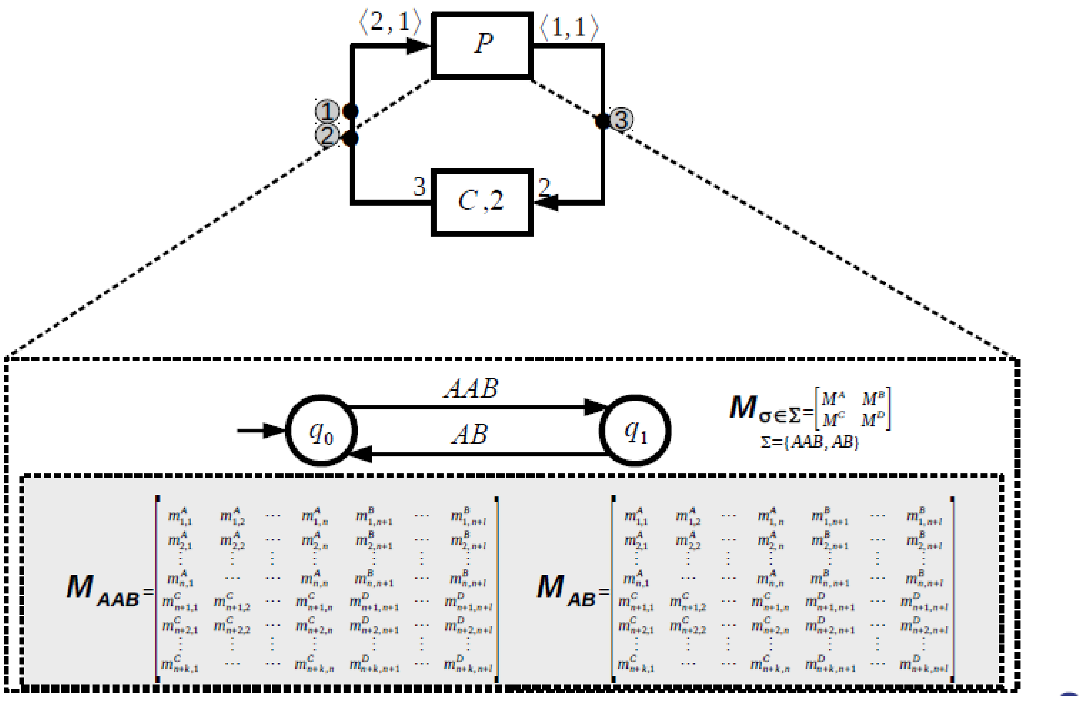


Figure 15: FSM-SADF performance model for DSSF

The upper part of the Figure shows the top-level SDF model while the FSM-SADF performance model belonging to the DSSF profile of actor *P* (right profile in Figure 14) is depicted in the lower part of the figure. To get the performance model for the top-level profile (graph), we need to perform symbolic simulation of the graph. In the simulation, we do not unfold the model, but use the FSM-SADF performance model of the DSSF profile of *P* consisting of the scenario FSM and two (max,+) matrices belonging to scenarios "*AAB*" and "*AB*". In the concrete case:

$$M_{AAB}=\begin{bmatrix}-\infty & 2 & -\infty\\ -\infty & 2 & -\infty\\ 3 & -\infty & 1\end{bmatrix} \text{ and } M_{AB}=\begin{bmatrix}2 & -\infty & -\infty\\ 3 & 1 & 1\end{bmatrix}.$$

The matrices can be used in the symbolic simulation of the top level structure where according to the schedule ((*A*,*A*,*B*), *C*, (*A*,*B*)).

After firing sequence (*A*,*A*,*B*), tokens are produced in places 5, 6 and v_1 carrying the following timestamps:

$$\begin{bmatrix}t_5'\\ t_6'\\ t_{v_1}\end{bmatrix}=\begin{bmatrix}-\infty & 2 & -\infty\\ -\infty & 2 & -\infty\\ 3 & -\infty & 1\end{bmatrix}\otimes\begin{bmatrix}0 & -\infty & -\infty & -\infty\\ -\infty & 0 & -\infty & -\infty\\ -\infty & -\infty & -\infty & 0\end{bmatrix}=.$$

$$\begin{bmatrix}-\infty & 2 & -\infty & -\infty\\ -\infty & 2 & -\infty & -\infty\\ 3 & 0 & -\infty & 1\end{bmatrix}$$

The firing of *C* generates three token hosted by channel (*C*,*P*) with the symbolic timestamps

$$
\begin{aligned}
& t_1^{'} = t_2^{'} = \left(\begin{bmatrix}3 & 0 & -\infty & 1\end{bmatrix} \oplus \begin{bmatrix}-\infty & -\infty & 0 & -\infty\end{bmatrix}\right) \otimes 2 \\
& = \begin{bmatrix}5 & 2 & 2 & 3\end{bmatrix}
\end{aligned}.
$$

Finally, the firing sequence (*A*,*B*) completes the iteration as follows

$$
\begin{aligned}
& \begin{bmatrix} t_4^{'} \\ t_{v_2} \end{bmatrix} = \begin{bmatrix} 2 & -\infty & -\infty \\ 3 & 1 & 1 \end{bmatrix} \otimes \begin{bmatrix} 5 & 2 & 2 & 3 \\ -\infty & 2 & -\infty & -\infty \\ -\infty & 2 & -\infty & -\infty \end{bmatrix}. \\
& = \begin{bmatrix} 7 & 4 & 4 & 7 \\ 8 & 5 & 5 & 6 \end{bmatrix}
\end{aligned}
$$

If we gather the symbolic timestamps of tokens produced in places 1, 2, 3 and 4 we get the performance model of the top-level profile (graph):

$$
M = \begin{bmatrix} 5 & 2 & 2 & 3 \\ 5 & 2 & 2 & 3 \\ 8 & 5 & 5 & 6 \\ 7 & 4 & 4 & 7 \end{bmatrix}.
$$

To conclude, FSM-SADF can be effectively used to incrementally generate performance profiles for DSSF actors. This way, we have added a powerful performance analysis framework to enhance the DSSF-based design flow by making it applicable in a real-time setting without the need to unfold the hierarchical structure. This means that performance profiles as well as the DSSF profiles can be used in any context provided as a component library.

2.3.6. Analysis flow

The envisioned analysis flow of the SDF^3 tool is shown in Figure 16.

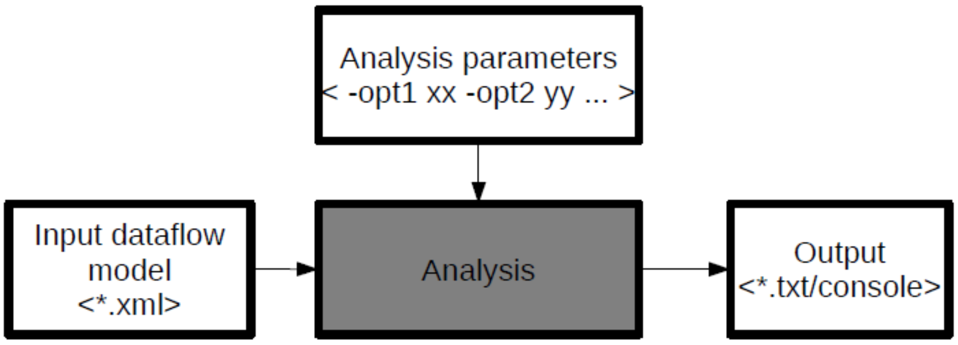


Figure 16: SDF^3 analysis flow

The core of the flow is the actual analysis engine of SDF^3 (the grey box in Figure 16). The algorithms implemented in the analysis engine are based on the theoretical discussions presented above. The input to the analysis engine is an XML file that contains the hierarchical dataflow model of the application that is to be analysed. The results of the analysis (throughput and latency quantities) can be viewed on the terminal or in a specific file.

2.3.7. Conclusion

In this work, we focused on throughput analysis of hierarchical dataflow models. We have shown ways how to develop new methods that can take advantage of the hierarchical semantics of SDF. We base our method on the existing state-of-the-art symbolic simulation method that we combine with the system-theoretic view on SDF graphs where hierarchy elements need not be flattened during the symbolic simulation but where their state-space representation can be used instead. This way we remove the need for the repeated simulation of encapsulated subgraphs of hierarchical actors that includes the administration of actor firings and all of the produced and consumed tokens. By using a state-space representation, we can focus on the tokens that are part of the state by means of matrix multiplications. We argue that symbolic simulation endowed with this feature can help to mitigate the difficulties that the standard flavour experiences when dealing with graphs with large repetition vectors.

In addition (beyond the progress reported in D5.1) we have generalized our hierarchical throughput analysis to the case of FSM-SADF, which is a dynamic dataflow model, and applied it in the context of DSSF by enhancing the modular abstraction of DSSF primarily intended for modular code compilation with reusable performance profile models. The enhancement enables performance analysis of profiles without the need for graph unfolding, which promises a significant improvement of the scalability of the analysis.

2.3.8. Current status

To add support for analysis of hierarchical SDF models the analysis engine of SDF^3 needs to be enriched with implementations of Algorithm 1 and Algorithm 2 with the addition of FSM-SADF related specifics presented. Furthermore, the input XML specifications of dataflow models need to be able to account for hierarchical actors the structure of which may be nested in the encapsulating's model description or given in a separate XML file.

With respect to KPIs, we target [KPI1.2] (improve tool performance). The progress achieved so far is accounted for in the methodology presented in the previous sections that is ready to be implemented in the tool with expectation of significant performance gains as simulation of the operational semantics of SDF is replaced by simple matrix multiplications yielding a complexity decrease of the analysis.

2.3.9. Use case

All the activities described above are part of VDL's use case (TNO_UC_01).

2.4. RTANA2: System-Level Timing Analysis of Real-Time System Models

2.4.1. Introduction

Embedded safety-critical systems must not only be functionally correct, but must also provide timely service. Timing and scheduling analysis is an important method in the design of such systems and is used for determining timing properties of a system, aiming at understanding and optimizing the timing of systems, and to verify that timing requirements are met.

Much work on real-time scheduling analysis has been done with increasing expressive power along the evolution of the analysis methods. These methods can be broadly classified as analytical methods on the one hand and computational methods on the other hand. Methods of the first class provide an efficient analysis by abstracting from concrete behaviour in terms of event streams. Scheduling is analysed by evaluating fixed-point equations. While such analyses are fast and scalable up to large systems, they often (depending on the system) deliver pessimistic results [10]. Methods belonging to the second class, such as model-checking on automata, consider the state space of a model and explore all actual execution paths resulting in precise figures for end-to-end timing of functional chains. In addition, more complex safety properties can be verified. Of course, this comes at a price: model-checking is computationally expensive and can suffer from state-space explosion.

As part of the OFFIS contribution to the ASSUME project, OFFIS is extending a computational timing analysis tool RTANA2. Given a real-time model the tool determines properties of that model such as response times per task, end-to-end latencies of functional chains of tasks, maximum number of pending activations per task and a task matrix per resource containing for each task which other tasks may pre-empt it on that resource. Further, observer automata can be used to check whether the real-time model satisfies a more complex requirement. The tool will then determine whether such an automaton enters a deadlock.

2.4.2. Input

The input for the tool is a real-time model consisting of a task network deployed on a set of resources, where each resource is assigned a scheduling strategy. The task network consists of a set of tasks, where each task has a set of input and output ports. Via connections of output ports to input ports, a precedence relation on the tasks is defined. Each input port of a task represents a possible source of activation of a task, meaning it is activated whenever an event is observed on any of its input ports. Input synchronizations are expressed by multiple connections to the same input port. In the example depicted in Figure 17, the task is either activated by an event *a* observed on input port *i1* or when an event *b* and an event *c* are observed on input port *i2*.

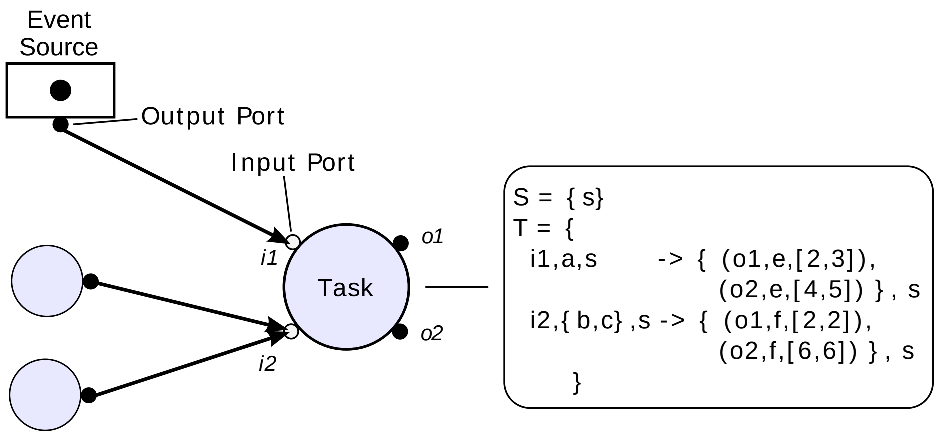


Figure 17: Task network example

Each activation causes a delay for processing, depending on the activating event and the state of a task. The delays are taken from intervals with best- and worst-case bounds for each output port on which the task sends an event. Referring to Figure 17, when activated by an event *a* on port *i1*, the task sends an event *e* on port *o1* after 2 to 3 time units. A characterization of the delay intervals can be obtained from measurement (e.g. by tracing the actual implementation), by analysis, the so-called worst-case execution-time (WCET) analysis, or by estimations (esp. in early phases of development). Where a preceding task is unknown, assumptions about the timing behaviour of the environment can be expressed by means of event sources. Such an event source has parameters like period, jitter, and offset, characterizing an event stream.

To define a real-time model, a task network is deployed on a set of resources. Typically, a real-time model contains more tasks than resources, meaning access to resources needs to be scheduled. Currently, the tool only supports scheduling according to fixed priorities with or without pre-emption of tasks.

2.4.3. Method description

The tool belongs to the class of computational analyses and relies on model checking based on ω-regular languages. In the following, the basic principles of the analysis method are described. Further details can be found in [11].

A discrete time model is assumed, where time is divided into slots of pre-defined equal length. All scheduling-related events, such as task arrivals, completions, and pre-emptions take place at these discrete time points. The behaviour of a real-time model is represented by means of an ω-regular language, which is computed by the method based on the real-time model and its parameters. For the representation of ω-regular languages, finite state machines (FSMs) are an obvious choice. The analysis is done by creating an FSM on-the-fly for event sources representing assumed environment behaviour and each resource, like ECUs and buses, with its allocated tasks. The FSM constructed for a simple strictly periodic event source, with a period ρ and jitter *j* = 0, is sketched on the left-hand side of Figure 18. The first event *c* is emitted non-deterministically within the first ρ time units due to the invariant on the upper bound of clock *x* in location l_1. Afterwards it is emitted each ρ time units, which is enforced by the invariant on the upper bound of clock *x* in location l_2 and the guard of the transition sending *c*, which constrains the clock to have at least progressed ρ time units.

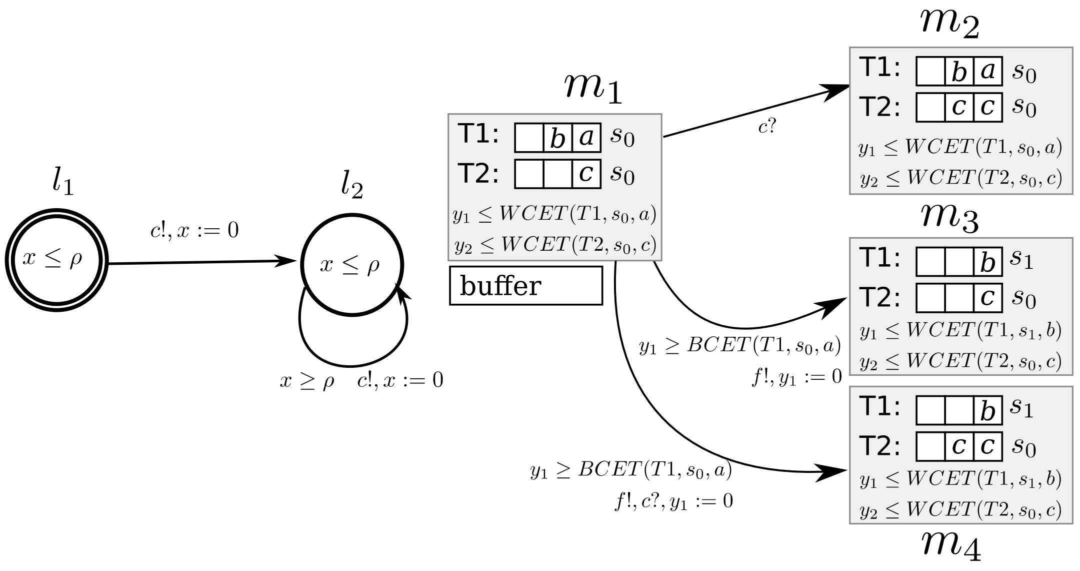


Figure 18: State representation for event model (left) and resources (right)

On the right-hand side of Figure 18 the encoding of a state of a resource is shown. Each task allocated to it is represented by a data structure similar to the one shown in Figure 18, consisting of a buffer storing activation events and the current state of the task. Activations are stored in the buffer as shown for an activation of task T2 by event *c*. Each task has its own private clock. The scheduling strategy determines which of the clocks of its tasks are stopped. In the shown example task T1 has higher priority. As tasks have an interval [BCET, WCET] for each output port, non-determinism may be introduced in the FSM (not shown in Figure 18). Each FSM is input receptive to all events. So if an FSM sends an event *c*, all other FSMs have a transition that synchronizes with that event. FSMs that are not sensitive to some received event *c* do not take part in the synchronization. The states of the real-time model are hierarchical. Figure 19 illustrates a top-level system state, consisting of a vector of states of event sources and resources, as well as a valuation for each clock of the real-time model. Assuming that BCET(T1,s_0,*a*) < ρ < WCET(T1,s_0,*a*) holds, the figure shows the successor states of some state with location vector (l_2,m_1) and a valuation of 0 for each clock. The local states are depicted in Figure 18: The event source could send an event *c* after ρ time units, leading to a successor state where its clock *x* is reset, the clock y_1 of the scheduled task has progressed by ρ time units and the clock y_2 of the suspended task is unchanged. If in addition task T1 sends an event *f*, also its clock y_1 is reset. If only the task T1 sends an event, its clock y_1 is reset, y_2 is unchanged and the clock *x* of the event source has progressed to a value within the range [BCET(T1,s_0,*a*), ρ-1]. For each value of this interval, a separate system state is constructed.

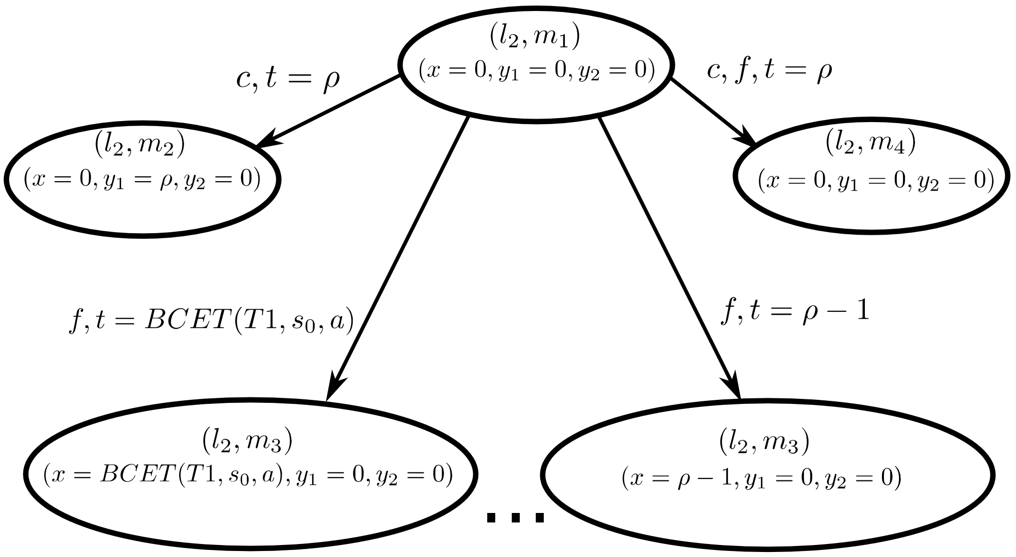

(l_2, m_1)
$(x = 0, y_1 = 0, y_2 = 0)$
$c, t = \rho$
$c, f, t = \rho$
(l_2, m_2)
$(x = 0, y_1 = \rho, y_2 = 0)$
(l_2, m_4)
$(x = 0, y_1 = 0, y_2 = 0)$
$f, t = BCET(T1, s_0, a)$
$f, t = \rho - 1$
(l_2, m_3)
$(x = BCET(T1, s_0, a), y_1 = 0, y_2 = 0)$
(l_2, m_3)
$(x = \rho - 1, y_1 = 0, y_2 = 0)$
...

