Affordable Safe & Secure Mobility Evolution

Methodology draft

Industrial requirements analysis

Definition of desired tool extensions and tool flows

Interface definitions

Deliverable D4.1

Deliverable Information			
Nature	Document	**Dissemination Level**	Public
Project	ASSUME	**Project Number**	14014
Deliverable ID	D4.1	**Date**	12.01.2017
Status	Final	**Version**	1.0
Contact Person	Dumitru Potop-Butucaru	**Organisation**	Inria
Phone	+33-180494010	**E-Mail**	dumitru.potop@inria.fr

Author Table

Name	Company	Email
Dumitru Potop Butucaru	Inria	dumitru.potop@inria.fr
Gunther Siegel	ESTEREL TECHNOLOGIES	gunther.siegel@ansys.com
Xavier Leroy	Inria	xavier.leroy@inria.fr
Reinhold Heckmann	AbsInt	heckmann@absint.com
Marco Bekooij	NXP/UT	marco.bekooij@nxp.com

Change and Revision History

Version	Date	Reason for Change	Affected sections
0.1	05.12.2016	Initial version	All
0.2	26.12.2016	ESTEREL TECHNOLOGIES Contribution	2.3.2, 2.4.2
0.3	12.01.2016	CompCert section (INRIA and AbsInt)	2.3.4
0.9	08.02.2017	Merged all contributions	All
1.0	16.02.2017	Minor local updates (typos…) & minor local updates on ESTEREL TECHNOLOGIES contribution	All

Table of Contents

AUTHOR TABLE 2

CHANGE AND REVISION HISTORY 2

TABLE OF CONTENTS 3

1. EXECUTIVE SUMMARY 4

2. AVIONICS CLUSTER 5

2.1. Analysis of industrial use cases and requirements 5

2.2. Methodology principles 5

2.3. Proposed tool flow and desired tool extensions 6

2.3.1. Verified Lustre 8

2.3.2. Scade KCG6 parallel 8

2.3.3. Heptagon/Lopht 9

2.3.4. CompCert 11

2.3.5. Litmus 11

2.4. Interface definitions 12

2.4.1. Common API for code generation on Kalray MPPA256 12

2.4.2. Calling conventions used in SCADE code generation 12

2.4.2.1. Input model annotations 13

2.4.2.2. Communication primitives 13

2.4.2.3. Worker 14

2.4.2.4. Integration on Specific Targets 14

2.4.3. Non-functional annotations in Heptagon 14

3. FAULT TOLERANT VISION CLUSTER 16

3.1. Analysis of industrial use cases and requirements 16

3.2. Proposed tools and methods 16

3.2.1. HAPI dataflow simulator 16

3.2.2. Definition and computation of the architecture vulnerability factor (AVF) 16

3.2.2.1. Simulation based AVF measurement: Counting Residence Times 17

3.2.2.2. AVF = PVF*HVF 18

3.2.2.3. Approximate-compute oriented PVF 19

3.2.3. Approximate DMR-based Reliable Vector Processor 21

3.2.4. Particle filter based state estimation algorithm 21

3.2.5. Fault-Tolerant Network-on-Chip architecture 21

4. SAFE VEHICLE CONTROL CLUSTER 23

4.1. Analysis of industrial use cases and requirements 23

4.2. Proposed tools and methods 23

4.2.1.1. Xenoclea 24

4.2.1.2. Dezyne 24

5. CONSUMER ELECTRONICS CLUSTER 25

5.1. Benchmark Data Structures and Synchronization Primitives 25

5.2. Proposed tool flow and desired tool extensions 26

5.2.1. Global toolflow organization 26

5.2.2. Tool status and needed extensions 27

5.2.2.1. CIVL \ Concurrent Boogie 27

5.2.2.2. BoogieASM 27

5.3. Interface definitions 27

5.3.1. Common Interface for CIVL and BoogieASM 27

In addition to the language allowed by CIVL, we require following restrictions / extensions on the input program *P*: 27

REFERENCES 28

1. Executive Summary

Work in WP4 is organized in 4 clusters, according to participation in case studies. General principles, common to multiple clusters, have been identified, such as the reliance on formal methods, or the use of data-flow formalisms. However, early on in the project it has been recognized by partners that the 4 classes of use cases covered here need specific tools, and specific methodologies. This explains the clear partition of project partners among the 4 clusters, which are:

- Avionics. WP4 participants: Inria, ENS, AbsInt, Kalray, ESTEREL TECHNOLOGIES. Use case providers: Airbus, Safran Aircraft Engines SAS, Safran Electronics & Defense.
- Fault tolerant vision. WP4 participants: Tue, NXP, UT, Recore. Use case providers: NXP/Recore.
- Safe vehicle control: WP4 participant: Verum, UT, TNO. Use case providers: TNO, VDL.
- Consumer electronics. WP4 participants: Koç University, Koç Sistem.

This organization of the work is followed in the organization of the various parts of this deliverable. In the first three clusters, industrial case studies and requirements have been analyzed, and comprehensive design methodologies have been proposed. The delay in the funding of the Turkish partners means that the fourth cluster is less advanced. However, significant results are presented in this deliverable.

2. Avionics cluster

The development of avionics embedded applications is subject to strict certification requirements, which resulted in highly formalized and largely automated development processes. Our work seeks to improve these processes by providing solutions to the following problems:

- Safe and efficient automatic parallelization of avionics applications onto multi-/many-core platforms.
- Providing formal correctness guarantees for avionics software.

2.1. Analysis of industrial use cases and requirements

For the first 18 months of the project, work of the avionics cluster has focused on three case studies:

- *AIR_UC01 (from Airbus)* – control/command synchronous application of high criticality
- *SAF_UC01 (from Safran)* – synchronous application of high criticality
- *SAF_UC2 (from Safran)* – mixed criticality application

These applications share key features:

- High-level data-flow functional modelling (SCADE, Simulink, or an in-house formalism).
- They are either high criticality or mixed criticality applications, meaning that code generation must aim for high integrity implementations.

On all 3 applications the common objective is safe and efficient automatic parallelization, with Kalray MPPA as the preferred initial execution platform. In addition to parallelization, depending on the case study, main objectives are:

- Formal verification of the correctness of the implementation flow, either by compiler verification or by translation validation methods. Formal correctness proofs can be useful even on sequential code generators.
- Parallelization under real-time requirements.
- Safe interaction between software components of different criticalities.

2.2. Methodology principles

Following long-standing practice in the avionics industry, the design methodology we propose is based on the use of synchronous languages for the high-level functional specification of applications. More specifically, we shall be using various dialects of the Lustre synchronous data-flow language, whose industrial version SCADE is commercialized by ESTEREL TECHNOLOGIES. The standardization of the functional specification level is essential allowing the cost-effective construction of tools. When the industrial process requires the use of a functional specification formalism different from Lustre/SCADE, process-specific automatic translation tools are needed to ensure seamless integration by translation into a Lustre/SCADE dialect.

For parallel and/or real-time implementation, our methodology also requires the formalization of the non-functional specification. Non-functional specification is formed of a description of the execution platform and a set of non-functional requirements of various types. The content of the non-functional specification depends on the code generation objectives. It can range from a few parallelization annotations, when the objective is to produce multi-threaded C code, to complex descriptions of both the execution platform (topology, arbitration, timing characterization) and the non-functional requirements (real-time, partitioning, allocation, etc.).

Starting from the high-level functional and non-functional specifications, our methodology requires the use of high-level compilers to produce C code and, if needed, OS configuration files. Resulting C code is then compiled, along with legacy business code and with platform libraries to produce the executable code of the implementation.

The result is a seamless flow of automatic transformations going all the way from high-level specification to running implementation. Such a flow of transformations ensures the correctness of the resulting implementation with respect to the high-level specification provided that:

- the platform description faithfully describes the behaviour of the execution platform (HW, libraries, and possibly OS)
- the high-level compiler and C compiler are correct
- the (optional) process-specific importer tool is correct

2.3. Proposed tool flow and desired tool extensions

During the first 18 months of this project, we have integrated the tools of the various partners to fit the global methodology defined above and to address the needs expressed through the industrial use cases. To this end, work has advanced on several axes:

1. Formally proved compilation from Scade/Lustre to sequential executable code. The formally proved compiler Verified Lustre from a dialect of Lustre to C has been completed and interconnected with the CompCert C compiler.
2. Generating parallel code for the industrial use cases and the project platforms (including, but not restricted to Kalray MPPA), for both real-time and non-real-time targets.
    a. The Scade KCG compiler has been extended to allow parallel code generation and manual allocation of the parallel threads onto the target MPPA256 platform.
    b. The Heptagon/Lopht tool flow has been improved with a simpler language for specifying non-functional requirements. A formally proved translation validation tool has been developed for a restricted version of the Lopht tool.