Figure 19: System state representation

The complete behaviour of the real-time model is determined by iteratively computing the product FSM of all event source FSMs and resource FSMs. The algorithm is a mild variation of the one proposed in [41], which is an approach to reachability analysis of closed timed automata based on encoding their discrete time semantics by means of a data structure called time-darts. Based on the reachable state space, typical timing properties like end-to-end latencies can be computed by means of path analysis, involving breadth-first search (BFS) on the product FSM. At each state where a task in the chain is finished, the algorithm starts a BFS to find the paths to completion of the successor task in the chain.

2.4.4. Interface to concurrency defect analyses

Besides typical timing properties like response times per task and end-to-end latencies of functions, the tool can also determine, as a by-product, possible pre-emption scenarios between tasks. As explained in Section 2.4.3, the computation of next states of a resource takes into account which tasks are active and selects a (subset) of the tasks to be running during the next step. Depending on the scheduling strategy it might be the case that a task gets pre-empted by the new activation of a higher priority task. If so, the analysis marks the task executing in the current state as being pre-empted by the task executing in the next state. This marking is then extended to all other tasks that are active in the current state. So if a task B is executed during the current state and task C is active but not executing and a task A is executed during the next state pre-empting task B, then both tasks B and C are marked as being pre-empted by task A.

Further, the analysis also takes corner cases into account like seemingly simultaneous task activations resulting from the underlying discrete time model. If a set of different tasks are activated at the same discrete step, the analysis assumes that these activations can happen in arbitrary order. This ensures that pre-emption scenarios are sound independently from the length of discrete time slots.

The result of the analysis is a function $PREEMPT: T \times T \to \{true, false\}$ on the set T of tasks allocated to a resource. For each resource, such a function is computed. The function assigns to each pair (τ_X, τ_Y) of tasks a boolean value denoting whether a scenario is possible where τ_X can be pre-empted by τ_Y. This function provides valuable insights where concurrency defects might occur and in particular where concurrency defects can be excluded based on the function $PREEMPT$ inferred from the timing behaviour of the system.

2.4.5. Current status and future development

In the ASSUME project we focused on two aspects to improve the timing analysis tool:

- We add features to the analysis, like support for more scheduling strategies, for multi-core architectures and for functions executed by operating system threads.
- We investigate methods to alleviate the state space explosion problem.

Regarding the first item, one extension is to provide support for scheduling strategies that are used by commercial operating systems. For example, the OSEK standard defines a scheduling strategy where pre-emptable and non-pre-emptable tasks can be mixed on the same operating system. Further, tasks can be grouped. All tasks within a group are non-pre-emptable among each other, but can be pre-empted by tasks with a higher priority than the highest priority of all tasks within a group. Another extension is to lift the restriction that per resource at most one task is executing. This can then be used to analyse multi-core systems where tasks can migrate between different cores.

During the ASSUME project we have realized a refinement of the task model, allowing the modelling of top-level functions executed by operating system tasks. Given that characterizations of execution time intervals of these functions are available, the analysis can determine timing properties like response times and end-to-end latencies based on these functions, as well as pre-emption scenarios based on functions. As an example, consider the AUTOSAR standard. An AUTOSAR model defines a set of so-called runnables, which in turn are executed by operating system tasks. So the extension can be used to get more fine-grained results based on runnables instead of tasks.

Regarding the second item, we have three lines of attacking the state space explosion: The first one is to improve the representation of the state space by considering better symbolic methods, especially regarding time progress. The second one is to develop a compositional method, where a real-time model is sliced into different sub-models that can be analysed separately. The third one is to consider combinations with analytical approaches.

2.4.5.1. Verification based on Difference Bound Matrices

During the ASSUME project we have added support for an alternative verification engine based on difference bound matrices (DBMs), a data structure often used for the analysis of timed automata (cf. [42]). It is well known that though discrete time approaches like time-darts or BDD-based representations of a discretized state space perform well for some timed-automata models [41][43], zone-based methods outperform such discretized methods when the size of the constants in clock constraints of a timed-automaton is increased [43]. This effect can be seen in the exemplary system states shown in Figure 19: Multiple system states with location vector (l_2,m_3) are reachable from the state with location vector (l_2,m_1). The transitions leading to these states have the same events, only the time progress differs. Using DBMs, such phenomena are avoided because the constraint BCET(T1,s_0,a) ≤ x ≤ ρ-1 is stored in the DBM resulting in just one successor state. In case pre-emptive scheduling strategies are used, we needed to overcome the problem that clocks cannot simply be stopped if DBMs are used. This is because the reachability problem is undecidable for the resulting model of stop-watch automata. To solve this problem, we made use of an approach similar to the one illustrated in [44]: Instead of stopping clocks, the clock constraints of pre-empted tasks are modified in a monotonically increasing way according to the time progress of running tasks. Consider the following example: The invariant of the clock y_2 of some task T2 is given by $\leq WCET(T2)$ and the guard for its termination is $\geq BCET(T2)$ in some state and T2 is preempted by T1, which has the private clock y_1. Then the new invariant of y_2 becomes $\leq WCET(T2) + ub(y_1, S)$ and the new guard becomes $\geq BCET(T2) + lb(y_1, S)$, where

$ub(y_1, S)$ is the upper bound of the clock y_1 in the system state S where T1 finishes its execution, and $lb(y_1, S)$ is the lower bound. Note that the interval of the valuation of clock y_1 at state S determines for how long T1 has executed. This interval is contained in the interval defined by the BCET and WCET of T1. This ensures that the actual time T2 spends executing reflects the amount of time it is preempted by T1. Not surprisingly, this approach turned out to be more efficient than the encoding based on the time-darts data structure. Depending on the real-time model, it can however lead to over-approximations of its behaviour. To see this, let us instantiate the previous example with concrete numbers for the execution time intervals of T1 and T2: Assume T1 has an execution time interval [3,9] and T2 has an execution time interval [4,4]. Further, we assume that T1 is always activated 3 time units after T2 and reactivation of T1 and T2 occurs after both tasks have finished their execution. It is easy to see that T2 has a response time interval of [7,13]. Resetting the clocks y_1 and y_2 when the respective task gets activated, the difference $y_2 - y_1$ is 3 when T1 gets activated. Given that T1 is activated 3 time units after T2, preempting T2 for [3,9] time units, T2 needs to execute for exactly 1 time unit after T1 has finished its execution, because its execution time interval is [4,4]. This relation is independent of the actual time span of the range [3,9] by which T2 got preempted. While the DBM analysis engine indeed correctly calculates the response time interval, it over-approximates the relation of the clocks y_1 and y_2 at the time T1 finishes its execution. The approach of increasing the clock invariant and guard of y_2 by the amount of preemption by T1 results in the condition $7 \leq y_2 \leq 13$ for T2 to finish its execution. Since the difference of the clocks is $y_2 - y_2 = 3$ when T1 gets activated and T1 executes for [3,9] time units, the valuation of y_2 at the point T1 finishes its execution is $6 \leq y_2 \leq 12$. Combined with the condition for y_2 we get that T2 finishes between 0 and 7 time units after T1 has finished its execution. The cause of the over-approximation is that the increase of the invariant and guard of y_2 must exactly be the amount of time T1 has executed. If T1 has executed for 3 time units, the new interval where T2 can finish its execution must be [4,4] + [3,3], if it has executed for 4 time units, the new interval must be [4,4] + [4,4], and so on. This would lead to the correct relation between the times where T1 and T2 finish their execution. However, using the usual encoding in DBMs [42], it is impossible to accurately capture this relation.

2.4.5.2. Verification based on Future Difference Bound Matrices:

To tackle the over-approximation of the DBM-based verification engine, we developed another engine, which we refer to as future-DBMs in the following. This approach is based on the same data structure, a square matrix of linear clock constraints. The algorithm for computing the reachable states is quite different though. The basic idea is to remove the special zero clock of DBMs that is used to reason about the valuation of each clock [42]. So future-DBMs only represent a constraint system on the differences between clocks, but do not explicitly track the valuation of a clock. This simplification comes at a price: Symbolic reachability analysis using future-DBMs cannot be applied to general timed safety automata (as supported by Uppaal), but only to a subclass similar to Event Recording Automata [66]. In event recording automata, clocks have a fixed predefined association with the symbols of the input alphabet. The valuation of a clock of the input symbol a is a history variable whose value always equals the time of the last occurrence of a relative to the current time. In contrast to timed automata, clocks thus cannot be reset arbitrarily. We also require such a fixed predefined association of clocks to input symbols. A guard on a clock may only appear on a transition where its associated symbol occurs. We also require that for each clock, all transitions from a given location have the same constraints regarding the clock. An additional condition regarding clock guards to be fulfilled is that a clock guard may not change from location to location, but only on the next occurrence of a. So the constraint on a clock is solely determined by the input word. Further it is required that a clock constraint is either a closed finite interval or a clock is not constrained at all (corresponds to the interval $]-\infty,\infty[$). Whenever a symbol occurs on a transition, its associated clock is reset. A clock may also be reset by a transition, if it is not constrained in the current location. While these are quite some restrictions, the resulting automaton class is still powerful enough for a broad range of schedulability and timing analyses.

In order to discuss the algorithm based on future-DBMs, let us first briefly review the algorithm for symbolic reachability analysis for timed automata based on DBMs [42]. First we introduce the syntax of timed automata and operational semantic.A timed automaton A with a set C of clocks is a tuple $\langle N, l_o, E, I\rangle$, where

- N is a finite set of locations
- $l_0 \in N$ is the initial location
- $E \subseteq N \times B(C) \times \Sigma \times 2^C \times N$ is the set of edges and
- $I: N \to B(C)$ assigns invariants to locations. These invariants typically constrain the maximum valuation of clocks.

To keep track of the changes of clock values, functions known as *clock assignments* are used, mapping C, the set of clocks, to the non-negative reals $\mathbb{R}_+$. Let u, v denote such functions and use $u \in g$ to mean that the clock values given by u satisfy the guard $g \in B(C)$. For $d \in \mathbb{R}_+$, let $u + d$ denote the clock assignment that maps all $x \in C$ to $u(x) + d$. For $r \subseteq C$, let $[r \mapsto 0]u$ denote the clock assignment that maps all clocks in r to 0 and agrees with u for the other clocks in $C \backslash r$. The semantics of a timed automaton is a timed transition system where states are pairs $\langle l, u\rangle$, and transitions are defined by the rules:

- $\langle l, u\rangle \xrightarrow{d} \langle l, u+d\rangle$ if $u \in I(l)$ and $(u+d) \in I(l)$ for a non-negative real $d \in \mathbb{R}_+$
- $\langle l, u\rangle \xrightarrow{a} \langle l', u'\rangle$ if $\langle l, g, a, r, l'\rangle \in E$, $u \in g$, $u' = [r \mapsto 0]u$ and $u' \in I(l')$

The reachability analysis is typically based on a symbolic semantics, that in turn is based on the notion of *zone graphs*. A zone is the solution set of a clock constraint, i.e. the maximal set of clock assignments satisfying the clock constraint. A DBM represents such a zone. A symbolic state of a timed automaton is a pair $\langle l, D\rangle$ representing a set of states of the automaton, where l is a location and D is a zone. A symbolic transition describes all the possible concrete transitions from the set of states. We define $D^{\uparrow} = \{u + d \mid u \in D, d \in \mathbb{R}_+\}$ and $r(D) = \{[r \mapsto 0]u \mid u \in D\}$. The symbolic transition relation $\rightsquigarrow$ over symbolic states is defined by the following rules:

$$\langle l, D\rangle \rightsquigarrow \langle l, D^{\uparrow} \wedge I(l)\rangle$$

$$\langle l, D\rangle \rightsquigarrow \langle l', r(D \wedge g) \wedge I(l')\rangle \;\; if \; \langle l, g, a, r, l'\rangle \in E$$

The symbolic semantics corresponds closely to the operational semantics in the sense that $\langle l, D\rangle \rightsquigarrow \langle l', D'\rangle$ implies for all $u' \in D'$, $\langle l, u\rangle \to \langle l', u'\rangle$ for some $u \in D$. Further, a state $\langle l, u\rangle$ reachable in the semantics implies that also a state $\langle l, D\rangle$ with $u \in D$ is reachable in the symbolic semantics. Based on this, a symbolic computation of the reachable states of a timed automaton can be realized by the following algorithm:

```
$PASSED = \emptyset, WAIT = \{\langle l_0, D_0 \rangle\}$
$\textbf{while}\ WAIT \neq \emptyset\ \textbf{do}$
    $take\ \langle l, D \rangle\ from\ WAIT$
    $\textbf{if}\ D \nsubseteq D'\ for\ all\ \langle l, D' \rangle \in PASSED\ \textbf{then}$
        $add\ \langle l, D \rangle\ to\ PASSED$
        $\textbf{for all}\ \langle l', D' \rangle\ such\ that\ \langle l, D \rangle \rightsquigarrow \langle l', D' \rangle\ \textbf{do}$
            $add\ \langle l', D' \rangle\ to\ WAIT$
        $\textbf{end for}$
    $\textbf{end if}$
$\textbf{end while}$
```

The algorithm mentioned above can be used to compute the state-space of an automaton under consideration. Starting with the initial state $\langle l_0, D_0 \rangle$, it iteratively computes the reachable symbolic states according to the transition relation of the automaton.

Technically, a zone D is represented by a DBM and the operations and relations on zones have corresponding operations on DBMs. These operations are explained in detail in [42]. For example, the operation $D^{\uparrow}$ corresponds to an operation $up(D)$. That operation removes the upper bounds on all clocks by setting them to ∞. More precisely, it is the difference constraint $x - x_0$ that is set to $\leq \infty$, where x_0 is the special clock mentioned before whose valuation is always zero. The operation $r(D)$ corresponds to an operation $reset(D, x)$. That operation sets the difference constraint $x - x_0$ to 0 and updates the differences $x - y$ accordingly. The implemented scheme is to alternate between computation of time delay (the first rule of the symbolic transition relation) and computation of successor states reached by transitions of the automaton (the second rule of the symbolic transition relation).

Given the described restrictions on the kind of automata, we can derive an algorithm for symbolic reachability analysis that neither needs the operation $up(D)$, nor the operation $reset(D, x)$ and also does not need the zero clock. First we introduce some notations: Let $g(x, l)$ be the constraint associated with clock x in location l, x_a be the clock associated with input symbol a, and a_x the input symbol associated with clock x. Further we denote by $lb(g)$ and $ub(g)$ the lower and upper bounds of a guard interval. The same notation is used to refer to the bounds of a clock difference $x - y$.

The algorithm starts with an initial symbolic state $\langle l_0, D_0 \rangle$, where D_0 is the conjunction over all clock differences $x - y = g(x, l_0) - g(y, l_0)$. If one of x or y is not constrained, then the difference constraint is $x - y =]-\infty, \infty[$. Now the differences of clocks in D_0 represent the relationships of the points in time at which the transitions of the automaton can fire relative to each other and their corresponding clocks are reset. The idea of the algorithm is to update these relations between clocks in D whenever a transition is taken. The general algorithm for reachability analysis is the same as for the DBM-based reachability analysis, only the symbolic states and their computation differ. Let $\langle l, D \rangle$ be a state added to $PASSED$. The algorithm examines all transitions $\langle l, g, a, r, l' \rangle \in E$ and first computes $D^{\leftarrow}(a) = D \wedge (x_a - y \leq 0)$, where y ranges over all clocks $y \in C$, such that $g(y, l) \neq]-\infty, \infty[$. The constraint system $D^{\leftarrow}(a)$ thus reflects that the chosen transition is taken before all other pending transitions. During this step $D^{\leftarrow}(a)$ might become inconsistent, i.e. its solution set might be empty. This can happen if some transition cannot be taken before another pending transition. In the next step, the algorithm updates the clock differences to other clocks by considering the constraints for clocks in r in the next location

l'. The following constraint system is computed: $fw(D^{\leftarrow}(a), g, r) = \{[r \mapsto g]u \mid u \in r(D^{\leftarrow}(a))\}$. As before, $r(D^{\leftarrow}) = \{[r \mapsto 0]u \mid u \in D\}$ denotes the operation mapping all clocks in r to zero. However, since the future-DBMs do not have a zero-clock, its technical realization is different from $reset(D, x)$: For any pair $x, y \in r$, $r(D^{\leftarrow})$ just sets $x - y = [0,0]$. The term $[r \mapsto g]u$ denotes the clock assignment that maps all clocks x in r to $u(x) + d$, with $d \in g(x, l')$, and agrees with u for the other clocks not in r. So this operation increases the difference $x - y$ by the guard interval $g(x, l')$ for all clocks $x \in r$. So far this algorithm works for constructing the symbolic state space for the considered class of automata when no pre-emption occurs. The symbolic semantics is given by the following rule:

$$\langle l, D\rangle \rightsquigarrow^{a} \langle l', fw(D^{\leftarrow}(a), g, r)\rangle \quad if\ \langle l, g, a, r, l'\rangle \in E$$

In order to deal with pre-emption of a task P by task N we have to modify our rule. Observe that a task N can only pre-empt P if N has been idle in the current state $\langle l, D\rangle$, meaning its clock was unconstrained in l, and N is non-idle in the next state $\langle l', D'\rangle$, meaning its clock is constrained to some finite interval (the execution time interval of N). Now suppose that clock y belongs to the preempted task P and clock x belongs to the pre-empting task N. As thoroughly discussed in the paragraph about verification using DBMs, the time interval P has left to finish its execution is independent from the actual amount of pre-emption suffered from N. Therefore, we let $[r \mapsto g, pr]u$ denote the clock assignment that maps all clocks x in r to $u(x) + d$, with $d \in g(x, l')$, and all preempted clocks y to $u(y) - e$, with e equal to the value the pre-empting clock x is mapped to, and agrees with u for the other clocks. So this operation increases the difference $x - y$ by the guard interval $g(x, l')$ for all clocks $x \in r$, increases the difference $y - z$ of a preempted clock y by the guard interval $g(x, l')$ and sets the difference $y - x$ of a preempted clock y and a pre-empting clock x to $y - x_a$. Setting the difference $y - x$ to $y - x_a$ is because $y - x_a$ represents the time interval a preempted task P has left to finish its execution. The symbolic semantics is thus given by the rule:

$$\langle l, D\rangle \rightsquigarrow^{a} \langle l', fw(D^{\leftarrow}(a), g, r)\rangle \quad if\ \langle l, g, a, r, l'\rangle \in E$$

where $fw(D^{\leftarrow}(a), g, r) = \{[r \mapsto g, pr]u \mid u \in r(D^{\leftarrow}(a))\}$.

2.4.5.3. Compositional Analysis Strategy Combining State-based and Analytical Methods

During the ASSUME project we have realized a compositional analysis strategy, where the analysis is applied in combination with an analytical analysis approach. Figure 20 shows a diagram of the resulting analysis strategy. All of the steps are completely automated. First the given real-time model is sliced into parts, each of which can be analysed independently of the other slices. Analysis results obtained for each slice can be merged easily. For each constructed slice, an analytical analysis is started, as well as an analysis based on time-darts. The latter is however setup to create a potential under-approximation of the behaviour of the real-time model. For the analytical approach, we have chosen pyCPA[1]. This analytical approach renders potentially over-approximated results. After having executed both analyses for a slice, the results reported by the analyses are compared. If they are equal, the result must be exact. If they are not equal, then the exact result must be somewhere in the range formed by the under- and over-approximation. The analysis then tries to further reduce the slice by removing objects from the real-time model, for which it has exact results. Of course, an object can only be removed, if no other object directly or indirectly depends on it. On the resulting slice, an exact computational

[1] A research-grade implementation of compositional performance analysis (https://bitbucket.org/pycpa/pycpa/)

analysis is carried out using the future-DBM-based verification engine. This delivers the missing exact results for a slice. The exact results determined for each slice are then aggregated forming the results for the entire real-time model.

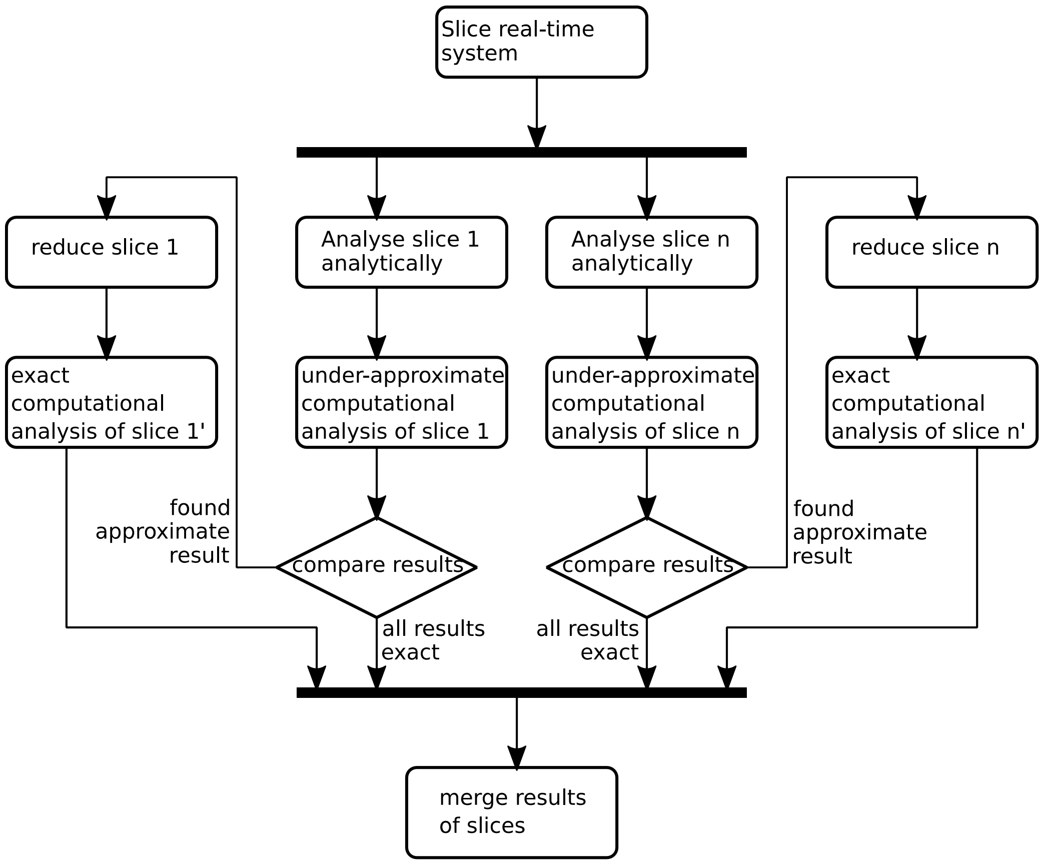
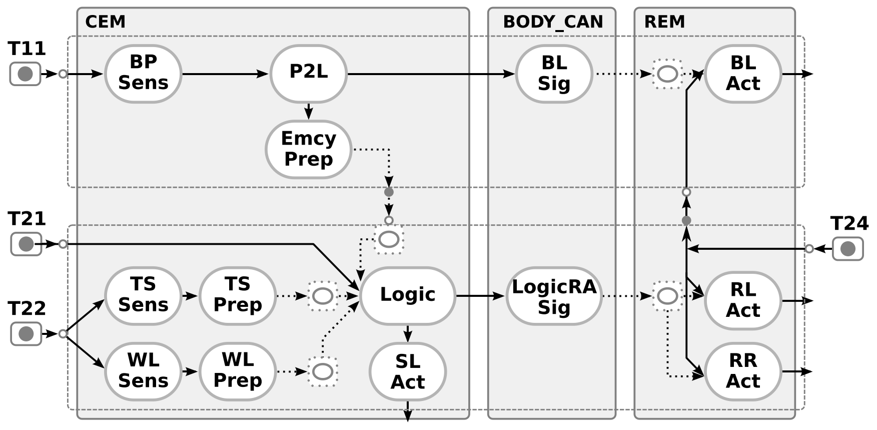

Slice real-time system
reduce slice 1
Analyse slice 1 analytically
Analyse slice n analytically
reduce slice n
exact computational analysis of slice 1'
under-approximate computational analysis of slice 1
under-approximate computational analysis of slice n
exact computational analysis of slice n'
found approximate result
compare results
compare results
found approximate result
all results exact
all results exact
merge results of slices

Figure 20: Flow Diagram of compositional combined real-time analysis

The analysis strategy described above is geared towards obtaining exact performance figures. However, it can be easily adapted for the case, where timing and performance requirements are given and the analysis shall just determine whether these are met or not. In this scenario, one can often avoid the step of having to execute an exact computational analysis.



Task	Execution Time (ms)
BPSens	1
P2L	2
EmcyPrep	2
BLSig	0.25
BLAct	2
Logic	4
TSSens	2
TSPrep	1
WLSens	1
WLPrep	1.5
SLAct	2
LogicRASig	0.25
RLAct	2
RRAct	2

Figure 21: Case-Study: Exterior Light Management System

In order to evaluate the DBM-based verification engine and the combination with analytical approaches, we compared the performance of the analysis strategies on a case study from the automotive domain depicted in Figure 21. The system under investigation has two functions that control the signal lights of a car according to the driver's actions. The *BrakeLight* function controls the rear brake lights according to the driver's brake pedal position. The brake pedal position is periodically sensed and preprocessed by the tasks *BPSens* and *P2L*, respectively. The latter sends the generated control values via the bus to the actuator task *BLAct*. The *TurnLight* function controls the turn lights according to the position of the turn switch and the warning light switch at the driver console. The function has sensing and pre-processing tasks for the turn switch (*TSSens*, *TSPrep*) and the warning lights (*WLSens*, *WLPrep*). The pre-processed data is read by the central *Logic* task, which generates control values for the individual actuator tasks. The data is sent via the CAN bus to the actuator tasks *RLAct* and *RRAct*, which are hosted at the ECU REM. The task also generates control data for the switch lights that reside in the driver console (*SLAct*). The front lights are omitted in the model. The system also implements an emergency brake signalling feature. Whenever the driver performs an emergency brake (which is indicated by a brute force brake action), then the car should activate both rear turn lights in order to signal following drivers about the emergency brake situation. The task *EmcyPrep* calculates whether an emergency brake took place. The dotted elements with a rectangle symbol indicate shared variables that store control values. The variables at the ECU *CEM* store the data from the pre-processing tasks. Whenever the *Logic* task is activated, it reads the stored values to generate the actuator control values. The shared variables at the ECU *REM* store these values for the actuator tasks. In the analysis model, we omit these variables. All resources of the model are scheduled using FPS. The event sources *T11*, *T21* and *T22* each have a period of exactly 20 and a jitter of 0. The event source *T24* has a period of 10 and a jitter of 0. The trivial execution time intervals are noted in Figure 21. Two end-to-end effect chains were of interest for this case-study: One addresses the delay between sensing the brake pedal position (*T11*) and actuating the brake lights (*BLAct*). The second one concerns the emergency brake feature and addresses the delay between sensing the brake pedal position (*T11*) and actuating rear indicator lights according to an emergency brake (*RLAct* and *RRAct*).

The analysis results and a comparison of the analysis performance regarding analysis time and memory consumption are summarized in the following table. As expected, an analysis based on DBMs generates less symbolic states than the variant based on time-darts. The runtime of the analysis also improved. However, as discussed before, the analysis based on DBMs may over-approximate the behaviour. The analysis based on future-DBMs is exact though and also performs better than the DBM-based analysis, as the numbers in the table indicate (in terms of runtime of the analysis, as well as its memory consumption). For the sake of completeness, we also analysed the system using only an analytical approach. Of course, this analysis is the fastest one and has the lowest memory consumption. This must be trade-off against the analysis results, which are over-approximate. The compositional analysis combining a computational with an analytical method shows promising results. The sizes of the state space and the memory consumption are several orders of magnitude lower than the holistic approach purely based on model-checking using time-darts, DBMs or future-DBMs. So at least for the case study, [KPI1.2], which expects a performance (run-time) increase of analysis tools by 50%, is fulfilled.

	Time-Darts	DBMs	Future DBMs	Analytical	DBMs + Analytical	Future DBMs + Analytical
EffectChain1	[5.25,15.25]	[5.25,15.25]	[5.25,15.25]	[5.25,16.25]	[5.25,15.25]	[5.25,15.25]
EffectChain2	[15.25,45.25]	[15.25,45.25]	[15.25,45.25]	[11.25,54.5]	[15.25,45.25]	[15.25,45.25]
System states	10,244,412	3,544,602	2,689,774	0	16,992	12,765
Explored states	133,188,232	35,053,532	11,147,689	0	1,661	1,038
Run-time	60s	17s	14s	<1s	<1s	<1s
Memory consumption	12,837 MiB	7,116 MiB	5,466MiB	11 MiB	30 MiB	24MiB

Future work after ASSUME consists of extending the interface to concurrency defect analyses to also benefit there from the decomposition of the analysis problem and the combination of computational and analytical methods. We have also successfully applied the analysis on a realistic case-study, the use-case THA_UC01, which is a model of a flight management system provided by THALES.

3. Static Analysis of C Code for Concurrency Errors

Several tools have been proposed for the static analysis of C code for concurrency errors. These are Gropius by University Kiel (Section 3.1), Goblint by Technical University Munich (Section 3.2), and Astrée by ENS, Sorbonne University (the former UPMC) and AbsInt (Section 3.3). Advantages of static analysis are that the program is analysed without actually executing it and that the results of static analysis cover all program runs with all possible inputs.

3.1. Gropius

Gropius is a static analysis tool focused on concurrency errors arising in software in the automotive domain. Gropius was in an early stage of development at the start of ASSUME and since then it matured into a reliable analysis software.

Identifying data races in the source code of embedded systems is a challenging task, especially in the automotive domain with legacy code developed for single-core processors. There are several reasons for this. Firstly, there is no ubiquitous standard on how embedded software should be structured. Thus, it is not always the case that there is a commonly used standard API for managing threads, locking resources, enabling and disabling interrupts. Secondly, more often than not, definition and treatment of critical sections is done implicitly (e.g. via priority scheduling) and therefore critical sections are very hard to automatically identify reliably. Documentation about critical sections and how mutual exclusion is implemented is incomplete or even non-existent. Lastly, there are cases of deliberately tolerated data races, where performance considerations outweigh the occasional violation of the involved critical sections.

Because specifications are incomplete the search for unwanted data races cannot be fully automated. Instead, the search should be modelled as an iterative process where the analyst is required to add additional knowledge about the system under consideration to filter out false positive results. To make this approach feasible, there are some requirements which led to the design of Gropius.

Transparency. Having trust in the tool is very important for its adoption. For each discovered data race candidate, there must be a detailed and comprehensible explanation of the tool's inference process that led to it. Only then the analyst is able to discern whether the data race is real or not.

Reduction of the number of false alarms. If a data race candidate is not real, the analyst can derive a sufficiently general rule based on the tool's inference process to reject this and other similar data race candidates. The influence of that new rule must be communicated to its creator immediately, so that the desired effect can be easily verified.

Speed. Gropius needs to be able to handle big projects with at least 250,000 lines of code within a reasonable amount of time. It starts with running analyses that have low complexity, but result in big over-approximations. It then gradually reduces the list of data race candidates by running algorithms with higher complexity.

Ranking. There must be some mechanism to sort data race candidates based on some metric, such that the engineer knows where to look at first. After the initial run of an analysis project, the user of the tool is typically confronted with several thousands of candidates. It would be too time consuming to wade through them all. Instead, with some ranking heuristics in place, the engineer can make a decision based on the top 10 or so candidates about which rule to add next.

Modularity. This requirement is based on the need of transparency explained before and the desired capability of defining a wide range of rules that strengthen or weaken aspects of previous analysis results. The output of analyses must be fine-grained enough, such that it is not only comprehensible to the user, but also provides hooks for the addition of new rules.

3.1.1. Input

Gropius needs the LLVM bitcode (currently version 3.9) of the program and a set of task entry points. The input program may need some tweaking to make it analysable. Synchronisation API function definitions should be replaced by stubs which make use of special functions provided by Gropius.

The function names below have special meanings for Gropius. They can be used to create stub implementations for the synchronization API used in the program.

- `Handle gropius_spawn_thread(Fn entry_fn_ptr, …args)`

  Create a new thread with `entry_fn_ptr` as its entry function. If `entry_fn_ptr` takes arguments then these can be supplied via `args`. Return a handle to the created task.
- `void gropius_join_thread(Handle h)`

  Mark the thread that is associated with the handle `h` as completed.
- `void gropius_lock(void *ptr)`

  Declare the memory location pointed to by `ptr` to act as a guard. The guard is set to "active".
- `void gropius_unlock(void *ptr)`

  Declare the memory location pointed to by `ptr` to act as a guard. The guard is set to "inactive".
- `Handle gropius_get_thread_id()`

  Returns the handle of the thread which calls this function.

3.1.2. Output

The analysis results are provided via an SQLITE database, text or an HTML file containing:

- List of access records to shared memory. An access record contains a task reference, the position in the call graph, accessed memory location, if it is a read or write access and a program location.
- Data dependency graph where vertices are access records and edges denote data dependencies between accesses.
- List of potential data races.
- Discovered threads and their call graphs.

3.1.3. Method description

The analysis in Gropius is realized via several sub-analysis steps. Initial project setup and the results of each analysis step are stored in a project database, such that each analysis does not depend on volatile information. This encourages an iterative and exploratory approach to analyse a software project.

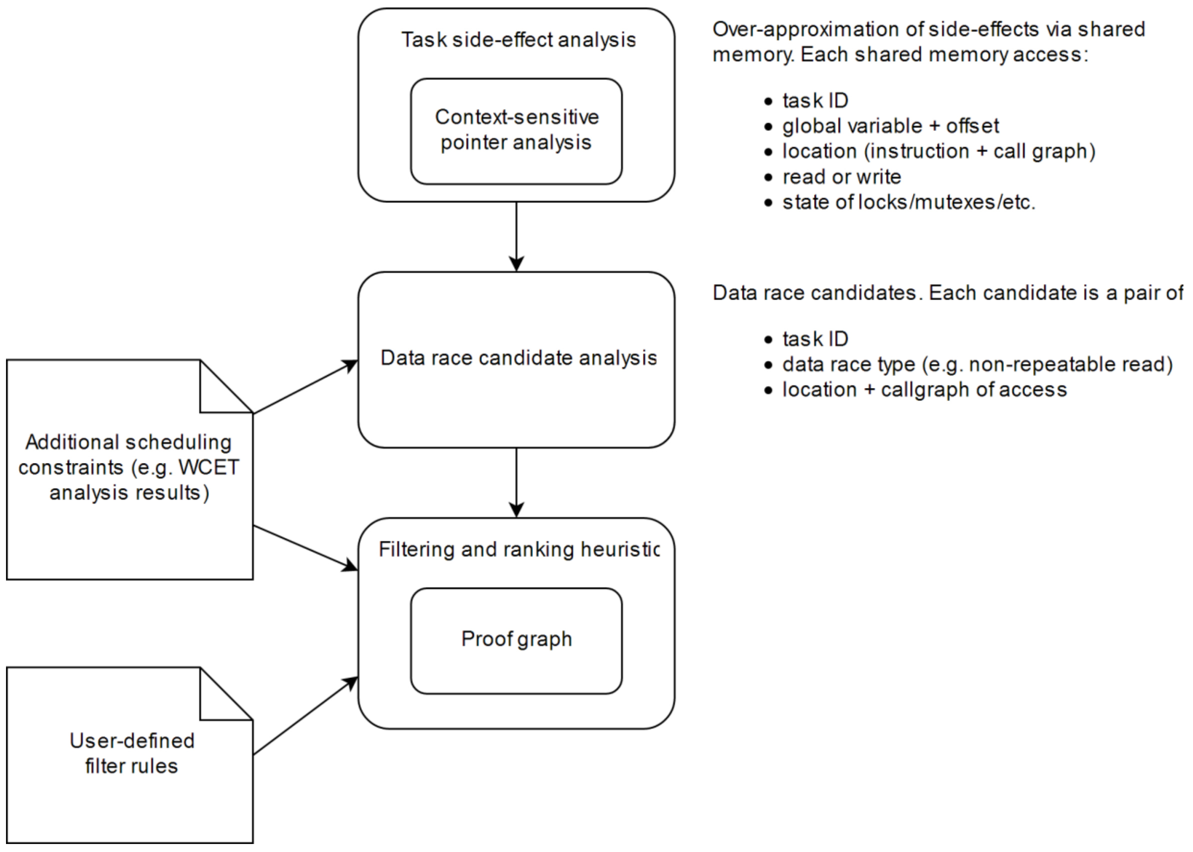

Task side-effect analysis
Context-sensitive pointer analysis
Over-approximation of side-effects via shared memory. Each shared memory access:
• task ID
• global variable + offset
• location (instruction + call graph)
• read or write
• state of locks/mutexes/etc.
Data race candidate analysis
Data race candidates. Each candidate is a pair of
• task ID
• data race type (e.g. non-repeatable read)
• location + callgraph of access
Additional scheduling constraints (e.g. WCET analysis results)
Filtering and ranking heuristic
Proof graph
User-defined filter rules

Figure 22: Analysis steps in Gropius

The main analysis steps are depicted in Figure 22. After initialization of the project with the LLVM bitcode and a set of task definitions, the *task side-effect analysis* creates an over-approximation of the tasks' accesses to shared memory. An access to shared memory consists of the memory location (global variable or dynamically allocated memory and an offset), the type of access (read or write) and the context (task, call stack, responsible instruction, state of locks, and so on).

The task side-effect analysis is based on abstract interpretation to over-approximate all reachable program states. As a consequence, false positives cannot be completely avoided, but the algorithm detects all potential data races including those which would stay hidden during testing. A sophisticated memory model keeps track of the abstract state of global variables, stack and heap memory allocations. The analysis is built upon a fast context-sensitive pointer analysis in a dataflow framework. During the analysis, pointers stored in variables and memory are tracked precisely while other data types like integers are over-approximated very coarsely, so that the analysis scales to large code bases.

The *data race candidate analysis* uses the set of shared memory accesses from the task side-effect analysis and groups them together to find data race candidates based on pairs of accesses to the same memory location. Some candidates can already be refuted at this stage by taking the locking context of the accesses into account or by using scheduling constraints provided by other tools. These are filtered out before the next analysis stage.

The data race candidates are then filtered and ranked by the *filtering and ranking heuristic* step. Not every data race is equally harmful (some are even tolerated) and a substantial percentage of the found data race candidates in the last step consists of false positives due to over-approximation and lack of information about the project. The race candidates thus need to be sorted to present the user the ones that require the most attention.

3.1.4. Ranking heuristic

All information that arises in the analysis is organized in an acyclic graph called proof graph, where nodes are proof items and edges describe the logical dependencies. A proof item can denote a function, a global variable, one edge of an approximated call graph, a data race, and so on. In short, it can be any named entity or some statement that could be later referenced in order to build a chain of reasoning. Furthermore, every proof item v has a weight $w(v) \in [0; 1]$, which denotes a probability of how “true” the proof item is.

Initial proof items of the proof graph are derived by the supplied information for Gropius. Proof items are generated for

- each function and global variable in the input LLVM module. Their initial weight is set to 1.
- each task definition. A task definition is linked to the proof item of its entry function and contains additional information like priority, CPU core, interrupt handler flag.

The following analyses are implemented in Gropius.

Call graph analysis. This analysis derives a call graph for each task proof item.

Variable access analysis. An over-approximation from where global variables may be referenced in functions. The analysis creates a proof item for each read and/or write access that may occur by running an abstract interpretation algorithm. The results of this analysis are context-sensitive, which means that a variable access proof item is linked to a certain task definition.

Data race analysis. Combines call graph information, task definitions and variable accesses and generates an initial set of proof items denoting data race candidates. Each data race candidate is linked to proof items $t_1, a_1, r_1, t_2, a_2, r_2$, where t_1 and t_2 are the tasks that may run in parallel, a_1 is a variable access within t_1 and a_2 is a variable access within t_2. At least one a_i must be a writing access. r_1 and r_2 are reachability proof items that denote the claim that there exists a feasible path in the call graph such that a_1 is reachable by t_1 and a_2 is reachable by t_2.

After the coarse-grained data race analysis, there are typically many data race candidates which are false positives. There are several ways to reduce the list of candidates with further analyses.