    In both cases, code generation targets simple APIs facilitating the retargeting of the code generator.
3. Design of run-time communication and synchronization primitives that are verified against weakly consistent shared memory models (Kalray MPPA, Arinc 653), using the Litmus tool.
4. Construction of process-specific automatic translation tools, application to use cases, and providing feedback to use case providers. Most notably, a first version of the Airbus-specific importer tool has been used in the Year 1 *AIR_UC01* demonstrator.

Work along these axes resulted in the tool flow whose current structure is presented in Fig. 1. The tool flow incorporates the tools of all cluster partners.

The inputs of the tool flow, on the left, can be roughly divided in 2 categories:

- Functional specification (blue boxes) is provided in a dialect of Lustre/Scade, possibly extended to allow a more natural modelling of multi-rate or multi-period systems. For the Velus (verified Lustre) compiler, this language is called itself Velus, and it is a strict sub-set of Lustre. The other dialects (Scade, Lustre, Heptagon) are quite close to one another in syntax and expressive power, to the point where some tools (e.g. the Heptagon compiler) can already take several of them as input without changes in the internal data structures. The ASSUME project will certainly lead to convergence between these formalisms, but specific aspects should remain, given that Scade is designed by ESTEREL TECHNOLOGIES as part of a commercial product, whereas Heptagon and Lustre are academic tools.
- Non-functional specification is divided between platform descriptions (in red) and non-functional requirements (in purple). Typical non-functional requirements are real-time requirements, allocation requirements, etc. Typical platform description features include the number and type of processors, the memory hierarchy, the on-chip networks, etc. During the duration of ASSUME, both Scade and Heptagon were extended with annotations allowing the specification of non-functional requirements (e.g. parallelization, allocation, or real-time). This means that the frontier between the functional and non-functional specifications becomes increasingly blurred. One of the expected outcomes of ASSUME is better languages for functional and non-functional specification, adapted to the needs of the use cases.

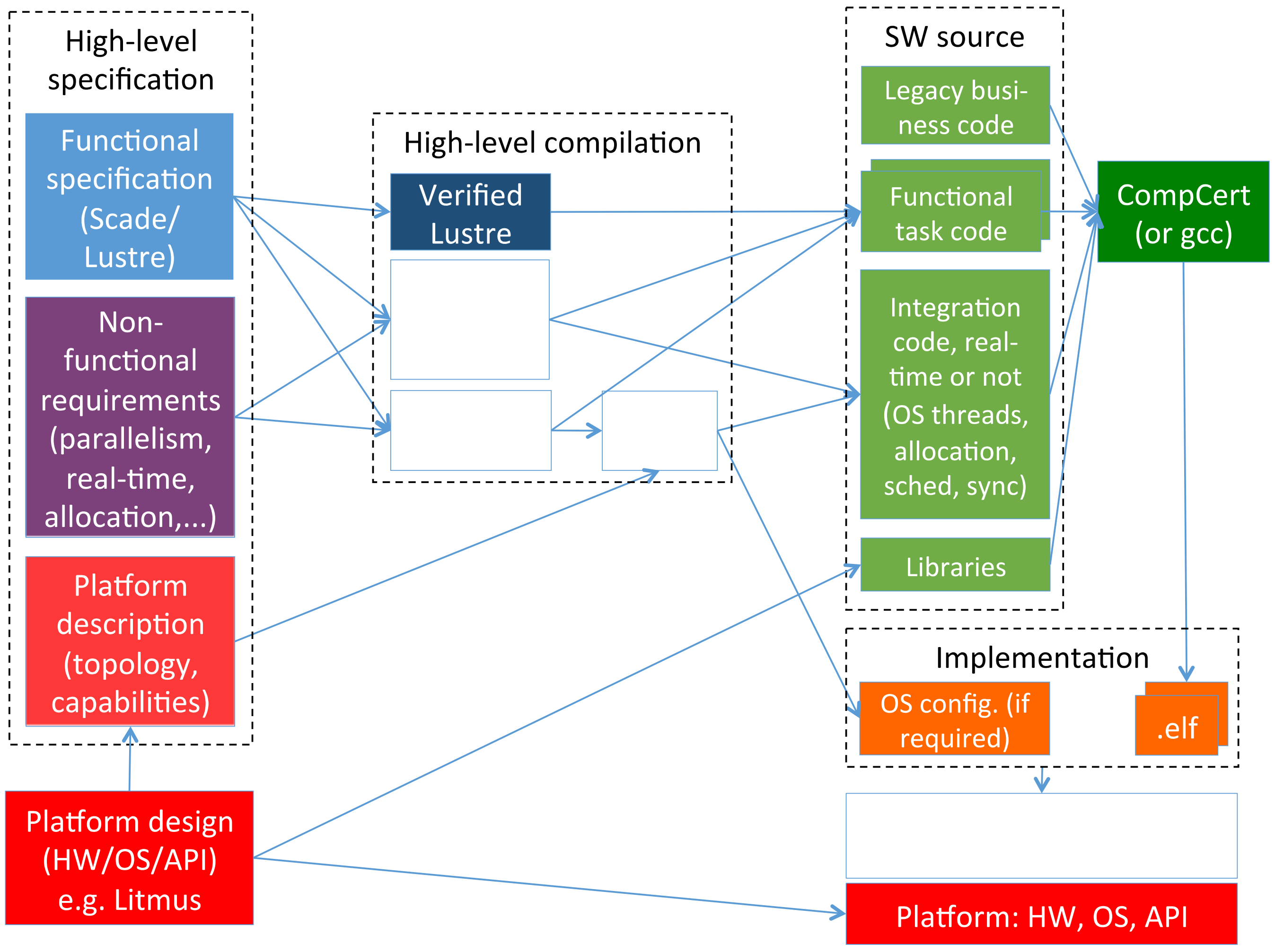

High-level specification
Functional specification (Scade/ Lustre)
Non-functional requirements (parallelism, real-time, allocation,...)
Platform description (topology, capabilities)
Platform design (HW/OS/API) e.g. Litmus
High-level compilation
Verified Lustre
SW source
Legacy business code
Functional task code
Integration code, real-time or not (OS threads, allocation, sched, sync)
Libraries
CompCert (or gcc)
Implementation
OS config. (if required)
.elf
Platform: HW, OS, API

Figure 1 Tool flow organization of the WP4 avionics cluster

The functional and non-functional specifications are used by the three high-level compilation tools whose development embodies efforts on the various R&D axes:

- Velus focuses on formally verified compilation
- The extended Scade v6 compiler focuses on generation of parallel code where allocation is manually specified.
- The Heptagon and Lopht compilers are used together to allow the generation of parallel real-time code for which allocation and real-time scheduling can be automatically synthesized.

In all 3 cases, the output of the high-level compiler is C code, and possibly some configuration information directed to the lower-level tools (loader) and to the runtime. In addition to generated code, intermediate artefacts can be output, such as the scheduling tables or allocation information, facilitating external verification of the system correction.

If the objective is to provide formal correctness guarantees, then the C code output by the high-level compilation tools must be in turn compiled using CompCert (but gcc is currently used on the Heptagon/Lopht branch, as it allows very precise control over memory allocation and optimization).

The Litmus tool is needed to validate efficient communication/synchronization primitives against the model of the memory sub-system. The parallel code generators will use these primitives.

The following sub-sections present the current status and desired tool extensions of the individual partners' tools.

2.3.1. Verified Lustre

Solid progress has been made on extending the Vélus compiler to turn it into a practical tool that can run on real code. An overview of the current state is shown in Figure 2. Most recently, we have implemented a parser and elaboration routine, added the scheduling pass, improved the ability to display intermediate results, and refined the final correctness lemma. These results are explained in more detail in the following.