Manual filtering of proof items. This is always possible and the most direct way for the analyst to influence the output of Gropius. There are several use cases for this approach. For example:

- Sometimes there are global variables that are susceptible to data races, but are unimportant with respect to the system’s intended behaviour. For instance, counting variables as a simple means for reporting statistics fit into this category. Changing the weight of such a variable to 0 invalidates all proof items derived from them including data race candidates.
- Removing edges from the call graph. The call graph construction algorithm is very fast, but imprecise. There are cases where a function f is not reachable from task t because of subtle details in the control flow. By changing the weight of a call graph edge proof item to 0, the analyst can counter the effect of the over-approximated call graph.
- Some parts of the code should be excluded from data race analysis like the logging framework or the error reporting mechanism in order to focus on finding relevant data races in the actual application part of the software. By setting the weight of a function f to 0, the analyst can filter out data races that are in some way related to f.

Note, that by setting a proof item’s weight to 0, the cumulative weight of each proof item that is a direct or transitive dependency via regular dependency edges is also 0. However, it is also possible to choose a value between 0 and 1 such that in the end, the data race candidates are weighted differently and can be sorted accordingly.

Project specific knowledge. This is another instance, where choosing weights between 0 and 1 can be meaningful. Consider the following scenario in a program with a global variable G, which is a C-struct with two members x and y, and two tasks t_1 and t_2. Task t_1 writes to $G.x$ and $G.y$, whereas task t_2 reads from $G.x$ and $G.y$. Should Gropius mark this as a race condition because $G.x$ and $G.y$ may not be updated atomically? Only the analyst has domain specific knowledge about the underlying software and can submit rules that pattern match on some proof items and change their weights.

3.1.5. What changed between D5.0 and D5.1

Most of the work in the second year of ASSUME was spent on improving the accuracy of the task side-effect analysis. The initial concept of having a very fast analysis and then filter the results using domain specific knowledge and input from tools of other analysis domains is maintained. However, it is very beneficial to reduce the false positives of initial data race candidates. For that the pointer analysis domain has been rewritten. A precise pointer analysis is essential for resolving function pointers and accesses to shared memory. The analysis must be context-sensitive with respect to the analysed task and its call stack. Otherwise the over-approximation is too imprecise to be useful for data race analysis. The new pointer analysis is fast and can handle projects with 500k LOC within 30 minutes.

3.1.6. What changed since D5.1

In the third year of ASSUME, there have been several major improvements in Gropius in various areas. Concurrent programs with locks and a non-static number of threads are supported by the analysis. Gropius provides a set of special C functions which can be used as basic blocks to implement stubs for APIs for threads and synchronisation. This allows a more precise analysis of programs which consist of an initialization phase and a regular task scheduling phase. Effects of Inter-Task communication are now handled. This closes a big gap in the over-approximation. The analysis speed was improved and a further speed up was gained by making the analysis concurrent such that it makes better use of the available cores on the host machine. This reduced the analysis time of projects that took 30 minutes to compute (at the time of D5.1) to around 6 minutes.

3.1.7. Use-Cases

Gropius was evaluated in the use-cases DAI_UC1 and FORD OTOSAN.

The use-case DAI_UC1 was the main driver behind the direction of Gropius's development. Since the beginning of the project, a sound detection of shared data and code in concurrent applications [Req_DAI_01] has been the main goal. For concurrent embedded software with hard deadlines as specified by the use-cases this requirement is met.

In the use-case FORD OTOSAN, Gropius was used for the analysis of the Advanced Emergency Braking System (AEBS). The software is used mainly in collaboration with WP2, but a concurrent version based on Pthread was provided for WP5. The Pthread API for thread creation and mutexes was successfully stubbed by using Gropius's native thread and synchronisation API.

3.1.8. KPI Status

[KPI1.2] At project start Gropius was not able to completely analyse a real-world software project provided by Daimler. Some parts of the code were difficult to analyse (for instance a single C function with over 5000 LoC and nested loops) and induced a time-out or memory-out. The current version of Gropius can handle this kind of projects.

[KPI1.3] Pointer analysis was very weak at the beginning. A large part of the generated data race candidates could be attributed to that. Compared to the current version of Gropius, the number of data race candidates dropped significantly.

[KPI2.1] Data race analysis so far has been focused on finding *dirty read* and *lost update* errors. Other data race classes like *non-repeatable read* are detected as well, but not yet classified as such. This is still work in progress.

3.2. Goblint

The Goblint analyser is developed at Technische Universität München (TUM) and University of Tartu (UT). Goblint is a static analysis framework that supports static analyses where some parts of the program state may be analysed flow-insensitively while others may be analysed both flow- and context-sensitively. This flexibility allows for analysing multi-threaded programs in a modular way. The focus of Goblint so far has been on analysing the locking behaviour of multi-threaded C programs. Goblint also supports potentially failing lock operations and is able to perform path splitting in order to track different behaviour of the program depending on the outcome of locking. Goblint is written in OCaml and makes use of the CIL framework [12] as the front end. A high-level overview of Goblint is given in Figure 23. First, the control flow graph is constructed from the standard textual representation. From the control flow graph and a given analysis specification, a side-effecting constraint system is generated. Next, the selected generic solver solves the constraint system. All of the presented high-level components are implemented in Goblint as modules or functors in OCaml.

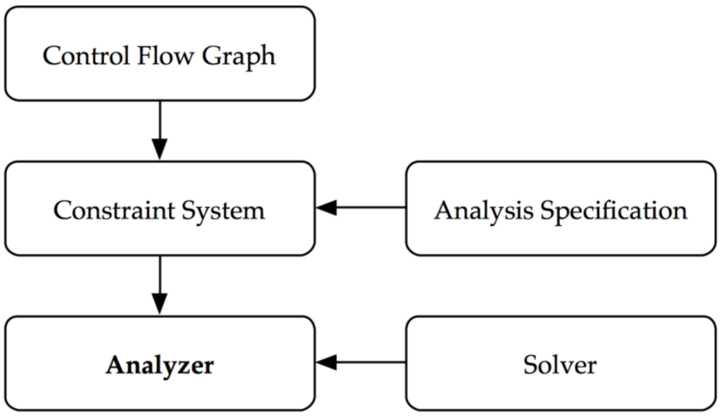
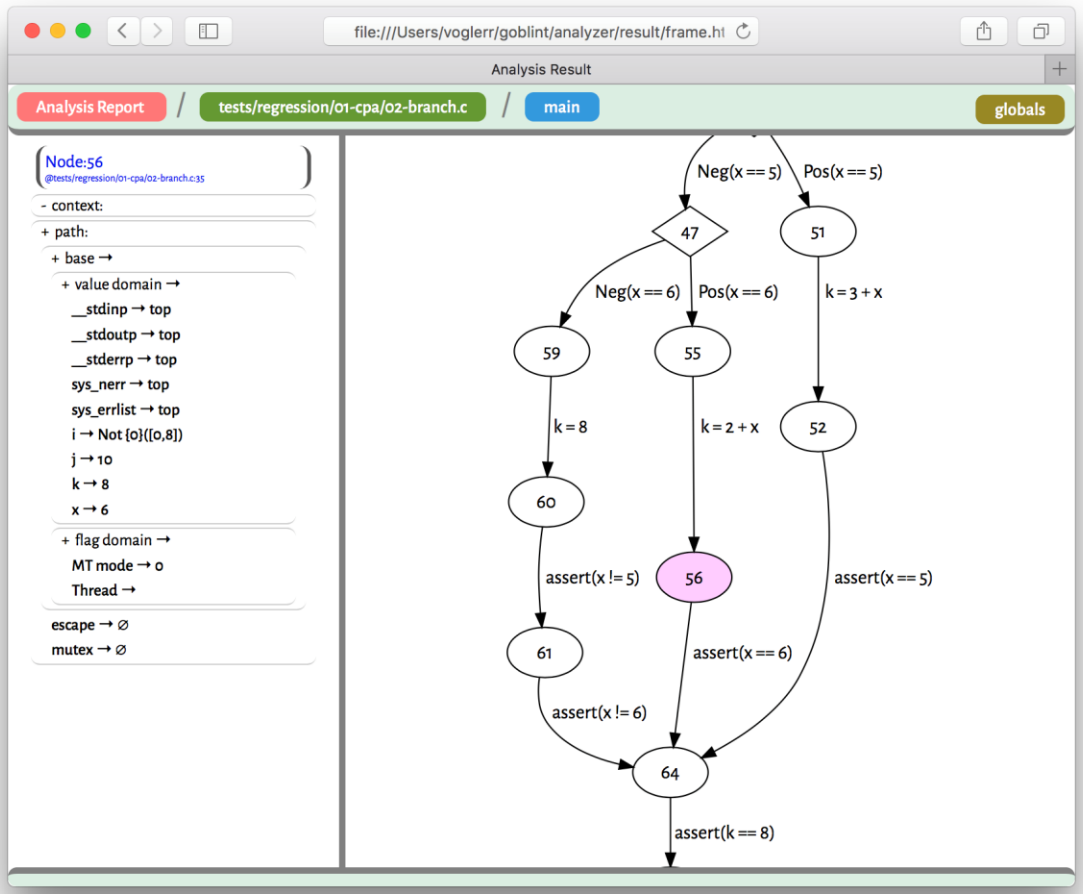


Figure 23: Structure of Goblint (see [13])

3.2.1. Input and Output

Goblint takes C source files and, if needed, an include directory. Then Goblint runs the C pre-processor to produce a single file, which is passed to the front end CIL. After termination of the analysis, the analysis results can be written to a file in several formats. For the JSON format, the results can be saved to a file, or be persistently stored in the database MongoDB for later inspection and post-processing. The default output format is XML, which can be transformed to an HTML representation for inspection by the user. The HTML output includes the highlighted source code with analysis information for every code line. Alternatively, a control-flow graph representation can be displayed (see Figure 24). In both views, dead code as well as other warnings such as violated assert statements or data races are visualized.



```
Node:56
@tests/regression/01-cpa/02-branch.c:35
- context:
+ path:
  + base →
    + value domain →
      __stdinp → top
      __stdoutp → top
      __stderrp → top
      sys_nerr → top
      sys_errlist → top
      i → Not {0}([0,8])
      j → 10
      k → 8
      x → 6
    + flag domain →
      MT mode → 0
      Thread →
  escape → ∅
  mutex → ∅
```

Figure 24: Goblint's HTML output

3.2.2. Configuration

Goblint is highly configurable, as most of the analyses implemented in Goblint are parameterized in some way. Different analysis parameters are used, for example, to enable experimental features, or to adjust the precision in order to increase the speed of the analysis. In some cases, an analysis requires extra information about the code to be analysed, e.g., is it a Linux device driver, an OSEK program, or a program using the POSIX interface. An overview over all available options can be obtained by running `goblint -print_all_options`. To allow flexible configuration of the analyser, a JSON (JavaScript Object Notation) based sub-framework is available. The framework simplifies the task of adding new options and querying the value of the options inside the analyser. All options and values are stored centrally to allow for better overview of the settings. Additionally, the system allows grouping of settings, merging of settings from different sources, and writing the active settings back to a file. At start-up, the default configuration is active, however, there are several ways to change the settings – before they are accessed. One way is to merge an external JSON file with the currently active configuration. For that, the name and path of the external file can be passed to Goblint via the `--conf` command line option.

3.2.3. Method description

Goblint is based on a clear distinction between the specification of the analysis and the solver engine. This simplifies arguments about the correctness of each component. The analysis designer provides abstract domains and specifies the local abstract behaviour of statements and conditions. Together with the control-flow graph as provided by the front end, a side-effecting constraint system is generated. The analysis result then is obtained as a solution to the given constraint system by one of the solver engines. In some cases, a further post-processing phase may be required to identify potential bad behaviour such as data races. For those cases, Goblint searches for possibly concurrent accesses to global data.

3.2.4. Analysis of Asynchronous Programs

The analysis of concurrent tasks or function calls in Goblint is simplified if possibly shared data are analysed flow-insensitively. In this case, each task can be analysed individually with respect to a global invariant for the shared data. This approach has been realized for C with POSIX threads, for OSEK/AUTOSAR as well as for ARINC653. As synchronization primitives, this approach nicely supports locking and unlocking as well as dynamic changes in thread priorities as provided, e.g., by OSEK. ARINC programs may also make use of setting of and waiting for events. Further functionalities, which need to be supported, are various forms of wait and blocking operations on blackboards or buffers. Moreover, dynamic changes of the task status via suspend, resume, stop, start must be handled, as well as changes of the partition mode. In order to deal with all these and soundly analyse the potential violation of liveness properties such as starvation of tasks or deadlocks, we have realized a two-stage approach. First, Goblint is used to extract a decently small model, which (under reasonable assumptions) over-approximates all possible concurrent behaviours of the original program. This model then is fed into a standard model-checker, which simulates all possible executions on the ARINC platform and is able to verify LTL formulas.

3.2.5. Scalability

In close contact with Daimler (use case DAI_UC1), TUM has tried to accommodate Goblint to the analysis of large concurrent OSEK applications.

The space consumption during the analysis is a severe obstacle to the scalability of Goblint to larger programs. In order to deal with that, several new options have been provided:

- It is now possible to iterate over the control-flow graph with basic blocks as nodes, rather than individual statements. This saves on average 30-40% of constraint variables, without making compromises concerning the run-time of the analysis.
- Widening on contexts has been introduced to reduce the number of calling contexts distinguished by the analyser. This allows dealing with programs that may contain recursive functions – even if they use integer arguments.
- Finally, an experimental generic solver has been constructed that only keeps values at widening points and reconstructs values at other nodes only after the fix-point has been reached. This cuts down the number of constraint variables during the fix-point iteration by a factor of ~10.

3.2.6. Improvements since D5.0 and KPI Status

[KPI1.2] The newly implemented space-efficient solver has a 10x improvement in the number of variables. The new version is also guaranteed to always terminate. It terminates for all programs of our benchmark suite, while the previous version did not terminate for two of 26 tests.
The new implementation of our address domain now uses hashing to find comparable elements, which reduces the costs for operations on large address sets.

[KPI1.3] The new implementation of our address domain now keeps apart different offsets for structs, which leads to more precise results / fewer false positives.
During the evaluation of Goblint on Ford Otosan's AEBS use case, we improved the precision of the deadlock analysis.

3.3. Astrée

Astrée is a static program analyser that has been developed by ENS and licensed by AbsInt for industrialization and also addition of new features in cooperation with ENS and Sorbonne University (the former UPMC). Astrée finds runtime errors and invalid concurrent behaviour in safety-critical embedded applications written or generated in C. This is done by static program analysis by means of abstract interpretation. The analysis covers all possible program runs, including all possible inputs and all possible thread interleavings allowed by the scheduler, without actually executing the program. Astrée is sound: if no errors of a certain class are signalled, the total absence of errors from this class has been established. Moreover, Astrée aims at efficiency, targeting C code with millions of lines and dozens of concurrent processes. This is achieved through efficient abstractions and a thread-modular analysis scheme.

In WP5, the emphasis is on finding potential data races, deadlocks, inconsistent lock/unlock operations, and further invalid calls to OS services. The project also made progress on the analysis of AUTOSAR applications, on the support of dynamic priorities, including the priority-ceiling protocol, and on more precisely exploiting scheduling policies to remove spurious false alarms.

3.3.1. Input and Output

Astrée offers an interactive mode, with a graphical user interface, and a batch mode that works without user interaction under the control of a project file.

The main input of Astrée is C source code, either original or already pre-processed. Secondary input is given by directives that provide additional information about the analysed program and its environment. Directives may be inserted into the C source code or given in separate annotation files. An annotation consists of a directive and a description of the program point to which the annotation applies. Directives can be provided by users, but Astrée also offers interfaces to certain model-based code generators. These interfaces automatically convert relevant model information into Astrée annotations.

To better support the analysis of applications running under the OSEK/AUTOSAR operating system, Astrée was extended with an OIL converter that extracts all information specified in an `.oil` configuration file and automatically generates the corresponding C data structures and access functions. This allows a direct analysis of OSEK/AUTOSAR applications.

Astrée outputs a list of potential and definite run-time errors, with source location and context information (such as call stack, offending variable values, etc.). Astrée also gives information about variable ranges, pointers to data and code, shared data-structures, the call graph, and dead code. This information can be navigated interactively in the source code interface. Astrée also produces a detailed textual report file for human inspection and an XML report file that may be read by other applications.

3.3.2. Features of Asynchronous Programs

Besides being able to cope with synchronous programs that have to deal with asynchronous events, Astrée can perform a precise and sound analysis of asynchronous programs, i.e., concurrently executing tasks. To avoid the cost of analysing generic operating system functions, these can be replaced by so-called stubs, i.e., small functions in which the operating-system primitives are replaced with Astrée intrinsics that tell Astrée about the effect of the primitives without an extra analysis. Stubs are also useful to model parts of the programs or the operating-system written in assembly code. Astrée is provided with ready-to-use libraries of stubs to model significant parts of popular operating systems, including ARINC 653 (for avionics), OSEK/AUTOSAR (for automotive applications), and POSIX.

Internally, Astrée models concurrent executions using two classes of built-in objects: processes and synchronization objects. A process is an execution unit and conforms to a process in ARINC 653 terminology, a thread in POSIX terminology, or a task in OSEK/AUTOSAR terminology, i.e., it is an execution unit with independent control, but sharing the memory of other execution units. Synchronization objects include mutually exclusive locks (mutexes), events, and barriers.

Each object is denoted by a unique identifier that can be an arbitrary integer or pointer. Identifiers can be chosen freely by the concurrent library stubs linked with the analysed program. Each class of objects (processes, mutexes, events, barriers) has its own identifier namespace.

3.3.2.1. Processes

Astrée processes are intended to be used to model OS-level threads, tasks, and processes. They can also be used to soundly model other forms of asynchronous executions, such as POSIX signals, interrupt handlers, models of hardware controllers, models of asynchronous environments and so on.

Processes have to be registered to Astrée before being used. This is done with an Astrée intrinsic of the form `__astree_create_process (task, id, priority)` where the first parameter is the name of the entry function of the process, the second is the process identifier, and the third is the initial priority of the process.

Processes share all global variables. There is no need to declare which global variables are actually shared as this is automatically computed during the analysis. It is not possible, however, to share local variables between processes: accessing a local variable of a process from another process is reported as an error. Additionally, while Astrée supports dynamic memory allocation for sequential programs, dynamic memory allocation in concurrent processes is currently not supported and results in alarm messages.

A process can be in one of the following states:

- stopped: the process has not been started yet or it has finished its execution and has not been started again;
- waiting: the process is waiting for an external resource which can be a mutex, an event, a barrier or a non-deterministic wait;
- suspended: the process has suspended itself and is waiting to be resumed by another process;

- pre-empted: the process is currently pre-empted by the execution of another process with higher priority;
- running: the process is currently executing.

A process that is neither stopped nor waiting nor suspended is called runnable. Note that a process enters waiting state only when executing certain analyser-intrinsic functions and hence can only wait for at most one external resource at a time. A process can, however, wait on a resource and be suspended by another process in which case the process must be resumed and get its resource to start running again.

Each process has a priority which is an integer value. Processes start with the priority specified during process creation. However, their priority can change during execution, either by direct priority modification or through the priority-ceiling protocol. Additionally, a process can be created specifying two different priorities: one which is used when the process is running and one which is used when it is not running (i.e., while waiting, being suspended, or pre-empted). It is possible to have several processes with the same priority. In that case, one of several scheduling policies can be used, on a per-process basis: a real-time process can only be pre-empted by a process with strictly higher priority, while a non-real-time process can also be pre-empted at any time by a process with the same priority. The scheduling policy is specified as an optional additional argument to the `__astree_create_process` primitive, the default being real-time scheduling.

3.3.2.2. Synchronization Objects

Synchronization objects fall into three classes: mutual exclusive locks (mutexes), events, and barriers.

Mutexes can be locked and unlocked. This is done by the Astrée intrinsics `__astree_lock_mutex(id)` and `_astree_unlock_mutex(id)` where the parameter is the identifier of the mutex. A mutex locked by a process is held by this process and cannot be locked by another process. Any process attempting this is put into waiting state until the process holding the mutex unlocks the mutex. Upon unlocking a mutex, the process with highest priority waiting for the mutex gets the mutex and is put into running state. In case there are several waiting processes with equal highest priority, Astrée considers that any of these can get the mutex, thus covering all possible scheduling policies. The unlocking process is pre-empted if it has lower priority. A mutex has to be unlocked by the process locking it and cannot be unlocked by another process. Mutexes are non-recursive, i.e., locking the same mutex a second time is a no-operation. It is, however, possible for a process to lock several different mutexes. At program start, mutexes are not locked by any process.