The parser was implemented using the menhir software which also generates proofs of completeness and correctness in the Coq proof assistant. The elaboration routine turns an unannotated syntax tree into one annotated with types and clocks. In the context of a verified compiler like ours, it also verifies the correctness of these annotations and several other required invariants. That is, it formally validates assumptions used in later passes and their proofs of correctness.

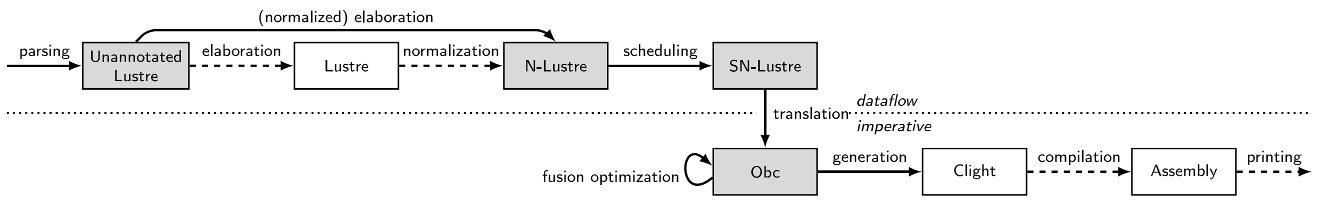


Figure 2. Velus toolchain status

The scheduling pass orders the dataflow equations based on their interdependencies. The semantics of these equations is independent of their order, but this order is maintained in the generation of imperative code whose semantics does depend on it. Our implementation calls an external OCaml routine to find a suitable schedule or print an error message explaining why this is impossible. The reordering itself is done by a verified sorting routine (from the Coq standard library) and then validated by a verified decision procedure. This approach gives greater liberty in implementing equation scheduling and maintains the same high level of correctness guarantees but without requiring complicated proofs. Besides satisfying the data dependencies, the scheduler tries to place similarly activated equations together to increase the effectiveness of a later fusion optimization on the imperative (Obc) code. Finding an optimal ordering is NP-hard, so we implement a greedy algorithm based on simple heuristics.

We have implemented several “pretty printers” to display the results of intermediate compilation passes. This is straightforward and standard practice, but greatly aids debugging and practical evaluation of the tool chain.

The final correctness lemma gives a guarantee that the assembly code generated by our compilation passes coupled with the CompCert ones correctly implements the dataflow semantics of the input program. This end-to-end proof has now been stated and proved. Arriving at this point involved solving several technicalities around coinductive proofs and a change from the “big step” semantics used in the proof of correctness for the code generation pass (from Obc to Clight) to the “small step” semantics necessary to state the final lemma.

The new results described above and those of our earlier report on the translation and generation passes have been accepted for publication at the PLDI 2017 conference.

The current version of the compiler functions with an end-to-end correctness guarantee, but it only accepts a subset of “normalized” Lustre programs. We are working on implementing the normalization pass (from Lustre to N-Lustre) and proving its correctness. This task is non-trivial. It requires implementing rewriting by successive substitutions, introducing a new semantic model for the Lustre language, and showing the correctness of the rewriting with respect to the semantics.

2.3.2. Scade KCG6 parallel

An extension of our safety-critical qualified code SCADE Suite KCG generator has been prototyped. This prototype allows efficient code generation for multi/many core targets. The parallelization is currently user-driven. The user identifies parallel regions formed of operator instances that can be executed in parallel, splitting the model into independent components well-balanced with respect to their WCET.

This prototype generates tasks that communicate with one-to-one channels (i.e. Kahn process networks). One task is generated for the root operator and one task for each instance of operator in a parallel subset. The generated C code is target agnostic, macros are used for all operations (communication, synchronisation, …). A dedicated integration step is in charge of generating the main function and the macros definition for a given target. To verify the portability, we have developed instantiation for Pthread and Win32 API with semaphores. We also worked with Kalray to have an instantiation for their MPPA many-core architecture.

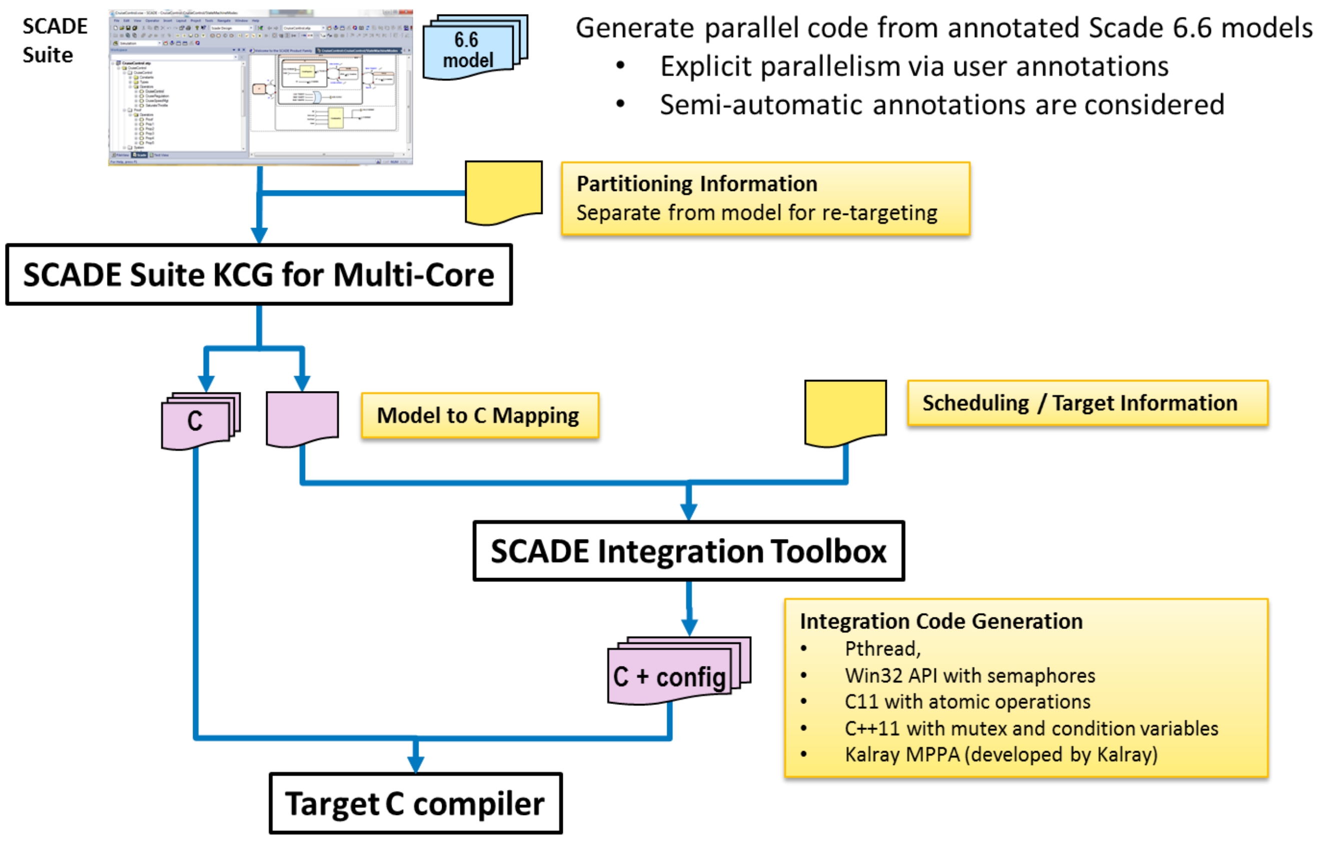

SCADE Suite
6.6 model
Generate parallel code from annotated Scade 6.6 models
• Explicit parallelism via user annotations
• Semi-automatic annotations are considered
Partitioning Information
Separate from model for re-targeting
SCADE Suite KCG for Multi-Core
C
Model to C Mapping
Scheduling / Target Information
SCADE Integration Toolbox
C + config
Integration Code Generation
• Pthread,
• Win32 API with semaphores
• C11 with atomic operations
• C++11 with mutex and condition variables
• Kalray MPPA (developed by Kalray)
Target C compiler

Figure 3: Scade Multi-Core Code Generation Flow

2.3.3. Heptagon/Lopht

Heptagon is both a Lustre dialect and an open-source compiler for this language. For the scope of this section, we shall disambiguate between the two by using the name heptc for the compiler. The heptc compiler can be used for two types of tasks:

- Generating sequential C code from synchronous programs (nodes) written in a Lustre dialect (much like SCADE KCG or Velus). Accepted Lustre dialects include Heptagon and Scade v4. The Scade front-end has been added during ASSUME to allow the handling of case studies.
- Translating Heptagon programs into the non-hierarchic data-flow language taken as input by the Lopht tool. In this case, an extension of heptc takes as input an extension of the Heptagon language, called Heptagon+NFP, where program annotations allow the specification of non-functional requirements, as described below, in section 2.4.3. We name this extension heptc+NFP.

The Lopht tool takes as input the non-hierarchic data-flow and the non-functional requirements output by heptc and a description of the execution platform (topology, WCETs). It either produces parallel implementation C code that is both functionally correct and respects the non-functional requirements, or reports why it was not able to produce such an implementation.

Given the capabilities of the tools, we advocate their use following the methodology and tool flow of Fig. 4. The flow starts with an importer tool that translates the industrial specification into Heptagon input. This tool is needed, as industrial use case specifications from both Airbus and Safran do not fit directly into the synchronous model (both in term of format and of underlying computational model. We have developed a tool that generates a functionally equivalent Lustre code from these specifications. This convertor tool is currently based on the specifications provided for the *AIR_UC01* use case, where specification is done in a formalism specific to the Airbus process. This work will be extended to fit the needs of the Safran *SAF_UC01* use case. This tool builds internally a dependence graph between the different tasks of the application. We plan to use this graph to analyse the application and to perform some transformation, such as retiming. We chose to write directly by hand the corresponding Lustre code for the SAF_UC2 use case of Safran.

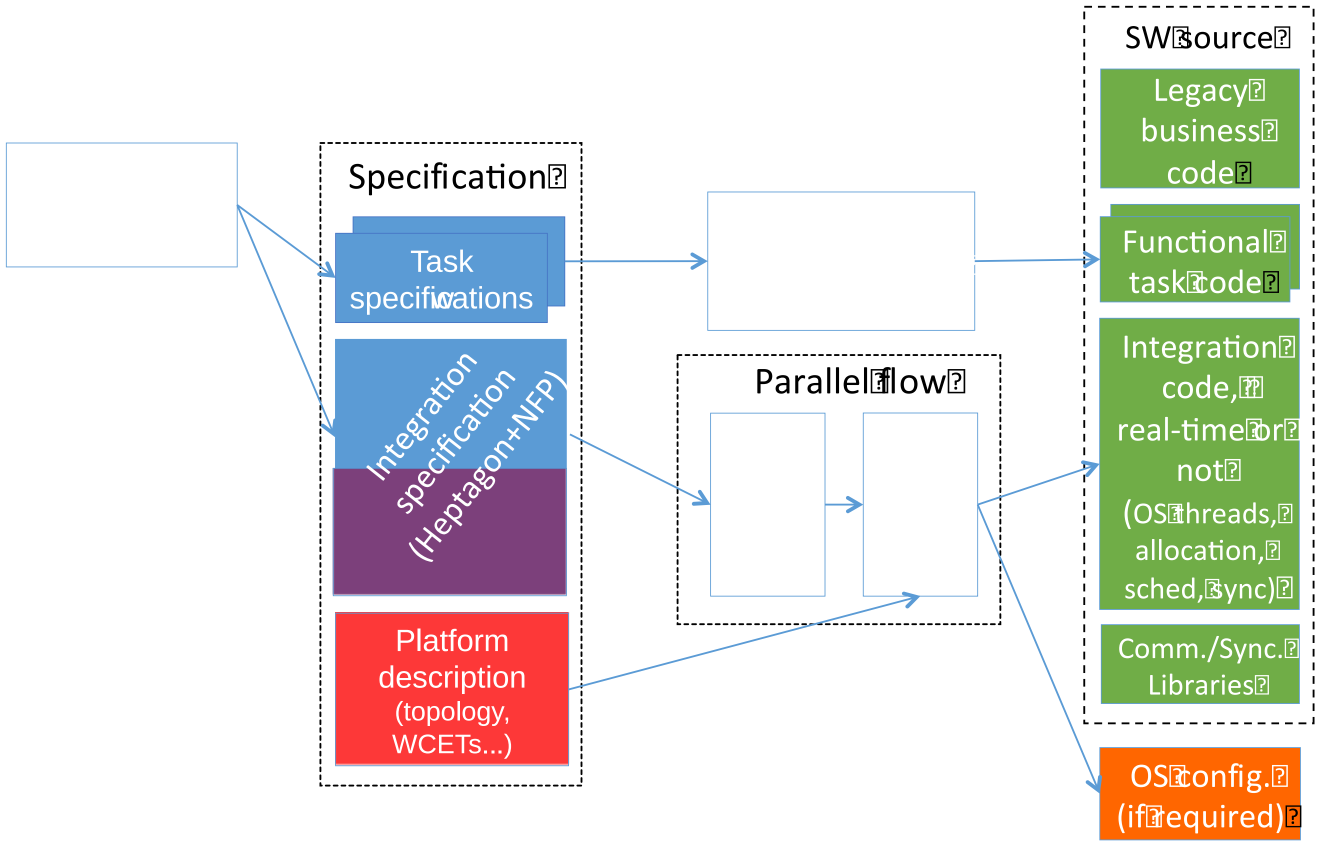

Specification
Task specifications
Integration specification (Heptagon+NFP)
Platform description (topology, WCETs...)
Parallel flow
SW source
Legacy business code
Functional task code
Integration code, real-time or not (OS threads, allocation, sched, sync)
Comm./Sync. Libraries
OS config. (if required)

Figure 4. Heptagon/Lopht tool flow

The Heptagon code is divided in two parts: the specification of the sequential tasks and the integration specification. A task specification consists of a synchronous component that must be separately compiled into one piece of sequential code. It contains no non-functional annotations. Compilation can either be done using heptc or (modulo Lustre dialect adaptations) with SCADE KCG or Velus. The integration specification is the system-level specification that defines the dependencies between tasks and contains all non-functional requirements (hence the mixed blue+purple color). All data-flow parallelism in the integration specification can be exploited by Lopht during allocation and real-time scheduling.

Expected extensions of the Heptagon and Lopht tools are the following:

- Extension of Heptagon with language constructs allowing the specification of n-synchronous (Cohen *et al.*, 2006) behaviors, and extension of heptc to allow the compilation of n-synchronous programs. This should facilitate the specification and implementation of multi-period systems.
- Further improvement of the Heptagon non-functional annotations
- Extension of Lopht to generate code using the Kalray MPPA 256 API defined in section 2.4.1.
- Extension of the Lopht translation validation tool to cover code generation on the Kalray MPPA 256 platform.

2.3.4. CompCert

Since 2015, the CompCert compiler is commercially available. What sets CompCert apart from any other production compiler is that it is formally verified, using machine-assisted mathematical proofs, to be exempt from miscompilation issues. In other words, the executable code it produces is proved to behave exactly as specified by the semantics of the source C program. This level of confidence in the correctness of the compilation process is unprecedented and contributes to meeting the highest levels of software assurance. In particular, using the CompCert C compiler is a natural complement to applying formal verification techniques (static analysis, program proof, model checking) at the source code level: the correctness proof of CompCert C guarantees that all safety properties verified on the source code automatically hold as well for the generated executable.

CompCert has been developed by INRIA and licensed by AbsInt for commercial exploitation. In ASSUME, AbsInt is working at improving the usability of the compiler to make it competitive in this respect with existing compilers. Since Deliverable D4.0, the following improvements have been implemented:

- The robustness of the frontend has been improved. Now more cases of invalid inputs are treated with proper error messages.
- New command line options have been introduced that allow for better control of the diagnostic output produced by CompCert. It is now possible to activate or suppress certain warnings and additionally to mark them as error.
- CompCert now fully supports C11 anonymous compound types. Such types are considered as transparent for their components so that their named parts may be accessed directly.