Mutex locking never fails, i.e., if and when the execution continues after the lock operation, the mutex is assumed to be locked by the current process. Nevertheless, potentially failing mutexes as in POSIX `pthread_mutex_timedlock` can be simulated along the following lines:

```
volatile int lock_successful;
int potentially_failing_lock (int id) {
   if (lock_successful) {
      __astree_lock_mutex(id);
      return 1;
   } else {
      __astree_yield();
      return 0;
   }
}
```

Since the variable `lock_successful` is declared as volatile, the analyser always follows both possibilities: successful lock and failing to lock.

Similarly, it is possible to model recursive (i.e., counting) mutexes where a process can lock a mutex several times and must unlock it an equal number of times for other processes to lock it. Operating system stub libraries thus model high-level synchronisation primitives transparently in terms of the simple mutexes supported internally by Astrée.

Mutexes are also used in operating system stubs to model other kinds of synchronisation objects (such as ARINC 653's backboards and queues, semaphores, etc.) and to protect internal data-structures (such as process tables, blackboard contents, etc.).

Events are synchronization objects with a binary state: an event is either set or reset. At program start, all events are in the reset state. Processes can set events, reset events, as well as wait for an event to be set. Event semantics is based on ARINC 653 and also compatible with OSEK/AUTOSAR. Waiting for an event does not consume it, and it stays set until explicitly reset. Furthermore, while being set, no wait operation on the event blocks. POSIX condition variables behave differently, but can nevertheless be simulated using Astrée events.

Barriers are synchronization objects that require a certain number of processes to reach and wait at the barrier before they are all simultaneously allowed to continue. In concurrency libraries, the number of processes that need to wait at a barrier is fixed at creation time. Astrée, however, takes this number of processes as an argument each time the barrier is used. This allows a simpler, state-less and creation-free semantics, but alternate semantics, such as POSIX barriers, can be easily simulated on top of these barriers in stub functions.

3.3.3. Analysis of Asynchronous Programs

The analysis of asynchronous programs is performed in two phases: the sequential phase, or sequential initialization, and the parallel phase. The first, sequential phase uses a purely sequential semantics. This analysis phase starts at the specified entry routine (such as `main`) and exits either when reaching the end of the entry routine, when returning from it, or when the analyser-intrinsic function `__astree_exit` is called at any point. During this phase, process creation orders issued via calls of the analyser-intrinsic function `__astree_create_process` are recorded. If no such order was issued, i.e., no process has been created, the analysis finishes after this phase with a sound sequential analysis result.

If process creation orders were issued, a second, parallel analysis phase is performed. In this phase, the interleavings of executions of the processes created in the sequential initialization phase are analysed. The initial program state for the parallel phase is the state when exiting the first phase. Hence, in the first phase, any initialization procedure required by the program before process execution can be performed. During the parallel phase, it is illegal to issue process creation orders. This constraint is generally required of embedded critical applications. Astrée thus enforces a static set of parallel processes, while allowing this set to be defined programmatically in the sequential initialization phase.

In order to analyse the process interleavings in an efficient way, Astrée performs a process-modular analysis. The processes are analysed separately, in no particular order but, during process analysis, the effect of each process on the global memory is recorded and considered in the analysis of the other processes. To take into account all possible interactions, processes may be analysed multiple times, until their analysis results stabilize. A fundamental theorem of abstract interpretation states that, upon stabilisation, the effects gathered indeed take into account all possible interleavings in a sound way. This technique proves to be more efficient for large programs than explicitly enumerating all possible interleavings, which does not scale up.

Astrée's two-phase execution model is derived from the ARINC 653 execution model. In ARINC 653, the first phase ends when a specific `SET_PARTITION_MODE` call is issued. This can be modelled as a call to `__astree_exit`. Astrée's model is also appropriate for OSEK/AUTOSAR: the first phase corresponds to synthetic code automatically generated from the OIL specification that takes care of system initialization and process creation. The second phase corresponds to the actual parallel execution of the created tasks. The current two-phase model does, however, not match the full generality of POSIX threads. Technically, POSIX allows thread creation at any point during execution. Nevertheless, in case the thread creation part of a POSIX application can be isolated from the part of the program actually running asynchronously, Astrée’s current model can still be used.

3.3.4. Execution Model

When a process is started, it starts its execution at the entry routine registered at process creation. A process execution stops when the execution reaches the end of its entry routine or `__astree_exit` is called. Additionally, process execution can be stopped prematurely, be suspended and resumed, made to wait for some resource, or be pre-empted by the scheduler to execute another process. Finally, a process can be made to yield, i.e., relinquish control for a non-deterministic amount of time, allowing other processes to run; this is useful to model waiting for an external resource or for some amount of time. Several scheduling policies are available:

- A classic real-time policy, which is the default, assumes that only the process of highest priority can run at any given point, and that it can only be pre-empted by processes of strictly higher priorities. This model assumes a mono-core execution: only one process runs at a time.
- A non-real-time policy, where only the process of highest priority can run at any given point, but it can be pre-empted by processes of equal priority non-deterministically at any point.
- Mixed models, where some processes obey the real-time scheduling policy and others obey a non-real-time scheduling policy. One example is a pre-emptive environment with non-pre-emptive interrupts.
- Using a non-real-time scheduling and processes of equal priority, arbitrary pre-emption is enabled. This model is also compatible with multi-core execution, where several processes actually run in parallel. It can also be used as fall-back model when the actual scheduling policy is not explicitly supported (such as real-time multi-core scheduling) as it considers all possible interleavings, and is thus sound for all policies.

A process can leave its running state under the following circumstances:

- the process calls an analyser-intrinsic function that makes the process wait on a resource;
- the process yields, i.e., waits for a non-deterministic amount of time;
- the process suspends itself, waiting for some other process to resume it;
- the process exits its entry routine or stops itself;
- the process releases a resource on which a process with strictly higher priority is waiting;
- the process resumes or starts another process with strictly higher priority;
- the process uses non-real-time scheduling and a process with equal priority is runnable;
- the process changes its own priority or the priority of another process such that it is no longer the runnable process with highest priority;

- there is a yielding process with strictly higher priority, whose yielding period ends non-deterministically.

Some of these pre-emptions are systematic (such as trying to lock a mutex already locked by another process), while others are non-deterministic (such as being pre-empted by a yielding process). Note that, in the last case, the current process is not pre-empted as a consequence of calling some function or primitive, but as a consequence of an event outside of its control. Hence, it can be pre-empted at any point of its execution. Astrée thus assumes that any process location is a potential pre-emption point. This assumption is vital for taking into account all possible process interleavings and soundly detecting all possible errors.

Each time a running process leaves the runnable state, Astrée selects a runnable process of highest priority to become the running process. In case several such processes exist, Astrée assumes that one is chosen non-deterministically. While this is in contrast to common OS specifications, that often resort to round-robin or FIFO policies in this case, Astrée's non-deterministic semantics is guaranteed to be sound with respect to all possible policies. A similar non-deterministic choice is assumed in case a resource becomes available and several processes with equal highest priority are waiting on this resource. Astrée's non-deterministic semantics is then again guaranteed to be sound with respect to all possible policies.

Astrée supports a directive, `__astree_set_process_priority(thread,priority),` allowing a process to explicitly change dynamically its own priority or the priority of another process. Astrée also supports the priority-ceiling protocol, which allows a process to automatically change its priority as a consequence of locking or unlocking a mutex. When locking a mutex, the process can specify a priority as an additional parameter. The new priority of the process becomes the maximum of its base priority (defined initially during process creation and possibly updated with a `__astree_set_process_priority` directive) and the priority associated to every mutex it has locked. When unlocking a mutex, the priority is lowered to take into account the remaining locked mutexes only and the base priority. Note that a priority is associated to a lock operation, and can change from one lock operation to another on the same mutex, which is more general than the classic priority-ceiling protocol and allows a simpler, creation-less semantics for mutexes. The classic priority-ceiling protocol can be easily constructed on top of these primitives. This also allows modelling the resource mechanism of OSEK/AUTOSAR. Moreover, Astrée can distinguish a different base priority for a process in its running and its non-running states. This makes it possible to model the internal resource mechanism of OSEK/AUTOSAR, i.e., resources with their own priority that are assumed to be taken when the corresponding process is running.

Astrée supports processes with several instances running concurrently. Each instance of a process has its own local variables that cannot be shared between the instances. The instances may interact with each other and the instances of other processes through global variables, which are implicitly shared. Hence, multiple instances of a process behave in many ways like multiple, distinct processes. The main difference, however, is that all instances of a process share the same identifier and creation attributes, while different processes have different identifiers and may have different creation attributes. As all instances of a process have the same identifier, it is not possible to distinguish one instance from another. Such instances can, however, be efficiently created in large or even unbounded numbers. This is particularly useful to model parametrized systems, such as servers, that take as argument a number *N* of requests and create *N* instances of the same process to handle a single request each. In Astrée, several instances of a process are created by calling `__astree_create_process()` several times with the same identifier and are analysed as efficiently as a single process.

Note that Astrée currently does not track the exact number of instances: once a process with the same identifier is created at least twice, Astrée considers that there is an unbounded number of instances of this process. This behaviour may be improved in future versions of Astrée.

For selected analysis options, the analysis performed by Astrée is also sound with respect to a large class of weakly consistent memory models, including Total Store Ordering (x86), Partial Store Ordering, as well as classic compiler optimisations. More precisely, when flow-sensitive and relational inter-process abstractions are disabled (which is the default) and all processes use a common priority and non-real-time scheduling, the analysis handles soundly the analysis of multi-core applications, including in the presence of data races. When these precision options are enabled, and the analysed program is a multi-core application running under a weak memory model, the analysis is sound only if it does not report any data race.

3.3.5. Abstraction of Concrete Behaviour

Apart from not considering the exact number of process instances, Astrée performs other abstractions of the concrete behaviour. The reason is that the model of concurrent execution presented above corresponds to a precise, concrete semantics that would be very costly to compute. Hence, Astrée reverts internally to a more abstract semantics that is simpler to compute. This more abstract semantics is nonetheless guaranteed to include all the concrete behaviour implied by the model we presented above. Due to abstractions, spurious behaviours may be introduced which may result in false alarms.

The most important abstractions that may cause a loss of precision are as follows:

- Astrée does not precisely track which processes have been started, stopped, suspended, and resumed; it only remembers whether a process may have been started at most once, or more than once during program execution.
- Astrée does not always exploit priority and scheduling information as precisely as possible. It mainly uses priorities to infer mutual exclusion of portions of codes from different processes, i.e., the fact that one portion cannot be pre-empted to give control to another portion. It does not however currently infer precedence information, such as the fact one portion of code from one process is always executed before a portion of code from another process.
- Astrée assumes that each event may be in set or reset state at every point of the program execution. This is a sound, but coarse abstraction.
- Astrée assumes that an arbitrary number of processes may have reached every barrier at every point of the program execution. This is a sound, but coarse abstraction.

The abstract semantics is subject to change to match the concrete semantics more closely, and so, improve the analysis precision. In fact, during the course of the project, the handling of priorities and scheduling has been improved to discover more mutual exclusion, hence, consider less spurious flows of information between processes, thus improving the precision. The concrete semantics, i.e., the precise rules described above, are fixed. Hence, modelling of concurrency should always take into account the concrete semantics and not rely in any way on a specific abstract semantics. This information about the currently implemented abstract semantics is nevertheless relevant to help understanding the current limitations of the analysis results in terms of precision.

3.3.6. Error Reporting

The extension of Astrée to concurrent programs reports all the run-time errors reported by the sequential version of Astrée, including: non-initialisation, arithmetic overflows, invalid integer, floating-point and pointer operations, out-of-bound array accesses, memory access errors, assertion failures, dead code.

In addition, Astrée reports hazards specific to concurrency, including:

- data races: the access of the same memory location by two processes, one of the accesses being a write and the accesses being not protected by a common mutex. Astrée distinguishes in its report read/write from write/write data races. In the presence of a data race, Astrée nevertheless continues the analysis to report subsequent errors, in case the data race is considered to be possible but benign by the user.
- invalid use of concurrency primitives, including: locking a mutex twice, unlocking a mutex not locked by the current process, referencing a non-existent process or synchronisation object, etc.
- dead-locks: cycles of processes that are blocked waiting for a mutex that is locked by another process in the cycle, so that no process in the cycle can make any progress. Astrée also reports the case of partial dead-locks, where only a part of the processes in the system are blocked, as well as blocking cycles of arbitrary size.

Finally, Astrée reports information about the use of the synchronisation objects and the use of the shared memory (which variables are actually shared, and which processes access these variables, distinguishing read and write accesses).

3.3.7. Improvements during the Project and KPI Status

[KPI1.1] Reduce the effort required to set up and employ an analysis or synthesis tool.

During the project, new options have been added to facilitate the setup of analyses by achieving higher percentage of analysed code (pointer materialization) and by automatically generating absolute address directives. This improves turnaround times and supports component-level analyses.

A Jenkins plugin is now available to automatically start Astrée analyses and evaluate their results from within a continuous integration framework.

[KPI1.1] Increase the automatic coverage of environment models (e.g. AUTOSAR libraries).

A new AUTOSAR stub library for the Dcm component was added. Now in total three stub component libraries are available (NvM, Dem, Dcm).

[KPI1.2] Increase the performance (run-time) of the analysis tools.

The analysis time of certain use cases was reduced by more than 50% (one case from >10 days to 3d, another case from 26h to 2h30, a third case from 2h to 38 min).

[KPI1.3] A significant reduction of false positives.

At the end of the first project year, Astrée assumed most of the time that any process may pre-empt any other process, disregarding the relative priorities and real-time statuses of the processes. Now, Astrée takes relative priorities into account and even supports dynamic priorities, including the priority-ceiling protocol, and more precisely exploits scheduling policies. These improvements led to the removal of many spurious false alarms: The number of data race alarms has been reduced by more than 80% from 1184 to 215 on some use case.

[KPI1.3] Reduce the effort for inspecting runtime errors.

- A new data flow view for exploring data races has been created.
- Concurrent accesses to shared variables are now graphically visualized in the call tree.
- Alarm messages about data races now indicate whether the access path is volatile and atomic.

These changes made the inspection of data race alarms much more convenient and intuitive than before.

[KPI2.1] Incorporate at least three new error classes (mainly for multi-core software).

In the project, Astrée was extended by a new analysis module for detecting potential deadlocks during concurrent program execution.

4. Analysis of Hardware-Dependent Software

Within the scope of the ASSUME project, FZI is developing a methodology and an accompanying tool chain to support analyses of Hardware-dependent Software (HdS) [14]. The ever-increasing system complexity and tight integration of hardware (HW) and low-level software (SW) components makes a strong case for an analysis that not only considers the SW/HW components in isolation but also their complex interplay. A key to performing a cross-boundary co-verification is to translate the HW and SW models to a common representation and thus enabling specification of system-level properties across HW/SW boundaries. The mechanisms for invoking functionality and sharing data among software modules are relatively few and well defined whereas in contrast the means of passing information between hardware and software are built up from nonstandard primitives that may include interrupts, memory-mapped I/O, and special-purpose registers. In addition, the hardware/software interface also marks a boundary between different threads of concurrent execution. There are three types of concurrency-related HW/SW interfaces: (1) hardware concurrency; (2) software concurrency; and (3) HW/SW concurrency. HW components are concurrent in nature and can run independently from each other whereas software components can run synchronously or concurrently in multiple threads. Emphasis of the developed methodology and tool chain is put on the analyses of asynchronously interacting HW/SW components and their interface. The HW/SW concurrency describes two situations. Most of the time, software and hardware components transition asynchronously, so their states do not affect each other. On the other hand, when hardware and software interact with each other, their synchronous transition will be decided by the states of both hardware and software.

In critical embedded systems, interfaces are often modelled as "volatile" variables and the interface specification typically as constraints on these variables. Modern "intelligent" HW components go beyond simple Port I/O and thus work directly on shared-memory, perform direct memory access (DMA), all asynchronously from the main processor. System side effects, caused by embedded assembly instructions, direct access to system memory and specific I/O-registers via Memory Mapped I/O (MMIO), make it impossible to verify the SW component without considering the HW component, because incorrect programming of the HW component can have severe consequences, such as memory zones being erased.

In addition to analysing HW/SW interaction, an analytic approach for identifying deadlocks and livelocks, based on the modelled hardware properties and a white-box model of the hardware/software system, is pursued. The algorithms for deadlock/livelock identification can take advantage of additional information in the form of timing abstraction of the system's software parts to reduce false positives during static software analysis.

4.1. Analysis of Asynchronous and Concurrent HW Dependent SW

The central idea of the developed approach is to augment the input to static software analysis back end tools by auxiliary information advancing the overall software analysis by considering specific hardware platform details. The code transformation tool C-SAPP, which is in continuous development at FZI, carries out this augmentation.

The tool can be thought of as a complex pre-processor for the software code under analysis. Its major tasks include the identification of hardware specific code sections (e.g., inline assembly code, device driver function calls, etc.) and a transformation step replacing those sections with directives that can be understood by a static analysis back end (e.g., Astrée). The pre-processing step is devised as an extensible sequence of actions, implementing various source code transformations and source code injection mechanisms that rely on analysis-specific hardware models. The hardware models that describe specific hardware properties and the functional behaviour of a hardware platform (e.g., the interrupt behaviour of a processor) are incorporated into the pre-processing step in order to generate additional information for the static analysis back ends. Besides augmentation, the developed tool can also act as an analysis back end and perform static analysis tasks. Figure 25 shows the overall workflow of the C-SAPP tool.

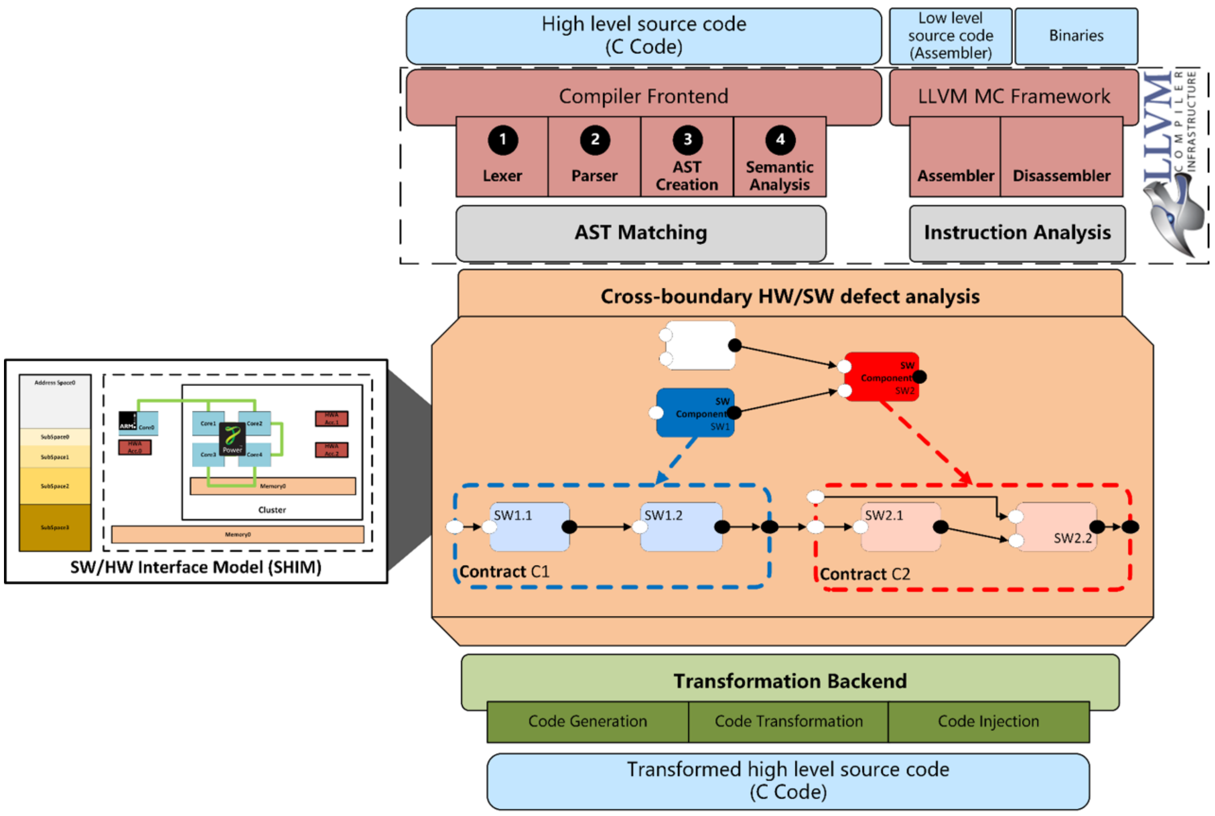


Figure 25: Analyses of Hardware-dependent Software (HdS) workflow

4.1.1. Input

Primary input of the C-SAPP tool is C source code complemented by the addition of common settings (e.g., include paths), necessary to generate an abstract syntax tree (AST) of the provided source code. In addition, the target binary (e.g., ARM `.elf` file) is needed to generate the timing behaviour of concurrent software components on task level.