In ASSUME's WP4, INRIA is extending the scope of the CompCert methodology to cover the needs of the embedded design process. Extension will concern both front-end, by considering the translation from Scade to C, and the back-end, by considering the generation of code for the Kalray MPPA architecture. Preliminary investigations concerning the possibility of a Kalray backend identified two main difficulties:

The first issue comes from the handling of double-precision floating-point numbers in the MPPA instruction set: such numbers must reside in pairs of consecutively-numbered registers, with the low 32 bits of a number residing in one register and the high 32 bits residing in the other register of the register pair. The register allocation pass of CompCert must be extended to handle the allocation of such register pairs for double-precision FP numbers. This is known to be non-trivial in allocators based on graph coloring such as CompCert's. Moreover, the semantic specifications and proofs of CompCert's last intermediate languages and passes (those occurring after register allocation) also need to be adapted to register pairs and the splitting of double-precision FP numbers in two 32-bit halves.

The second issue is to take advantage of the instruction-level parallelism offered by the MPPA architecture, so as to increase performance. The MPPA is a VLIW (Very Large Instruction Word) architecture, meaning that instructions intended for parallel execution on the processor must be explicitly grouped into so-called bundles by the compiler, while respecting logical dependencies between instructions and hardware limitations on the size and contents of a bundle. Our starting point here is earlier work by Jean-Baptiste Tristan and Xavier Leroy on verified instruction scheduling optimizations for superscalar architectures. The work that remains to be done is to adapt those techniques to VLIW architectures, probably moving the instruction scheduling from the Mach to the Asm intermediate languages of CompCert. Moreover, the techniques of Tristan and Leroy suffer from excessive compilation times in some cases, requiring an improved implementation using hash-consing to handle sharing of data structures.

In order to overcome these difficulties, INRIA plans to extend its workforce by hiring Gergö Barany, a post-doc with extensive compiler back-end experience and interest in mechanized proofs.

2.3.5. Litmus

The tool litmus is part of the diy tool suite <http://diy.inria.fr>: a set of software tools for the design and testing of weak, shared, memory models. More specifically, litmus runs test over actual hardware and collect results. Such tests are violations of the Sequential Consistency model and can be written by hand or, more conveniently, generated by the various generators of the diy tool suite. Those tests are usually written in assembly: supported architectures being ARMv7, ARMv8, PowerPC, MIPS and x86. Litmus consists in a 'compiler' part that translate the test instructions into a specific C file, using inline assembly entangled with specific code that, for instance forks threads, repeat experiments etc. That specific C file is linked with other C file that provides OS support or fixed non-changing infrastructure such as the data structure to collect results.

For exploring Kalray MPPA, we shall use tests written in C, thereby avoiding the burden of writing test generators for the non-conventional VLIW architecture. We shall instead concentrate on the organization or tests --- that is the techniques used for forking threads and collecting results. We have written a few tests totally by hand and succeeded in running our tests on one cluster of the MPPA only. We now plan to use the complete machine. As a more far reaching objective, we shall automatize test production by integrating MPPA specific infrastructure files into litmus.

2.4. Interface definitions

2.4.1. Common API for code generation on Kalray MPPA256

While the main execution platform targeted by the avionics cluster is the Kalray MPPA256 many-core, the ASSUME project is not meant to produce methods and tools fully tailored for only one platform. One particular aspect of platform-independence is code generation. The Kalray MPPA 256 platform provides multiple and complex communication, synchronization, and process management mechanisms in both hardware and software. Following an analysis of the industrial requirements and of the proposed code generation approaches, several partners have identified a minimal “bare metal” API that will be considered for code generation and which:

- Is simple, and thus easy to port on other platforms of interest.
- Is close to hardware, and thus is both efficient and facilitates predictable implementation.
- Is easy to characterize semantically, thus facilitating work on the proof of correctness for parallel code generation.

Our API currently covers code generation for single-cluster shared memory applications that communicate with their environment (other clusters and/or the exterior of the chip). The API consists in only 8 primitives, presented as library functions:

- Invalidation of instruction/data cache lines, and flush of data cache lines, both needed to ensure cache coherency.
- Global synchronization barrier, needed to ensure initial time synchronization of the processors in a cluster.
- Event-driven inter-processor synchronization using simplified binary semaphores (hardware locks).
- Time synchronization by waiting until a specific date (timer polling, not interrupt-based).
- DMA transfers between the SRAM memory of the computing cluster and its environment.

In addition to the API primitives, we also make assumptions on the form of the generated software. We shall assume that execution is non-preemptive, each processor running a single sequential thread.

2.4.2. Calling conventions used in SCADE code generation

The developed prototype generates from an annotated Scade 6.6 model C code with tasks communicating through one-to-one channels. The main program runs the root operator of the Scade program. It runs in parallel with *workers* which repeat the following behaviour:

- await a value on the input channel,
- execute a function
- and then send the result to the output channel.

The parallelization of the Scade model is specified using special occurrences pragmas of the form `#par_name`. All the operator calls with the same pragma are executed in parallel in fork-join parallelism.

C code can be generated by running:

```
> kcg-task -root <root node> -target C <List of xscade/xscade files>
```

The code generated for the root operator is the same as usual, except that the calls to *workers* are replaced by macro calls to send the inputs and await the results on communication channels.

The generated code uses macros defined in `kcg_channel.h`. It is thus independent from the target and from the allocation of workers to computation resources.

2.4.2.1. Input model annotations

The purpose of the annotations is to group several operator instances in one or several *parallel subsets*: each instance in a subset is executed in parallel. No dependencies are allowed between instances of the same subset, so that they can be executed in parallel. The causality analysis has been extended to raise an error if this is not the case.

Instances of the same subset can be put in different operators (if they end up in the same unexpanded operator). Parallel subsets can be nested: an operator in a parallel subset can itself contain another parallel subset.

The annotations are occurrence pragmas of the form `#par_name`, where `name` is the name of the parallel subset. `#par` pragmas can also be put on `map` or `mapi` iterators. In that case, one worker will be created for each instance of the iterated operator.

It is possible to provide the annotations from an external file (named "Partitioning Information" in Figure), given with the `-pragma_file` command-line option. It can be used to provide any pragma, including `par` pragmas. This file should contain lines of the form `<pragmas> <model_path>` where `<pragmas>` is a list of pragmas and `<model_path>` is the path of an element in the model. The syntax of pragmas and model path is given in ESTEREL TECHNOLOGIES SCADE KCG Tool Operational Requirements.

Example of external annotation file:

```
```
#par_1 root/(node1)
#par_1 root/(Node2)
```
```

Instances can also be described using their instance name (eg. `(#bla)`) or both the operator path and instance name (`(N#1)`). The pragmas are attached to all instances matching the path.

2.4.2.2. Communication primitives

Communication is abstracted using macros, which are defined in the `kcg_channel.h` file, which has to be provided by the user. A channel is a structure with several fields containing the values carried by the channel and a field called `data` of type `kcg_channel_data` type, which should be defined in `kcg_channel.h``.

User must define the `KCG_CHANNEL_RECV` and `KCG_CHANNEL_SEND` macros. The generated code first writes to the fields of the channel and then calls `KCG_CHANNEL_SEND` to signal that values are ready. Conversely, it first calls `KCG_CHANNEL_RECV` to await values and then reads the fields of the channel.

Example of generated code for a *root* operator

```
void root(inC_root *inC, outC_root *outC)
{
  N1_in_ch.i1 = inC->i1 - kcg_lit_int32(1);
  KCG_CHANNEL_SEND_N1_in_ch(N1_in_ch);
  F1_in_ch.i1 = inC->i2 * inC->i2;
  KCG_CHANNEL_SEND_F1_in_ch(F1_in_ch);
  KCG_CHANNEL_RECV_N1_out_ch(N1_out_ch);
  KCG_CHANNEL_RECV_F1_out_ch(F1_out_ch);
  outC->o = N1_out_ch.o1 + F1_out_ch.o1;
}
```

The `KCG_DECL_SENDER` and `KCG_DECL_RECEIVER` Macros are used by the generated code to declare a sender and a receiver on a given channel.

Note that implementations are provided in the `tools/tasks` directory for PThreads (`pthread/`), Windows threads (`windows/`), C++11 threads and mutexes (`cxx11`) and C11 atomics (`c11`).

2.4.2.3. Worker

For each operator instance in a parallel subset, a worker called `<operator>_worker` is generated, which takes as input the context of the operator (if any) and:

- awaits a value on the input channel(s)
- executes the step function of the operator
- sends the result to the output channel(s).

Note that the worker only executes one step of the operator. Like the main operator, it has to be put inside a loop to obtain the final behaviour.

The code generated is independent from the target and from the allocation of workers to threads. A simple integration consists in creating one thread for each worker. But it is also possible to put several workers in the same thread.

2.4.2.4. Integration on Specific Targets

Integration to a new target platform consists in providing the code for the macros and instantiating the application (thread creation, memory allocation, …). The main function of the program has to setup the communication channels and run in parallel the main program and the workers.

To perform this task all required information (workers, communication channels, etc.) is stored in a `mapping.xml` file which contains traceability information between the input model and the generated code. The folder `tools/mapping_file` contains a Python API to access the `mapping.xml` file generated by ESTEREL TECHNOLOGIES SCADE KCG. The documentation of this API can be found in `tools/mapping_file/doc`.

The script `tools/mapping_file/examples/multicore/main_gen.py` shows an example of how to generate a main file for Pthread and Win32 API using the mapping file API. The script takes as input an allocation of workers to threads (named “Scheduling / Target information” in Figure). This script also implements more advanced features like:

- it checks that the order chosen for workers in each thread is correct according to the order in which the corresponding channels are used in the root operator (see `check_scheduling` function). This is necessary to avoid a deadlock at runtime.
- it generates a `user_config.h` file which overwrites the definition of communication macros for channels between workers executed in the same thread. In that case, no synchronization is needed.
- …

The `tools/tasks` folders contains several folders providing an implementation of `kcg_channel.h` for a given target.

2.4.3. Non-functional annotations in Heptagon

As explained above, the Heptagon language has been extended with annotations allowing the definition of non-functional requirements. These annotations allow the definition of the system-level integration specification introduced in section 2.3.3. We introduce annotations through the simple example of Fig. 5. In this figure, black program text provides the functional specification (in plain Heptagon language). Non-functional annotations use red text.

Annotations allow the specification of the following requirements:

- Period. A single period can be currently specified, at system level. This allows the specification of single-period systems (multi-period systems can be specified by using a hyper-period expansion).
- Release date and deadline. Each program statement can be associated a release date and a deadline.
- Partitioning. Each program statement can be associated a partition. Partitions can later be used to define allocation requirements.
- Preemptability. On platforms that support preemptive execution, these annotations determine which computations can be pre-empted, and which not.

Heptagon-NFP programs are automatically translated into the input formalism of Lopht, described in (Carle, 2012).

```
open Externc

node main period(0x100000) () returns ()
var
  fs, hs: bool ;
  id, param: int ;
let
  partition(critical) release(0x80000)
                           fs = read_bool_sensor(0x1000) ;
  partition(critical)    hs = read_bool_sensor(0x2000) ;
  partition(noncriti) deadline(0x80000)
                           if hs then
                                 preemptive id = g() ;
                           else
                             var x,y : int; in
                               y = 15 fby x ;
                           id = f1(y) ;
                               x = f2(id) ;
                           end ;
  partition(critical)    if fs then
                             param = 12345 ;
                           else
                             param = id ;
                           end ;
  partition(critical)    () = act(param) ;
tel
```

Figure 5. Example of Heptagon+NFP integration specification with non-functional annotations

3. Fault tolerant vision cluster

The participants in the fault tolerant vision cluster (Recore, NXP, UT, Tue) work on the creation of hardware components and analysis techniques for vision systems but also car-radar systems are considered.

Concerning the hardware components the main focus is on fault tolerant processor design. The first step taken was to determine the vulnerability of the different components in a processor. The next step will be the selective introduction of redundancy to detect and correct errors. Furthermore, a technique based on role-back and re-computation at the software level is considered.

To relax the temporal requirements more robust state estimation and control techniques are under development in the cluster. Concerning state-estimation a particle-filter based approach is explored which can deal well with the non-linear behaviour that is a result of non-periodic sampling. This particle filter based state estimation is relevant for video and radar object detection and tracking. Furthermore, particle filters have some inherent redundancy which makes them potentially interesting candidate to be combined with low-cost fault tolerant HW techniques.

3.1. Analysis of industrial use cases and requirements

The use-case *REC_UC01* is defined by NXP/Recore and concerns fault tolerant processing of a vision/radar systems. Recore is especially interested in fault tolerant hardware design techniques, whereas NXP is more interested in software techniques that improve the robustness of a radar system. This can be achieved by making use of estimators for object tracking. These estimators should be made robust against variation in the interval of time between subsequent samplings as well as hardware errors. Variation in the interval between subsequent samplings can be a result of communication delay and variation in the execution time of the estimator.

3.2. Proposed tools and methods

No overall tool flow is planned because most of the tools do not require interaction with the other tools that are under development. Furthermore, this cluster works on hardware techniques as well.

3.2.1. HAPI dataflow simulator

HAPI is a recently introduced dataflow simulator. The novelty is that it allows to simulated shared resources without introducing any over-approximation or under approximation in the analysis results. Therefore it can be used to falsify analytical analysis techniques like the ones implemented in Xenoclea. Interfacing of Xenoclea and HAPI is desirable to guarantee consistency and simplify analysis of larger test-sets. Furthermore interfacing with the UPPAAL model checker is considered. Evaluation using practical case-studies will potentially result in the future in requests for extensions of HAPI. Furthermore, we currently started to consider extensions of HAPI for the simulation of hybrid systems. In these hybrid systems the continuous time-part of the system should be evaluated besides the discrete time part of the system.

3.2.2. Definition and computation of the architecture vulnerability factor (AVF)

A PhD student at the Tu/e has been working on a simulator for the evaluation of the vulnerability for hardware errors in processors. Unfortunately the PhD student has ended his contract. A new student will be hired to continue this work. However, the main objective is not the development of the simulation tool but the definition of hardware and software techniques that improve the error resilience of processors.

The summary of previous work about vulnerability modelling work is as follows:

The key to generating error-rate estimates is understanding that not all faults in a microarchitectural structure affect the final outcome of a program. As a result, an estimate based only on raw device fault rates will be pessimistic, leading architects to over-design their processor’s fault-handling features. We call the probability that a fault in a processor structure will result in a visible error in the final output of a program that structure's ***Architectural Vulnerability Factor (AVF)***. By definition, AVF is the probability that a fault (for example, induced by a particle strike, manifested as a bit-flip) leads to a user-visible error. It is usually estimated by the average fraction of time that a bit spends in a state that is required for Architecturally Correct Execution (ACE). Any fault in a storage cell that contains one of these bits, which we call ACE bits, will cause a visible error in the final output of a program in the absence of error correction techniques. We call the remaining processor state bits un-ACE bits, as their specific values are unnecessary for architecturally correct execution. A fault that affects only un-ACE bits will not cause an error. AVF can also be defined by Formula 1. AVF can be measured using fault injection, but in our case, we are building a model to measure vulnerability of components, namely, simulation based AVF measurement.

$$\mathrm{AVF} = \frac{\text{num of ACE Bits in structure}}{\text{total num of bits in structure}}$$

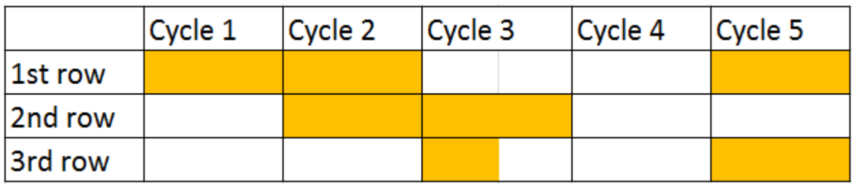
Formula 1

3.2.2.1. Simulation based AVF measurement: Counting Residence Times

Another commonly used method to estimate AVF is by counting the number of ACE bits in the structure every cycle and finding the average. This method is attractive because it is easy to implement in simulations of microarchitectures and thus gives a quick design time estimate of the AVF of a structure. With this method, RTL files, which normally is not available during design phase, are not required. With this method, AVF is defined as below. To make the definition more clear, there is a motivational example in Figure 6. It is a 3-entry 32-bit register files. The bits in yellow is ACE. To calculate the AVF of this 3-entry structure in 5 cycles, it is necessary to calculate the total bits in structure, total execution time and also residency time of all ACE bits in structure. Based on Formula 2, AVF of this example is 43.3%.