Concurrent software components are specified using Quality Contracts. A contract can be used to specify the scheduling behaviour and the binding of a task to a specific computing resource (e.g., CPU). Furthermore, one can specify additional restrictions in the form of assumptions (e.g., interrupts are disabled during task execution) as well as guarantees, which hold after task execution (e.g., task execution time is limited to an execution interval $[\tau_{min}, \tau_{max}]$) using contracts.

As secondary input, the tool requires a hardware model and a set of transformation rules. The hardware model – precisely SW/HW Interface Model – specifies the functionality and constraints of a specific hardware platform that are relevant for cross-boundary HW/SW defect analysis. The transformation rules control the actual code transformation under consideration of analysis goals, analysis back end (e.g., specific keywords used by a particular back end) and the hardware model. The hardware model can be seen as a formal representation of a hardware component, where hardware behaviours are modelled using atomic transaction functions and implementation details such as clock signals are abstracted away. A hardware transaction represents a hardware state transition that is atomic to software. Any terminating C function can be treated as a transaction function. To formally specify the HW/SW interface one has to capture all possible HW/SW behaviours that are allowed by the interface. In case of MMIO, the *read()* and *write()* access to the HW device registers are modelled.

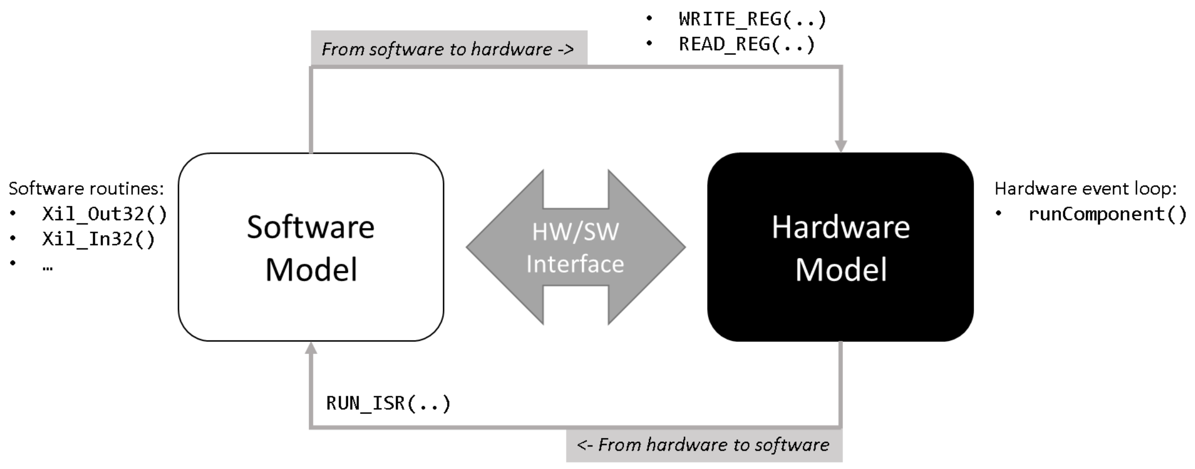


Figure 26: Specification of the HW/SW interaction

Figure 26 illustrates the specification of a HW/SW interaction. The software model is defined by the firmware routines that can access the hardware component. The hardware model describes the desired hardware behaviours when hardware works asynchronously with the software to realize system functionalities. The HW/SW interface, as the abstraction of the HW/SW stack layers between the hardware component and its software driver, propagates software events to hardware and vice versa. For example, when a driver writes to a hardware interface register, the HW/SW interface will update the related hardware registers accordingly. This can lead to a state transition in the hardware model generating side effects. In summary, the approach specifies hardware behaviour using C programs to model the state and the state transitions of the hardware component. When necessary, the parallelism between firmware and hardware is captured with a modelling approach that employs software concurrency in the form of asynchronous threads (e.g., pthreads).

The overall hardware platform is modelled using the SW/HW Interface Metamodel (see Figure 27), which allows for modelling the hierarchy and interconnection of hardware components. The SW/HW Interface Metamodel, which is implemented using the Eclipse EMF framework, is currently under development and enables the user to model a hardware platform from the software perspective. The hardware platform topology is modelled using Master/Slave components that can have a shared address space or their own subspace. Interactions between components are modelled using communication sets. In the current version of the metamodel, various communication interactions are considered, but can be easily extended in future revisions of the metamodel.

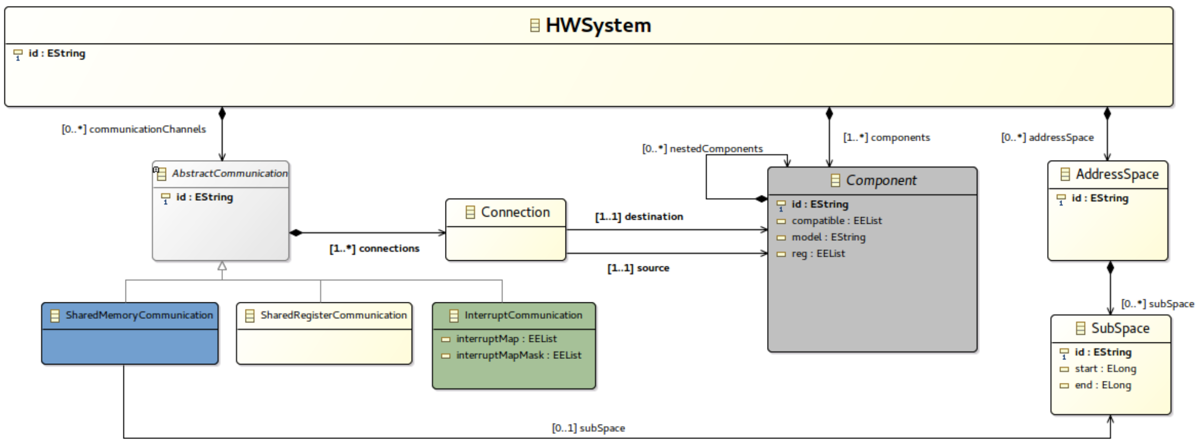


Figure 27: Excerpt from the SW/HW Interface Metamodel

A model of the whole hardware platform allows for the automatic generation of C code that represents the hardware components themselves as well as their interaction through their HW/SW interface. The HW/SW Interface Metamodel is very abstract and just covers the hierarchy of hardware components, their address space and their communication. Figure 28 displays a platform model of the FZI_UC01 demonstrator, where hardware accelerators (HWAs) interact with the software running on the ARM cores through MMIO and a common address space. This model can be used to automatically generate source code annotations and assumptions about the global memory access as well as the access to the hardware registers of the hardware accelerators through MMIO registers.

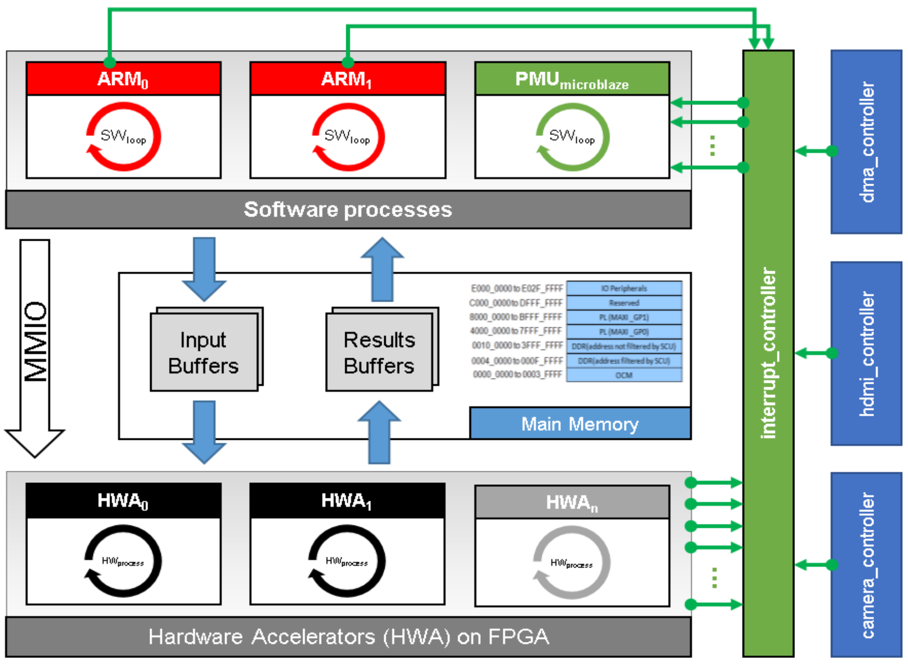


Figure 28: Platform model for the FZI_UC01 demonstrator

In addition, final and intermediate results of AbsInt's WCET analyzer aiT are used as auxiliary input. Statically computed tight bounds for the worst-case execution time (WCET) of tasks in real-time systems are extracted from the generated control flow graph that aiT produces, and exploited in an analysis that considers timing information.

4.1.2. Invocation

The C-SAPP tool is invoked on the command line in batch mode, which enables its use as a pre-processor in an automated analysis flow. Hardware modelling is done in an Eclipse environment using customized EMF modelling editors.

4.1.3. Method description

The core of the tool is based on pattern matching. Patterns will match parts of an AST and are associated with transformation rules that describe transformations of the AST that shall be performed for each match. Transformation rules may include meta-variables, which could for example refer to specific parts of a pattern, hardware model elements, analysis back end properties, or secondary analysis inputs.

Generation of C code that specifies hardware components is achieved through model-to-text (M2T) transformations using templates that are tailored to an analysis goal.

Besides code transformation, the C-SAPP tool also generates modified control flow graphs (CFGs) enriched with concurrency control primitives such as locks (e.g., mutexes) and annotated timing constraints, extracted from aiT's CFG. The generated graphs, which represent the control flow and the timing constraints between concurrency control primitives for each task, are used for deadlock analysis described in Section 4.2.

4.1.4. Output

The main output of the tool is the processed source code, in which hardware dependent code constructs will be replaced such that the code can be analysed by an analysis back end. In addition, hardware and platform specific properties (e.g., interrupt behaviour of a processor) are injected into the source code by the C-SAPP tool in form of functionally equivalent behaviour models. For example, if the correct use of processor modes (e.g., interrupts enabled/disabled) shall be investigated, the output may contain a global variable to model the current processor mode and its status flag register. C code, specifying the interaction of a hardware component with software, is generated and can be included in a static analysis of the software project.

4.1.5. Changes compared to D5.0 and future development

The modelling approach using the SW/HW Interface Metamodel has been improved by refining the metamodel and implementing a GMF based editor in Eclipse to visually support the modelling process. A new editor for specifying register interfaces using SystemRDL is being finalized. A code generator that takes the register specification and the HW/SW interface model as input and produces C code that can be used in a static analysis is under development. An Eclipse Rich Client Platform was setup as a common basis for integrating all modelling, analysis, generation and transformation steps. The integration of all sub steps into an overall workflow is currently ongoing. The methodology and tooling developed in WP5 covers the FZI_UC01 use case and addresses the requirements Req_FZI_001 and Req_FZI_002. The requirement Req_FZI_001 is fulfilled by reducing the effort to analyse hardware-dependent code segments using a model driven approach. A SW/HW interface model allows for the automatic generation of source code and source code annotations that expose hardware properties during a static analysis. Req_FZI_002 is fulfilled because communication of parallel instances (HW, SW) and their interaction can be verified using the generated code that models the hardware behaviour and an analysis back end such as Astrée or Goblint that is capable of detecting errors (data races, deadlocks, etc.) in parallel software.

4.1.6. KPI status

At the beginning of ASSUME, the C-SAPP tool and the accompanying methodology did not exist. Because the tool was developed from scratch during the ASSUME project, the KPIs do not reflect a relative improvement of the tool chain but are an approximate estimation of the overall development process.

[KPI1.1]: Our HW/SW interface model and the code generators facilitate the static analysis of low-level embedded SW. (KP1.1 reached to approx. 65%)

[KPI1.2]: Even though our tool can process large SW code bases, spanning numerous source files with millions of lines of code, only the driver code (low-level software) needs to be examined and transformed. Because our tool only examines the low-level part of the software, the code transformation, generation and annotation tasks scale independently of the overall code size (KPI1.2 reached to approx. 80%).

[KPI1.3]: Spurious warnings are reduced by exploiting domain knowledge about the HW platform and HW/SW constraints, introduced by the HW/SW interface model. Our developed deadlock analysis uses modeled HW characteristics and extracted timing constraints to reduce false positives (KPI1.3 reached to approx. 50%)

[KPI2.1]: Our analysis addresses HW-related runtime errors, which arise due to the interaction of low-level SW with HW. (KPI 2.1 reached to approx. 60%).

4.2. Deadlock analyses for real-time embedded systems

Real-time embedded systems can be differentiated from computation-intensive desktop applications by their execution of a fixed set of concurrent tasks that execute specific control and time-dependent operations periodically. The presence of multiple concurrent tasks operating on shared memory (e.g., sensor data) makes the embedded safety-critical system highly vulnerable to race conditions, which introduce non-determinism in the system and can cause violations of temporal constraints leading to system failure [45].

One solution to avoid data races is to make sure that global data is only accessed by one task at a time by introducing so-called critical sections and the concept of mutual exclusion. The principle of mutual exclusion states that resources (e.g., global variables, MMIO registers, etc.) within a critical section can only be accessed by one execution instance (thread, process, task, etc.) at any given instant of time. Lock-based resource protection and task synchronization mitigate the problem of data races by enforcing sequential access to data within critical sections but introduce a new type of problem that can cause violations of temporal constraints in form of a deadlock. A deadlock is a state in which two or more execution instances (threads, processes, tasks, etc.) mutually block each other forever, because each one is waiting for a lock owned by another one [46]. A deadlock situation can arise if and only if the four Coffman conditions hold simultaneously in a system [47]:

1. **Mutual exclusion**: Resources (e.g., global variables) can only be accessed by one execution instance at any given instant of time.
2. **Hold and wait**: An execution instance holds at least one resource and requests additional resources held by other execution instances.
3. **No pre-emption**: Resources cannot be removed from execution instances that are holding them until the resources are released by the execution instance itself.
4. **Circular wait**: A circular chain of execution instances exists, such that each execution instance holds one or more resources that are being requested by the next task in the chain.

Detecting deadlocks precisely is a difficult endeavour because they depend on intricate sequences of low-probability events and are sensitive to timing dependencies, which makes detecting such errors with classical testing techniques (e.g., unit tests) very hard [48]. Many methodologies have been devised to overcome the difficulties in deadlock detection and avoidance such as language-level approaches, where higher-level constructs for concurrency control embedded in the programming language prohibit error-prone uses [49]. Other approaches tackle the problem by dynamically tracking the set of locks that are held during program execution and performing deadlock detection checks at runtime [50]. Post-mortem techniques [51] are similar to the dynamic approach but analyse execution traces after the program execution finished. Both methods suffer from the same limitation that only errors along executed paths can be found.

Model checking [52] is a formal verification technique that can be used to find concurrency errors in programs [53] for all possible inputs and program paths, but suffers from possible state space explosion. Several completely automatic static analyses have been developed to find deadlocks in C code [48][54]. While these techniques guarantee to find all possible deadlocks, they suffer from imprecision caused by the abstraction of precise local information and thus generate spurious warnings because of overestimation.

To improve the precision of static deadlock analysis techniques a new method was developed to exclude spurious warnings by considering timing constraints.

4.2.1. Method description

In real-time safety-critical systems, deadlines for concurrent tasks are fixed because a violation of a computation deadline can lead to catastrophic consequences. Deterministic performance and response behaviour of the whole system is of uttermost importance and can imply a restricted concurrency execution model as follows:

- tasks are scheduled periodically (e.g., every 100ms) and the starting point of each task within a period is defined statically;
- the schedule as well as the assignment of tasks to execution units (e.g. CPUs) is defined statically;
- tasks cannot be interrupted by means of pre-emption;
- the number of tasks is fixed and new tasks cannot be created dynamically;
- locking primitives such as mutexes can only be released by the task that acquired the mutex or after the task terminates.

Using the assumptions of the restricted concurrency execution model, one can calculate the minimum and maximum execution time of each task in the presence of synchronization primitives such as mutexes. Using the maximum execution time and the schedule of the tasks, one can derive the maximum deadline of the task itself and verify that the overall deadline imposed by the periodicity of the schedule is not violated. If a deadlock between two or more tasks occurs, the maximum execution time will be infinite, exceeding the overall deadline and thus leading to a violation of the real-time constraints. Instead of identifying deadlocks directly, our approach can exclude possible deadlocks (false positives) between tasks based on the timing constraints of locking primitives and a static schedule.

4.2.2. Algorithm

The developed algorithm for deadlock detection, using timing constraints, utilizes an iterative approach for computing the overall maximum execution time of each task in the presence of synchronization primitives. Describing the algorithm in all its details would exceed the scope of this document, so a short introduction based on an abstract example is given.

Table 1 depicts a possible deadlock scenario with n tasks running in parallel. For the sake of convenience, we only consider three tasks $\{T_1, T_2, T_n\}$, which are displayed in the example. A deadlock can only occur between Task 1 and Task 2 (i.e., 4 Coffman conditions are met for T_1 and T_2) whereas Task n cannot influence the other tasks because it doesn't acquire locks that are used by the other tasks. Timing information is annotated as intervals containing the absolute min/max execution time to reach a synchronization primitive such as a lock/mutex. If two tasks acquire the same resource (e.g., lock L1), their timing intervals, necessary to reach the program state where they can acquire the lock, are in relation to each other. A task that can acquire the lock first changes the timing signature of other tasks that wait on the same lock. The waiting tasks can only acquire the lock after it is released by the task holding it and thus the timing intervals of the waiting tasks need to be adjusted accordingly to compensate for the time the task is waiting for a lock.

Table 1: Deadlock example

State	Task 1 (T_1)	Task 2 (T_2)	…	Task n (T_n)
1:	$[A_{L1_{min}}, A_{L1_{max}}]$	$[A_{L2_{min}}, A_{L2_{max}}]$	…	$[A_{L3_{min}}, A_{L3_{max}}]$
	acquire($\boldsymbol{L_1}$)	acquire($\boldsymbol{L_2}$)	…	acquire($\boldsymbol{L_3}$)
2:	$[A_{L2_{min}}, A_{L2_{max}}]$	$[A_{L1_{min}}, A_{L1_{max}}]$	…	$[R_{L3_{min}}, R_{L3_{max}}]$
	acquire($\boldsymbol{L_2}$)	acquire($\boldsymbol{L_1}$)	…	
3:	$[R_{L2_{min}}, R_{L2_{max}}]$	$[R_{L1_{min}}, R_{L1_{max}}]$	…	
	release($\boldsymbol{L_2}$)	release($\boldsymbol{L_1}$)	…	
4:	$[R_{L1_{min}}, R_{L1_{max}}]$	$[R_{L2_{min}}, R_{L2_{max}}]$	…	
	release($\mathbf{L_1}$)	release($\boldsymbol{L_2}$)	…	release($\boldsymbol{L_3}$)

In a first step, the timing information is extracted from a timing analysis with aiT for each task in isolation without considering the interference of other tasks. Because tasks holding the same locking primitive can interfere with each other and change their timing signature accordingly, the algorithm has to consider all possible interleavings of the time intervals for acquiring a lock. For example, if task 1 acquires lock 1 first, task 2 blocks until task 1 releases the lock thus changing the maximum time to acquire lock 1 of task 2 to: $T_2A_{L1_{max}} = \max(T_2A_{L1_{max}}, T_1R_{L1_{max}})$. Six possible interleavings between two threads and their timing intervals, necessary to acquire a lock, are possible.

In a second step, the algorithm iterates over all locks in one task and calculates all possible timing interleavings for acquiring a lock held by other tasks. If an interleaving with the same lock in another task is found, timing intervals are adjusted for the current lock and task under investigation. The adjustment of timing intervals is repeated for every lock in every task until timing intervals reach a fixed point and do not change any further. If a fixed point is reached, a deadlock cannot occur in the system. If the timing intervals do not stabilize, the maximum execution time of a task violates the overall deadline (i.e., abort criterion) implying that a deadlock is possible. The devised algorithm can only show that a possible deadlock (false positive) cannot occur under given timing constraints or reveal that a set of tasks could violate a global deadline in the presence of locking primitives.