$$\mathrm{AVF} = \frac{\sum_{\mathrm{cycle}=1}^{\mathrm{N_{cycle}}} \mathrm{N_{ACE}(cycle)}}{\mathrm{N_{bit}} \times \mathrm{N_{cycle}}}$$

Formula 2

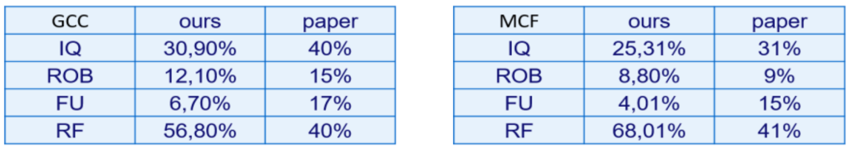


	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
1st row	🟨	🟨			🟨
2nd row		🟨	🟨		
3rd row			🟨⬜		🟨

Figure 6 Motivational example of performance model simulation based AVF estimation

Using simulator to estimate AVF is quicker than error injection based method, however, it requires the designers understand all the cases the errors can be masked. How to classify ACE and un-ACE bits? It is an important question and answered in detail in previous works. In these works, the authors listed most important cases of error masking including hardware masks (idle bits) and software masks (dynamically dead bits & logical masked bits).

We have evaluated AVF using SIM-SODA. All the parameters are the same except that cache is disabled in my experiment which brings more idle and stall stages and probably makes the results different. In SIM-SODA, the authors augment sim-alpha (Alpha version of SimpleScalar) with AVF measurement function. I ran our reproduction simulator with GCC and MCF of SPEC2000INT and compare AVF results. The results are listed below.



GCC	ours	paper
IQ	30,90%	40%
ROB	12,10%	15%
FU	6,70%	17%
RF	56,80%	40%

MCF	ours	paper
IQ	25,31%	31%
ROB	8,80%	9%
FU	4,01%	15%
RF	68,01%	41%

Figure 7 AVF result comparison between SIM-SODA and our reproduction with GCC and MCF testbench.

With the simulator-based method, the target application is executed in a simulator and the ACE bit residencies are counted. The AVF results of simulators indicate the overall reliability of the target application when it is run in the target hardware. However, the results cannot provide the reliability information of the target hardware separately or the target software separately. Thus, researchers are trying to divide AVF into hardware and software parts. In the next section, a methodology to divide AVF into Hardware VF and Program VF, which is proposed by (Sridharan and Kaeli, 2010) is introduced.

3.2.2.2. AVF = PVF*HVF

Although AVF analysis can yield an understanding of the reliability behavior of a hardware structure, no corresponding method has yet been developed to quantify the vulnerability of a program to hardware faults. Such a method would allow researchers to better understand the link between program code and reliability, and could enable the development of reliability techniques at a compiler or even programming language level. Sridharan and Kaeli (Sridharan and Kaeli, 2008) introduced the Program Vulnerability Factor (PVF) to quantify the portion of AVF that is attributable to a user program. This allows a software designer to measure the microarchitecture-independent vulnerability of a program during its design phase. PVF work does not, however, provide a method to quantify the non-PVF components of AVF, nor does it provide a method to re-compute AVF from PVF. In (Sridharan and Kaeli, 2010), the same authors introduced the Hardware Vulnerability Factor (HVF) to address these limitations. HVF quantifies the hardware portion of AVF, independent of program-level masking effects. AVF can then be calculated as the product of HVF and PVF. Computing HVF has three concrete benefits. First, using HVF analysis (in conjunction with AVF analysis) provides insight to hardware designers beyond that gained by AVF analysis alone. Second, separating AVF analysis into HVF and PVF steps can accelerate the AVF measurement process with a 2x reduction in simulation time with no loss of accuracy. Finally, runtime monitoring of HVF enables runtime estimation of AVF by combining HVF measurements with compile-time PVF estimates.

A system vulnerability stack to separate AVF into hardware and software components is shown in Figure 8. If a bit flip on a latch or a logic gate is latched by the following latch, then it is transferred to the functional interface and visible for Microarchitecture layer and is called a soft error. If a SE in Microarchitecture can reach ISA interface, namely, if an SE finally change the instructions of the target program, it is visible for program layer. To make it clear, two motivational example is discussed below. (1) if a fault is latched and stored in 1st entry of RF at cycle 5 which is not occupied until cycle 11, this fault reaches functional interface and visible for microarchitecture layer, however, cannot reach ISA interface and invisible for program layer. Thus, this soft error is masked in microarchitecture layer. (2) if a fault is latched and stored in 3rd entry of RF which is occupied and will be accessed afterwards as an operand of multiplication operation, this soft error is visible for both microarchitecture and program layers, as it changes the ISA by changing the operand value. Assuming the other multiplication operand is 0, this soft error is masked in program layer and invisible at the program output port. The masking effect of the 1st example contributes in HVF measurement, while the 2nd one is counted in PVF measurement. Occupancy can be used to approximate HVF. The correlation coefficient between occupancy and HVF is greater than 0.97 across all benchmarks and provide an upper bound of HVF and AVF, after multiplication with PVF. HVF and PVF is defined as Formula 3 & 4. In he two cited papers, there are 2 motivational examples to explain how to calculate HVF and PVF and how to calculate AVF with them. As occupancy measurement can be very quick and program profiling which is independent of hardware can be performed in compile time, the division of AVF into hardware and software components can not only guide designers to improve the reliabilities of hardware and software separately and wisely, but also reduce the AVF measurement time and thus, make accurate run-time measurement possible.

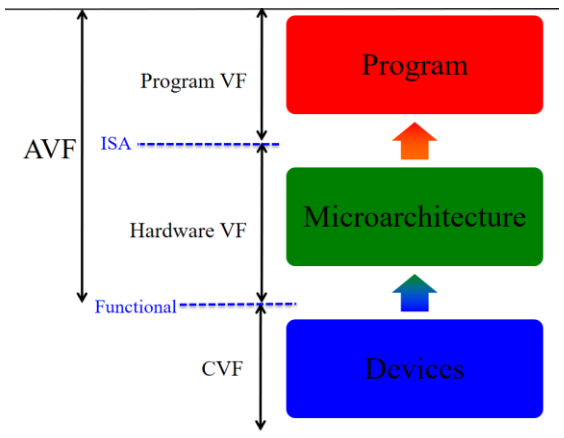


Figure 8 A system vulnerability stack to separate AVF into hardware and software components

$$\mathrm{HVF} = \frac{\sum_{n=0}^{N} unmasked\ faults\ in\ target\ hardware\ H\ at\ cycle\ n}{the\ size\ of\ H * cycle\ number}$$

Formula 3

$$\mathrm{PVF} = \frac{\sum_{i=0}^{I} unmasked\ faults\ in\ resource\ R\ at\ instruction\ i}{the\ size\ of\ R\ *\ instruction\ number\ I}$$

Formula 4

3.2.2.3. Approximate-compute oriented PVF

Approximate computation is very popular recently. For some applications, errors that are small enough can be treated negligible, namely invisible. In this case, a decrease in accuracy in an acceptable range can help improve the performance and save energy. The sizes of acceptable ranges vary depending on application requirements. As AVF is the probability that a fault can be transferred to a visible error, the errors which are small enough to be neglected should also be treated as masked error. We call it natural masking effect happening in the natural layer which is on top of program layer as shown in Figure 9. Natural masking effect should be counted during PVF measurement. In previous works, researchers only focus on the Soft Error Rate (SER), but neglected the seriousness of the errors, namely the value of the errors. What we are going to propose is a new term called Approximate-processing-oriented AVF (AAVF) which also takes the error seriousness and natural masking effect into consideration.