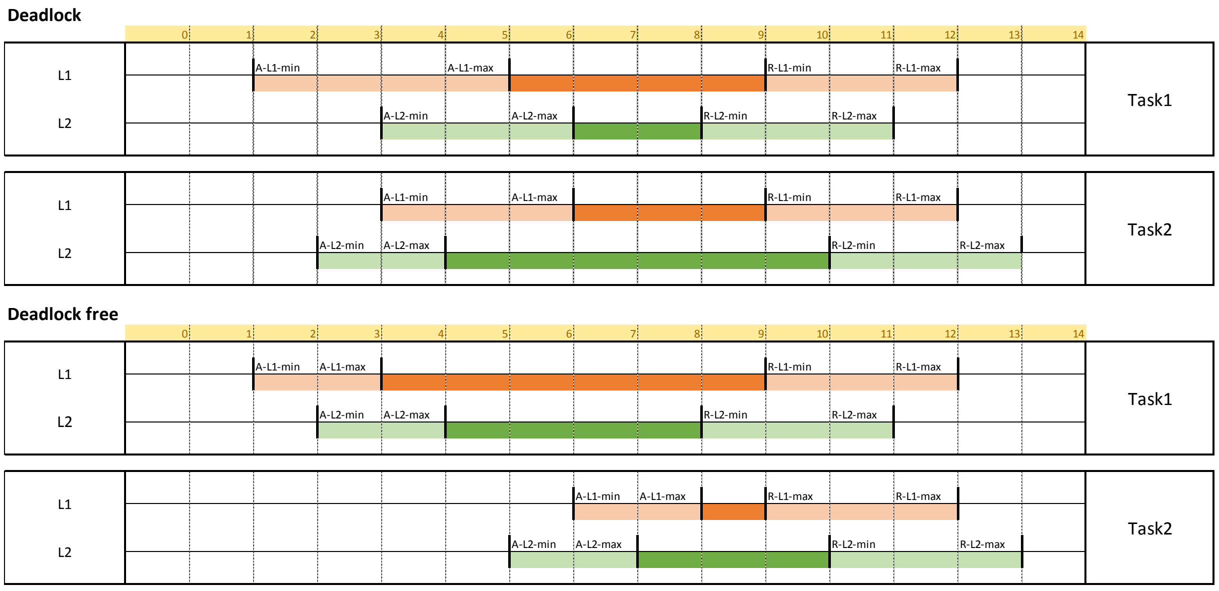


Figure 29: Two scenarios with interleavings between two tasks

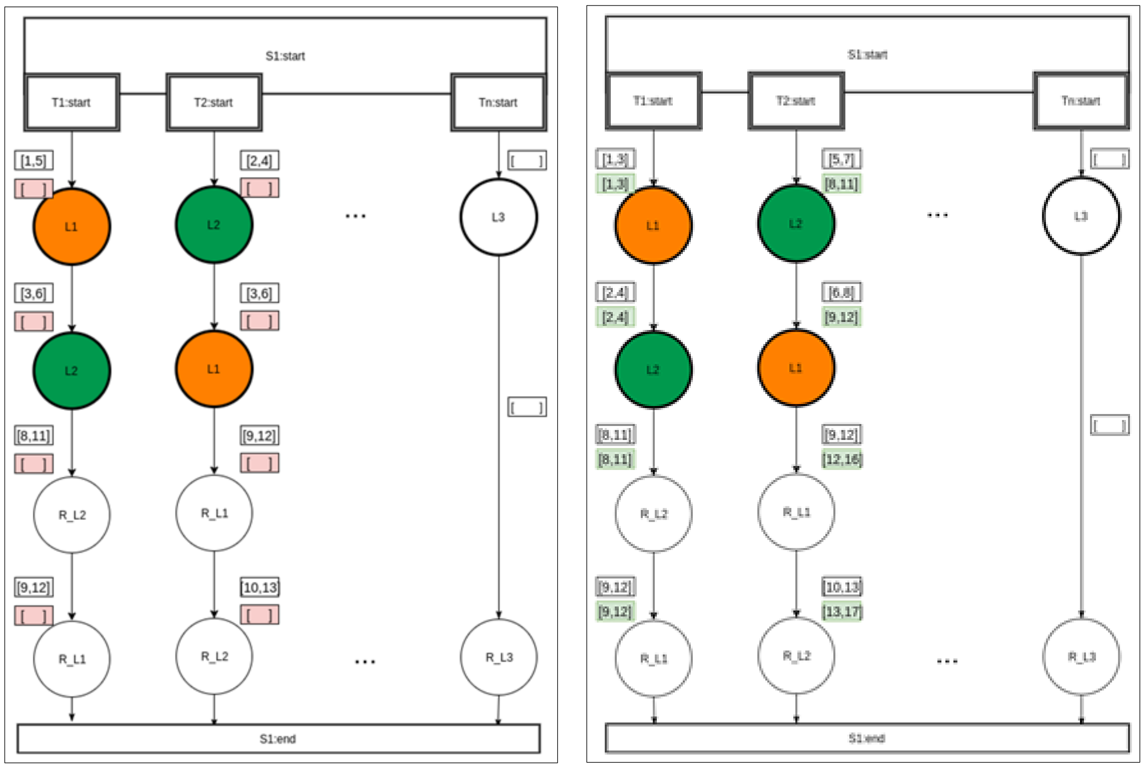


Figure 30: Calculated timing intervals for the two scenarios

Figure 29 depicts two possible scenarios and Figure 30 the calculated timing intervals for them. In the upper scenario of Figure 29, a deadlock occurs and the algorithm cannot calculate the maximum execution time for the two tasks (annotated in red boxes in Figure 30, left) because the timing intervals have a circular dependency leading to a violation of a global deadline. In the lower scenario, a deadlock can be excluded based on timing constraints. The developed algorithm can calculate the maximum execution time of each task because a fixed point, where timing intervals do not change, is found. Figure 30, right, shows the calculated timing intervals (green boxes) for this scenario.

5. Race Detection by Instrumentation

Data races in a binary program can also be detected by augmenting the program with some instrumentation code that detects and reports data races while the program is running. This approach is followed by Turkish partners within ASSUME.

5.1. EmbedSanitizer: Race Detection for ARMv7 POSIX-Thread Applications

5.1.1. Introduction

As part of the ASSUME project, *EmbedSanitizer* has been developed by Koç University, a non-profit private university in Istanbul, Turkey. The tool aims at detecting data races for Linux POSIX-thread applications developed for the 32-bit ARMv7 architecture.

EmbedSanitizer is embedded into the LLVM compiler infrastructure. Since late 2014 Clang and LLVM provide support for ARM Cortex-A17 and GCC. *EmbedSanitizer* expects C/C++ source code of a program as input. During compilation stages, it identifies all shared memory accesses and synchronization operations in the program and adds race detection callbacks to obtain a race detection mechanism. The resulting instrumented source program is compiled to an instrumented binary executable that runs natively on an ARM Cortex machine and collects information on possible data races. Thus, the data race checking is on the fly; it takes place while the program executes.

Just before the program terminates, a human readable data race report is generated that can be accessed through a command-line interface and can be redirected to a file. In particular, the tool reports where data races occur and other possible information to help the programmer fix the data race bugs.

For analysis and instrumentation, the tool supports both interactive and batch modes. It can be invoked through a special Clang compiler flag while compiling the C/C++ application for the Linux ARM Cortex architecture. Since the tool is integrated in the Clang/LLVM compiler tool chain, its mode of use is no way different from the compiler and its flags.

EmbedSanitizer is derived from *ThreadSanitizer* [67], an open-source industrial-level race detection tool. Originally developed by Google, it is now part of the LLVM compiler infrastructure (clang 3.2 and gcc 4.8), but only supports x86_64 as it relies on 64-bit address space for its internal optimizations. In contrast, *EmbedSanitizer* supports 32-bit ARM programs. It is intended to be modular; any race detection algorithm can be plugged-in and used. Possible algorithms are FastTrack (see Section 5.3) and variants of lockset-based race detection algorithms (see Section 5.2).

Lastly, *EmbedSanitizer* is developed in C/C++ and is available as open-source through the following website: https://github.com/hassansalehe/EmbedSanitizer

5.1.2. Method

EmbedSanitizer [40] is an improvement on *ThreadSanitizer* [67]. It can also be launched through Clang's compiler flag ***-fsanitize=thread***. To achieve this, we modified the LLVM/Clang compiler argument parser to support instrumentation of 32-bit ARM programs when the relevant flag is supplied at compile time. Next, *EmbedSanitizer* enhances parts of *ThreadSanitizer* to instrument the target program. Furthermore, it replaces the 64-bit race detection at runtime with a custom implementation of the efficient and precise FastTrack race detection algorithm, for 32-bit platforms. In this section, we discuss the important parts of *EmbedSanitizer* as well as its simplified installation process.

5.1.2.1. Architecture and Workflow

The workflow of *ThreadSanitizer* and the changes done for *EmbedSanitizer* are described in Figure 31. Figure 31(a) shows default and unmodified relevant components of *ThreadSanitizer* in LLVM/Clang. In Figure 31(b) these parts are modified to enable instrumentation and detection of races for 32-bit ARM applications.

At (1) in Figure 31(a), the Clang front-end reads the compiler arguments and parses them. If the target architecture is 64-bit, Clang passes the program under compilation through the *ThreadSanitizer* compiler pass for instrumentation (2). The pass then identifies all shared memory operations in the program and injects relevant race detection callbacks which are implemented in a race detection runtime library called *tsan*. Furthermore, the instrumented application and the runtime are linked together by the linker (3) to produce an instrumented executable (4). This executable runs on a target 64-bit platform and reports warnings about races in the program. We modify components in the workflow as discussed next.

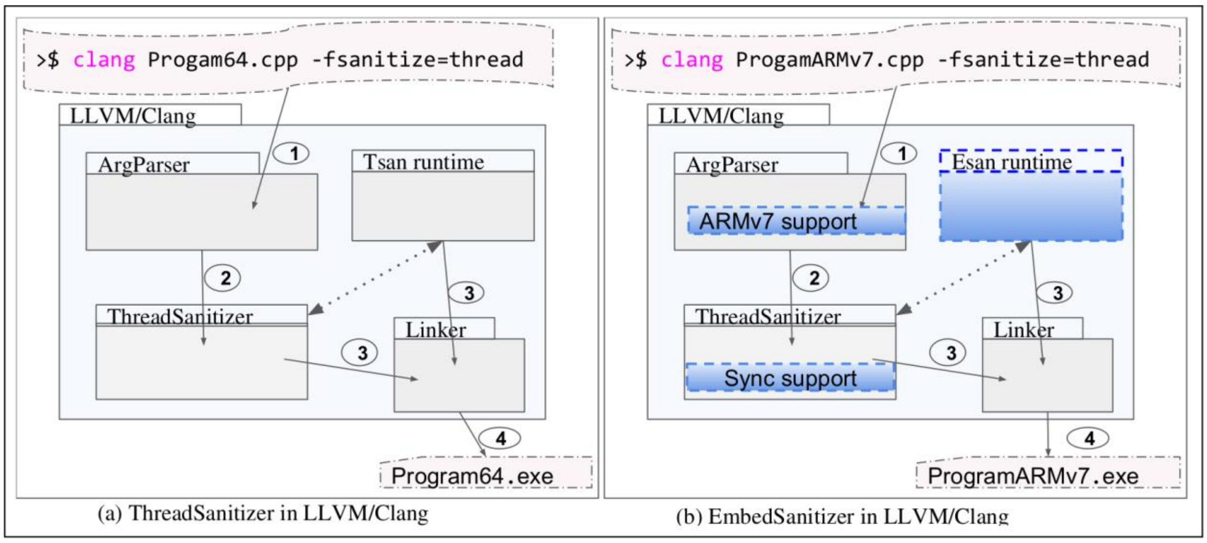


Figure 31: High level abstraction of ThreadSanitizer and EmbedSanitizer in LLVM/Clang.
(a) ThreadSanitizer: essential LLVM modules for race detection.
(b) EmbedSanitizer: same modules modified to instrument and detect races for 32-bit ARM

(a) Enabling Instrumentation of 32-bit ARM Code in LLVM/Clang:

We modify the argument parser of LLVM/Clang to support instrumentation once *EmbedSanitizer* is in place, Figure 31(b). Therefore, if ***-fsanitize=thread*** is passed while compiling a program for 32-bit ARM code, the instrumentation takes place. To do this we identified the locations where Clang processes the flag and checks the hardware before skipping the launching of the *ThreadSanitizer* instrumentation module because of the unsupported architecture.

(b) Modifying the ThreadSanitizer Instrumentation Pass:

Despite its instrumentation pass, *ThreadSanitizer* has become complex, partly due to its integration into the LLVM's compiler runtime. Therefore, we extended the available instrumentation pass to identify and instrument synchronization events and inject relevant callbacks and kept instrumentation of memory accesses as it is.

(c) Implementation of Race Detection Runtime:

The default race detection runtime in *ThreadSanitizer* uses memory shadow structures which rely on 64-bit architectural support. Due to the complicated structure of *ThreadSanitizer*, it was not possible to adopt its runtime for the 32-bit ARM platform. Therefore, we implemented a race detection runtime by applying the FastTrack race detection algorithm. The library is then compiled for 32-bit ARM and is linked to the final executable of the embedded program at compile time through the LLVM/Clang compiler infrastructure.

5.1.2.2. Installation

Figure 32 shows the building process of the LLVM compiler infrastructure with *EmbedSanitizer* support. To simplify this process, we developed an automated script with five steps. In the first step, it downloads the LLVM source code from the remote repository. Then it replaces files of the LLVM/Clang compiler argument (flags) parser with our modified code to enable *EmbedSanitizer* support for ARMv7. Third, the LLVM code is compiled using GNU tools to produce a cross-compiler which targets 32-bit ARM and supports our tool, *EmbedSanitizer*. Fourth, the race detection runtime which we implemented is compiled separately and integrated into the built cross-compiler binary. Finally, the built cross-compiler is installed which can eventually be used to compile 32-bit ARM applications with race detection support. This whole process is applied once.

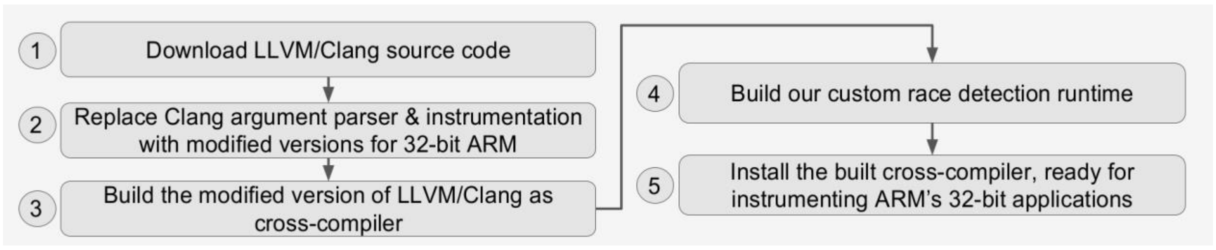

1
Download LLVM/Clang source code
2
Replace Clang argument parser & instrumentation with modified versions for 32-bit ARM
3
Build the modified version of LLVM/Clang as cross-compiler
4
Build our custom race detection runtime
5
Install the built cross-compiler, ready for instrumenting ARM's 32-bit applications

Figure 32: Showing the automated process for the initial build of EmbedSanitizer

5.1.3. Evaluation

We evaluate *EmbedSanitizer* for detecting runtime data races for 32-bit embedded ARM applications, based on two categories. First, we want to see how the precision of race detection in *EmbedSanitizer* deviates from that of *ThreadSanitizer* since *EmbedSanitizer* extends it by using its instrumentation features and implements a custom FastTrack algorithm for detecting races. Second, we want to compare the overhead of *EmbedSanitizer* when running on a target embedded device against when running on an emulator. The key motivation is to show that running race detection on a target device is better than on emulation.

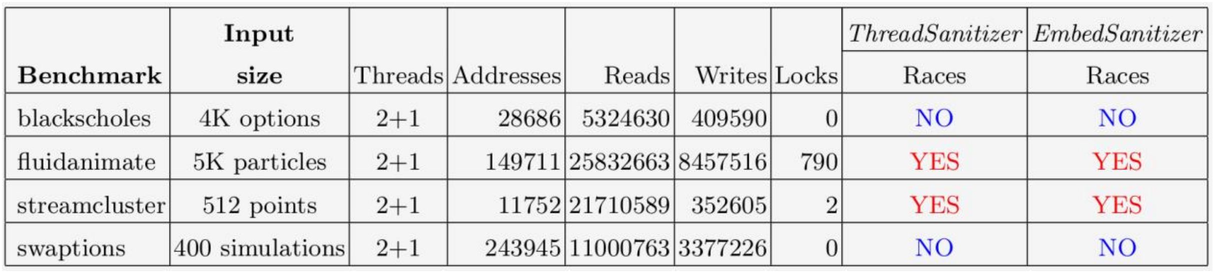
For experimental setup, we built LLVM/Clang, with *EmbedSanitizer* tool, as a cross-compiler in a development machine running Ubuntu 16.04 LTS with Intel i7 (x86 64) CPU and 8GB of RAM. As our benchmarks, we picked four (4) of the PARSEC benchmark applications. We adopted these applications to the Clang compiler and our embedded system architecture. A short summary about the applications we used for evaluation is given below.

- ***Blackscholes:*** parallelizes the calculation of pricing options of assets using the Black-Scholes differential equation.

- ***Fluidanimate:*** uses spatial partitioning to parallelize the simulation of fluid flows which are modeled by the Navier-Stokes equations using the renowned Smoothed particle hydrodynamics.
- ***Streamcluster:*** is a data-mining application which solves the k-means clustering problem.
- ***Swaptions:*** employs the Heath-Jarrow-Morton framework with Monte Carlo simulation to compute the price of a set of swaptions.

5.1.3.1. Tool Precision Evaluation

We compare the race reports detected by *EmbedSanitizer* against *ThreadSanitizer*. To do this we run the same benchmark applications with *ThreadSanitizer*, as well as with *EmbedSanitizer*. The instrumented program using *ThreadSanitizer* is run on an x86_64 machine, whereas the binary compiled through *EmbedSanitizer* is executed on ARM Cortex A17 TV. In this setting of four PARSEC benchmark applications, in an application where *ThreadSanitizer* reported races, *EmbedSanitizer* also reported them. Therefore, *EmbedSanitizer* did not sacrifice any race detection precision.



	Input						*ThreadSanitizer*	*EmbedSanitizer*
Benchmark	**size**	Threads	Addresses	Reads	Writes	Locks	Races	Races
blackscholes	4K options	2+1	28686	5324630	409590	0	NO	NO
fluidanimate	5K particles	2+1	149711	25832663	8457516	790	YES	YES
streamcluster	512 points	2+1	11752	21710589	352605	2	YES	YES
swaptions	400 simulations	2+1	243945	11000763	3377226	0	NO	NO

5.1.3.2. Tool Performance Evaluation

To compare the race detection overhead, we ran non-instrumented and instrumented versions of the benchmarks on an embedded TV with ARM-Cortex A17 CPUs of 4 logic cores and 933MB of RAM, and on a Qemu-ARM emulator running on a workstation. The slowdown is calculated as a ratio of the execution time of the instrumented program with race detection on and the execution time of the program without race detection. The number of threads was 3 because using the full set of 4 logical cores was crashing the TV. Next, the input sizes were the same in each benchmark setting. Results in Figure 33 show that detecting races in an emulator incurs between 13x and 371x slowdown whereas the slowdown in the TV is between 12x and 214x. In overall, the results in Figure 33 suggest that detecting races in a target hardware is faster than in an emulator.

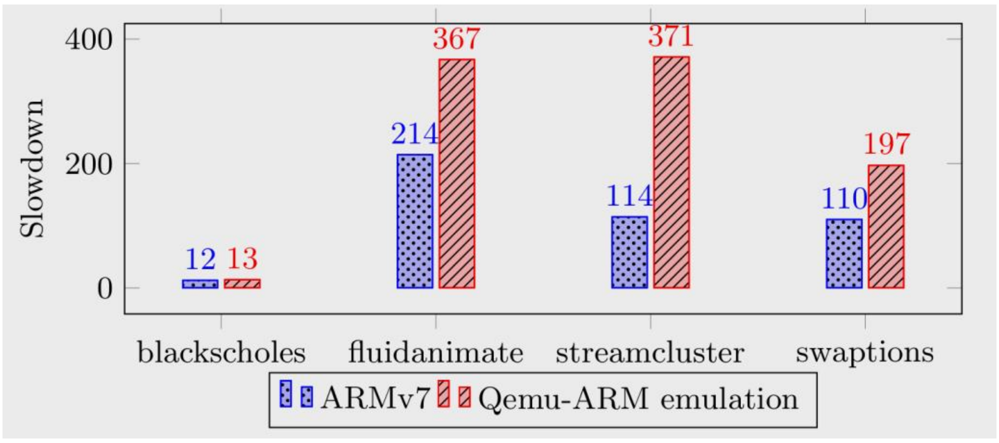


Figure 33: Slowdown comparison of race detection on ARMv7 vs on Qemu-ARM

5.1.4. Uses cases and requirements

Koç University, the developer of *EmbedSanitizer*, collaborates with Arçelik A.Ş. through the use case ARC_UC02 which is about detecting data races in Personal Video Recorder (PVR) of smart/connected TV. Figure 34 shows how *EmbedSanitizer* fits in when it detects data races for a 32-bit ARMv7 POSIX Threads smart TV software. The produced binary through instrumentation by *EmbedSanitizer* is installed into the smart TV and then it is rebooted. The new instrumented software reports race warnings while the TV is running the application software.

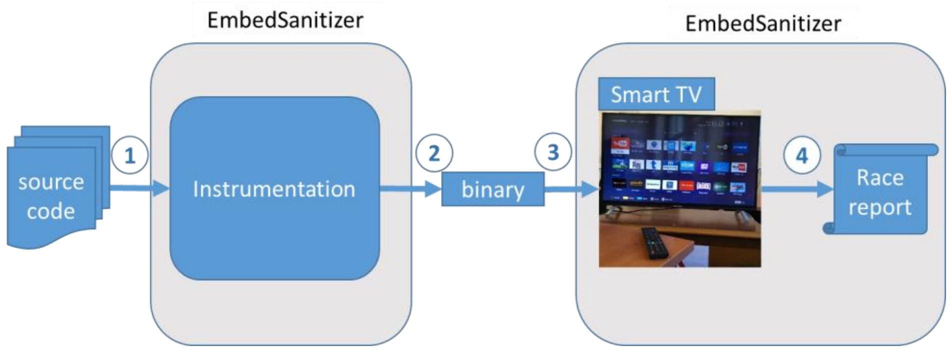


Figure 34: Alternative overview of EmbedSanitizer when used to detect data races in collaborator's 32-bit ARMv7 smart TV software for use case ARC_UC02

Two requirements have been fulfilled in use case ARC_UC02: (1) Req_ARC_03 (partial): Detection of race condition situations, and (2) Req_ARC_04 (partial): Provide detailed (race) reports

5.1.5. Status w.r.t. KPIs

The main goal ARC_UC02_EC_01 – aid in finding bugs – has been reached. Moreover, the following KPIs have been achieved:

KPI 2.1: Number and extent of error classes: *EmbedSanitizer* detects race conditions in multithreads programs in 32-bit embedded systems. Thus, it contributes to this KPI by addressing one class of errors by fully detecting data races.

KPI 2.2: Precision of reported results: *EmbedSanitizer* relies on a precise and efficient *happens-before* race detection algorithm called FastTrack.

5.2. Goldilocks

The Multicore Software Research Center of Koç University has been carrying out research on dynamic and static tools for multicore concurrent software for over a decade. The Center has developed verification tools such as the Goldilocks dynamic race detector and a series of concurrent software proof systems, culminating in CIVL (concurrent Boogie).

Goldilocks is an efficient and precise lockset-based race detection algorithm. It employs both happens-before concepts and locksets for efficiency and precision. Originally it was implemented as runtime race checking tool for concurrent Java programs in a Java Virtual Machine called Kaffe. It provides variants of lockset algorithms, can uniformly and naturally handle all synchronization idioms such as thread-local data that later becomes shared, shared data protected by different locks at different points in time, and data protected indirectly by locks on container objects. There are variants of Goldilocks implementations that provide additional capabilities, such as explicitly handling software transactions as a high-level synchronization idiom, and distinguishing between read and write accesses. To improve efficiency, Goldilocks uses techniques such as short-circuits for memory accesses and lazy evaluation for locksets. Short-circuits aim at eliminating unnecessary lockset update rules by constant-time happens-before relation checks for consecutive memory accesses of the same thread or protected by the same lock for a long time.

The input of the tool is the Java object code that will be instrumented to identify shared variable accesses and thread synchronization primitives. The algorithm callbacks are inserted at these points. Once instrumented, the instrumented program with the tool is invoked simply by running it inside a Java Virtual Machine (JVM). The race detection happens as the program executes and a data race report is presented at the end of execution. Output of the tool will be in the form of binary capture data that can be replayed after the completion of the run. In addition, it is planned to generate a human readable report from the capture data as a secondary output.

5.3. FastTrack

FastTrack is a high-performance dynamic race detection algorithm that performs significantly faster than most other precise dynamic race detectors. FastTrack uses an adaptive representation for happens-before relations to provide constant-time fast paths for these common cases without any loss of precision or correctness in the general case. In particular, it replaces the linear-time vector clocks with simple structures which, for most of the cases, require constant space and time without hurting precision of the algorithm. Being precise, it guarantees to find at least one race for a racy memory in a given execution. It reports a race between concurrent memory accesses to a variable without proper ordering or synchronization since, principally, it relies on happens-before relations between events.

The input of the tool is either the Java byte code that will be instrumented for race detection in run time or the java run time itself for dynamic instrumentation at run time. Output of the tool will be in the form of binary capture data and a human readable report.

5.4. DRDCheck Hybrid

5.4.1. Introduction

In this research, Ericsson and Yasar University have collaborated to develop a novel data race algorithm. The existing DRDCheck tool [61] has been selected as starting point since the tool is an open source on-the-fly detector of data races. The DRDCheck tool implements a Happens-Before algorithm that uses a logging mechanism. Every shared memory access is logged to see that it “happens before” prior accesses to the same location. Moreover, the Happens-Before algorithm [63] is dependent on the scheduler and thread interleaving.

The DRDCheck tool is however vulnerable to false alarms, hence we have considered that combining happens-before with a Lockset algorithm [62, 65] can improve detection accuracy.

5.4.2. Method

For simplicity, we decided to report READ-WRITE races only [64] and to ignore WRITE-WRITE races, reference assignments, and data races from third-party codes. The main achievement of our work in ASSUME is that we included a LockSet algorithm [62] into DRDCheck and thus created a new Hybrid algorithm inside the DRD Check tool.

The new tool combining happens-before and LockSet is called DRDCheck Hybrid Data Race Detector. Ericsson use cases ENK_UC01 (NE) and ENK_UC02 (VRC MM) have been investigated by the original DRDCheck tool and by the DRDCheck Hybrid Data Race Detector as far as this was possible.

The Hybrid Data Race Detector has been developed in JDK 7. For the ENK_UC02 (VRC MM) use case, a JDK 6 compliant version of the Hybrid agent would be needed. Creating such a JDK 6 compliant version is ongoing work.

5.4.3. Evaluation

[KPI 4.3] The original DRDCheck tool found 47 distinct races on the ENK_UC01 (NE) use case. A closer examination revealed that 43 of these were false positives.

The Hybrid Data Race Detector found 30 distinct races on the ENK_UC01 (NE) use case. Only 2 of these were classified as false positives by a closer examination.

[KPI 4.2] The average latency was also measured and found to be 698 milliseconds for a transaction in ENK_UC01 (NE) with the original DRDCheck tool, but only 70 milliseconds with the Hybrid Data Race Detector.

In summary, utilizing the Hybrid tool, a significant reduction of false positives (from 91% to 6%) in ENK_UC01 (NE) was achieved. Furthermore, the performance of the hybrid data race detector got 10 times better than with the old DRDCheck tool for the same use case.

The work of extending DRDCheck to DRDCheck Hybrid was done in the third year of ASSUME. Therefore, this section is new compared with D5.1.

6. Conclusions and Discussion

This deliverable presents the state of various tools for the analysis of concurrent behaviour at the end of the ASSUME project. The tools fall into different groups according to whether they operate on the model level, on the level of C source code, or closer to the hardware, and whether they perform static analysis or observe the runtime behaviour.

The deliverable covers the work of all partners active in WP5 during the project. It is based on D5.1, which was delivered at Month 24, but contains substantial updates reflecting the work done during the third year of the project.

References

[1] Lukas Mäurer, Tanja Hebecker, Torben Stolte, Michael Lipaczewski, Uwe Möhrstädt, Frank Ortmeier. "On Bringing Object-Oriented Software Metrics into the Model-Based World – Verifying ISO 26262 Compliance in Simulink." System analysis and modeling: models and reusability, pp. 207-222. Springer International Publishing, Berlin, 2014.

[2] Yanja Dajsuren, Mark G.J. van den Brand, Alexander Serebrenik, Serguei Roubtsov. "Simulink models are also software: modularity assessment." Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures, pp. 99-106. ACM, New York, NY, 2013.

[3] William Vesely, Joanne Dugan, Joseph Fragola, Joseph Minarick, Jan Railsback. "Fault Tree Handbook with Aerospace Applications." NASA, Washington, DC, 2002.

[4] Sander Stuijk, Marc Geilen and Twan Basten. "SDF3: SDF For Free." 6th International Conference on Application of Concurrency to System Design (ACSD 2006), pp. 276-278, June 2006. SDF3 is available via www.es.ele.tue.nl/sdf3

[5] Edward A. Lee, David G. Messerschmitt. "Synchronous data flow." Proceedings of the IEEE 75.9, pp. 1235-1245, 1987.

[6] Greet Bilsen et al. "Cycle-static dataflow." IEEE Transactions on signal processing 44.2, pp. 397-408, 1996.

[7] Marc Geilen, Sander Stuijk. "Worst-case performance analysis of synchronous dataflow scenarios." Eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, 2010.

[8] Rajeev Alur, David L. Dill. "A theory of timed automata." Theor. Comput. Sci. 126, 2, pp. 183-235, April 1994.

[9] Marc Geilen. "Synchronous dataflow scenarios." ACM Transactions on Embedded Computing Systems (TECS) 10.2, 16, 2010.

[10] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, Michael González Harbour. "Influence of Different System Abstractions on the Performance Analysis of Distributed Real-Time Systems." Proceedings of the 7th ACM & IEEE International Conference on Embedded Software, pp. 193-202, 2007.

[11] Ingo Stierand, Philipp Reinkemeier, Tayfun Gezgin, Purandar Bhaduri. "Real-Time Scheduling Interfaces and Contracts for the Design of Distributed Embedded Systems". 8th IEEE International Symposium on Industrial Embedded Systems (SIES'13), pp. 130-139, 2013.

[12] George C. Necula, Scott McPeak, S. P. Rahul, Westley Weimer. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs." Conference on Compiler Construction (CC 2002), pp. 213–228, 2002.

[13] Kalmer Apinis. "Frameworks for analyzing multi-threaded C." PhD thesis, Institut für Informatik, Technische Universität München, June 2014.

[14] Wolfgang Ecker, Wolfgang Müller, Rainer Dömer. "Hardware-dependent Software." Springer Netherlands, 2009.

[15] Baccelli, F., Cohen, G., Olsder, G.J., Quadrat, J.P.: "Synchronization and linearity: an algebra for discrete event systems", 2001.

[16] Bekooij, M., Moreira, O., Poplavko, P., Mesman, B., Pastrnak, M., van Meerbergen, J.: "Predictable Embedded Multiprocessor System Design", pp. 77–91. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[17] Bhattacharya, B., Bhattacharyya, S.: Parameterized dataflow modeling for DSP systems. Signal Processing, IEEE Transactions on 49(10), 2408–2421 (Oct 2001)

[18] Bhattacharyya, S.S., Deprettere, E.F., Theelen, B.D.: Dynamic dataflow graphs. In: Bhattacharyya, S.S., Deprettere, E.F., Leupers, R., Takala, J. (eds.) Handbook of Signal Processing Systems, pp. 905–944. Springer New York (2013)

[19] Bhattacharyya, S.S., Deprettere, E.F., Theelen, B.D.: Dynamic Dataflow Graphs, pp. 905–944. Springer New York, New York, NY (2013)

[20] Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.: Cycle-static dataflow. Signal Processing, IEEE Transactions on 44(2), 397–408 (Feb 1996)

[21] Buck, J.T.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, EECS Department, University of California, Berkeley (1993)

[22] Dasdan, A., Irani, S.S., Gupta, R.K.: Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In: Proceedings of the 36th Annual ACM/IEEE Design Automation Conference. pp. 37–42. DAC ’99, ACM, New York, NY, USA (1999)

[23] Deroui, H., Desnos, K., Nezan, J.F., Munier-Kordon, A.: Throughput evaluation of DSP applications based on hierarchical dataflow models. In: International Symposium on Circuits and Systems (ISCAS) (2017)

[24] Desnos, K., Pelcat, M., Nezan, J.F., Bhattacharyya, S.S., Aridhi, S.: PiMM: Parameterized and interfaced dataflow meta-model for MPSoCs runtime reconfiguration. In: 2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). pp. 41–48 (July 2013)

[25] Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity - the Ptolemy approach. Proceedings of the IEEE 91(1), 127–144 (Jan 2003)

[26] Geilen, M.: Reduction techniques for synchronous dataflow graphs. In: Proceedings of the 46th Annual Design Automation Conference. pp. 911–916. DAC ’09, ACM, New York, NY, USA (2009)

[27] Geilen, M., Tripakis, S., Wiggers, M.: The earlier the better: A theory of timed actor interfaces. Tech. Rep. UCB/EECS-2010-130, EECS Department, University of California, Berkeley (Oct 2010)

[28] Ghamarian, A.H., Geilen, M.C.W., Stuijk, S., Basten, T., Theelen, B.D., Mousavi, M.R., Moonen, A.J.M., Bekooij, M.J.G.: Throughput analysis of synchronous data flow graphs. In: Proceedings of the Sixth International Conference on Application of Concurrency to System Design. pp. 25–36. ACSD ’06, IEEE Computer Society, Washington, DC, USA (2006)

[29] de Groote, R., Kuper, J., Broersma, H., Smit, G.J.M.: Max-plus algebraic through-put analysis of synchronous dataflow graphs. In: 38th Euromicro Conference on Software Engineering and Advanced Applications. pp. 29–38 (Sept 2012)

[30] Ha, S., Oh, H.: Decidable Dataflow Models for Signal Processing: Synchronous Dataflow and Its Extensions, pp. 1083–1109. Springer New York, New York, NY (2013)

[31] Heidergott, B., Olsder, G.J., Van Der Woude, J.: Max Plus at work: modeling and analysis of synchronized systems: a course on Max-Plus algebra and its applications. Princeton University Press (2014)

[32] Kavi, K.M., Buckles, B.P., Bhat, U.N.: A formal definition of data flow graph models. IEEE Transactions on Computers C-35(11), 940–948 (Nov 1986)

[33] Nelson, A., Goossens, K., Akesson, B.: Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor SOC. Journal of Systems Architecture 61(9), 435 – 448 (2015)

[34] Piat, J., Bhattacharyya, S.S., Raulet, M.: Interface-based hierarchy for synchronous data-flow graphs. In: 2009 IEEE Workshop on Signal Processing Systems. pp. 145–150 (Oct 2009)

[35] Ritz, S., Pankert, M., Zivojinovic, V., Meyr, H.: Optimum vectorization of scalable synchronous dataflow graphs. In: Application-Specific Array Processors, 1993. Proceedings., International Conference on. pp. 285–296 (Oct 1993)

[36] Sriram, S., Bhattacharyya, S.S.: Embedded Multiprocessors: Scheduling and Synchronization. CRC Press, Inc., Boca Raton, FL, USA, 2nd edn. (2009)

[37] Stuijk, S., Geilen, M., Theelen, B., Basten, T.: Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In: Embedded Computer Systems (SAMOS), 2011 International Conference on. pp. 404–411 (July 2011)

[38] Stuijk, S.: Predictable mapping of streaming applications on multiprocessors. Ph.D. thesis, Eindhoven University of Technology (2007)

[39] Tripakis, S., Bui, D., Geilen, M., Rodiers, B., Lee, E.A.: Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs. ACM Trans. Embed. Comput. Syst. 12(3), 83:1–83:26 (Apr 2013)

[40] Hassan Salehe Matar, Serdar Tasiran and Didem Unat. EmbedSanitizer: Runtime Race Detection Tool for 32-bit Embedded ARM. The 17th International Conference on Runtime Verification, September 13-16, Seattle, USA.

[41] Kenneth Y. Jørgensen, Kim G. Larsen, Jiří Srba. "Time-Darts: A Data Structure for Verification of Closed Timed Automata". 7th Conference on Systems Software Verification (SSV), pp. 141-155, 2012.

[42] Johan Bengtsson, Wang Yi. "Timed Automata: Semantics, Algorithms and Tools". Lectures on Concurrency and Petri Nets: Advances in Petri Nets, pp. 87-124, 2004.

[43] Leslie Lamport. "Real-Time Model Checking Is Really Simple". Correct Hardware Design and Verification Methods: 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, pp. 162-175, 2005.

[44] Elena Fersman, Paul Pettersson, Wang Yi. "Timed Automata with Asynchronous Processes: Schedulability and Decidability". Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference, TACAS 2002, Grenoble, France, pp. 67-82, 2002.

[45] Telkar, N., Chatha, K. S., Lee, Y., Gannod, G., & Wong, E. (n.d.). A Technique for Verification of Race Conditions in Real-time Systems, 1–15.

[46] Coulouris, George (2012). Distributed Systems Concepts and Design. Pearson. p. 716. ISBN 978-0-273-76059-7.

[47] E. C. Coffman, Michael John Elphick, A. Shoshani: System Deadlocks. In: Computing Surveys. Band 3, Nr. 2, 1971, S. 67–78.

[48] Engler, D., & Ashcraft, K. (2003). RacerX: effective, static detection of race conditions and deadlocks. Race, 37(5), 237–252.

[49] J. Corbett. Evaluating deadlock detection methods for concurrent software. IEEE Transactions on Software Engineering, 22(3), 1996.

[50] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programming. ACM Transactions on Computer Systems, 15(4):391–411, 1997.

[51] D. P. Helmbold and C. E. McDowell. A taxonomy of race detection algorithms. Technical Report UCSC-CRL-94-35, 1994.

[52] E. Clarke, O. Grumberg, and D. Peled. Model Checking. MIT Press, 1999.

[53] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In IEEE International Conference on Automated Software Engineering (ASE), 2000.

[54] A. Miné. Static Analysis of Embedded Real-Time Concurrent Software with Dynamic Priorities. Electronic Notes in Theoretical Computer Science, 331, 3–39, 2017.

[55] Z. Birnbaum: On the importance of different components in a multi-component system. In: Multivariate Analysis, P. Krishnaiah Ed., New York: Academic Press, 581-592, 1969.

[56] W. E. Vesely, T. C. Davis, R. S. Denning, N. Saltos. Measures of Risk Importance and Their Applications, NUREG/CR-3385, DE83 902819, 1983.

[57] Tripakis, Stavros, et al. "Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs." ACM Transactions on Embedded Computing Systems (TECS) 12.3 (2013): 83.

[58] Gaubert, Stéphane. "Performance evaluation of (max,+) automata." IEEE transactions on automatic Control 40.12 (1995): 2014-2025.

[59] Geilen, Marc, et al. "Performance analysis of weakly-consistent scenario-aware dataflow graphs." *Journal of Signal Processing Systems* 87.1 (2017): 157-175.

[60] Geilen, Marc, and Sander Stuijk. "Worst-case performance analysis of synchronous dataflow scenarios." *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2010.

[61] D. Tsitelov, and V. Trifanov. Dynamic Data Race Detection in Java-programs using synchronization contracts. 2013 Tools & Methods of Program Analysis, 2013.

[62] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems (TOCS), 15(4), 1997.

[63] Lamport, L. 1978. Time, clocks and the ordering of events in a distributed system. Commun. ACM 21(7):558-565.

[64] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. A theory of data race detection. In Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD), 2006.

[65] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst., 15(4):391–411, 1997.

[66] Rajeev Alur, Limor Fix, Thomas A. Henzinger. A Determinizable Class of Timed Automata.

[67] The Clang Team. Clang 8 documentation: ThreadSanitizer https://clang.llvm.org/docs/ThreadSanitizer.html