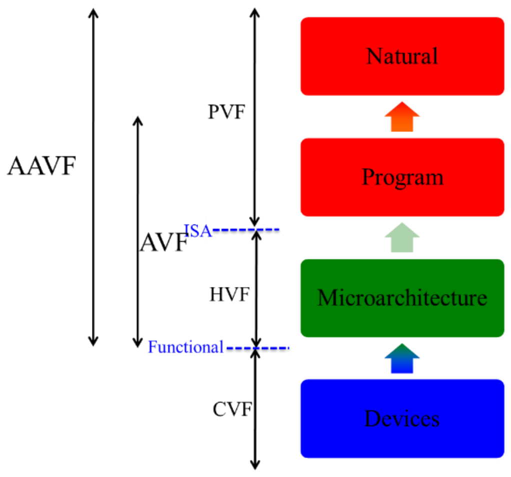
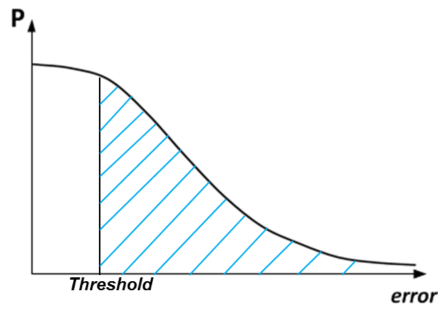


Figure 9 Approximate-compute oriented system vulnerability stack

a) The relationship between approximate processing and fault tolerance
 In this subsection, the relationship between approximate processing and fault tolerance is discussed. As shown in the graph 1, the threshold is the value of largest negligible error and the area in shadow is the

approximate-processing-oriented SER. Only the errors which are larger than the threshold are visible SE. The classical SER is defined as Formula 5, with the threshold as 0. Appr.comp.SER is defined in Formula 6. The average visible error is defined in Formula 7.



Graph 1 The relationship between approximate processing and fault tolerance

$$SER = \int_0^\infty P(error) * d_{error}$$

Formula 5

$$appr.comp.SER = \int_{Threshold}^{\infty} P(error) * d_{error}$$

Formula 6

$$Average.Error = \int_{Threshold}^{\infty} P(error) * error * d_{error}$$

Formula 7

b) AAVF: Approximate-compute oriented PVF (APVF) * HVF
Firstly, we need to figure out in which cases errors can be masked in natural layer. The answer is: ***(1) Low number of errors (2) Low concentration of errors (3) Small value of errors.*** The thresholds of these three parameters varies depending on application requirements. A frame with errors which are below these thresholds can be treated as correct frame.
Secondly, we need to combine natural masking effect with other masking effects in program layer and propose a new methodology to measure APVF.
In (Shafique *et al.*, 2010) , the authors propose an application to measure classical PVF. Instructions are taken as nodes in a tree, namely an application and outputs are taken as leaves. The authors traverse the tree and compute PVFs of all nodes. The algorithm 1 & 2 in this paper are used in their PVF measurement application. We can augment this application with natural masking effect. Based on the threshold of the largest negligible error, the tree can be traversed again and all PVFs are updated. For example, assuming 10% error is negligible, the predecessor of one leaf is an addition instruction, the error of the last 12 bits of the addition operands can be masked by the natural masking effect. In turn, the PVF of the predecessor of this addition node can also be updated in the same way. Considering low concentration of errors and small number of errors can blur the errors and decrease the error serious, the updated PVFs need deratings. After error number derating and error distribution derating to the updated PVF (e.g. Updated_PVF * Threshhold_of_number / Threshold_of_concentration), results in the APVF.

3.2.3. Approximate DMR-based Reliable Vector Processor

In this section, a reliable vector processor is proposed which applies approximate Double Modular Redundancy (DMR) technology and can tolerate not only transient errors but also permanent errors. Approximate DMR is similar to DMR. The only difference is approximate DMR take a simplified copy as redundancy, which helps save costs. DMR and TMR are the most popular fault tolerant technologies in microarchitecture level. For error detection, DMR is better than TMR, as it saves energy and takes smaller area. For permanent error recovery, DMR and TMR are the same, as the broken components are always needed to be replaced. However, considering transient error recovery, TMR is better, as with TMR, the correct value can be figured out by the voter and roll-forward recovery can be applied which saves time. While with TMR, rollback recovery is required, which brings overhead in runtime. However, what if we can reduce the SER? Then, the overhead of rollback recovery is reduced and TMR becomes an optimal selection. To reduce the SER, we can do fault prevention by hardening the most vulnerable components of the vector processor **with the guidance of AVF**.

1) Transient fault
   a) Detection: DMR
   b) Recovery: Rollback
2) Permanent fault
   a) Detection: DMR
   b) Recovery: Replace with the spare PE and spare Date Memory as shown in Figure 10. Normally the spare PE is not linked into the circular PE network. When one PE is broken, the broken PE is left out, its two neighbor PEs are linked together and the spare PE joins the network and is linked with the first and last PE. Memory banks recovery is in the same way.

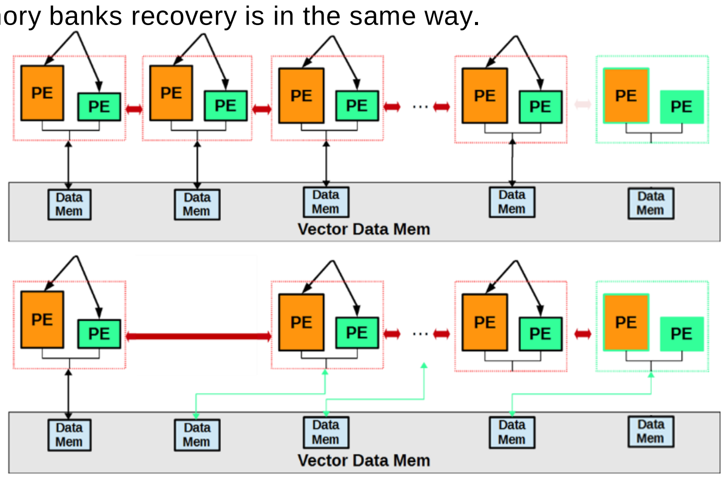


Figure 10 Reliable vector processor

3.2.4. Particle filter based state estimation algorithm

A PhD student of the UT is looking into error resilient object detection and tracking. As a first step, he is considering the problem of mitigating the effects of sampling uncertainty. This has resulted in the definition of a new particle filter based state estimation algorithm. This work will likely not result in a new tool but in new robust algorithms that are implemented on a fault tolerant vision platform and evaluated using the HAPI simulator. The hardware of such a platform is under development. NXP looks into the use of similar estimation techniques but then in the radar context.

3.2.5. Fault-Tolerant Network-on-Chip architecture

Recore Systems' work concentrates on fault-tolerant hardware and software techniques in the on-chip interconnect of multi-processor systems-on-chip (SoC) architectures. In these SoCs there are lots of processing elements which communicate with each other. For the communication between these elements, various interconnect architectures like simple bus, hierarchical bus, ring based bus, etc. have been in use.

However, as the number of cores increases, traditional bus based architectures face problems like bus contention, arbitration, etc., which can be overcome with a Network-on-Chip (NoC) solution. Due to its flexible, computer network like architecture, a NoC can support concurrent communicatio n between pairs of nodes in the network, and adapt to changing data transmission requirements.

The baseline architecture of a Fault-Tolerant Network-on-Chip architecture is defined. The NoC architecture design focuses mainly on the lower layers of the NoC, i.e. the data-link layer and the network layer. The NoC incorporates fault mitigation techniques obtained from an exhaustive literature survey.

4. Safe vehicle control cluster

The participants in the safe vehicle control cluster (Verum, UT, (VDL), (TNO)) work on tools for the creation of software components and temporal analysis techniques for advanced driver assistance functionality in cars and coaches. The main focus is on distributed systems.

Concerning analysis and synthesis of the functional behaviour in software the focus is on the use of the Dezyne tool of Verum. This tool allows the specification of state based event driven or concurrent components; these can be verified for hidden defects, such as deadlocks, after which code can be generated that is guaranteed deadlock free. The main activity has been the identification of a suitable use-case together with VDL which can be realized despite all the restrictions vendors of embedded-compute-units (ECUs) for cars place concerning reprogramming of their units. Furthermore, additional functional verification features are under development for the Dezyne tool.

Concerning temporal analysis techniques, a new refinement theory has been developed and published which allows reordering of data as a result of for example data-parallel computation. Furthermore, the dataflow simulator HAPI has been developed which is able to simulate sharing of resources. Also a more accurate temporal analysis technique has been developed and published. This techniques enables accurate modelling of an application using the more expressive CSDF model. The analysis method has been implemented in an analysis tool which is called Xenoclea.

Furthermore, TNO has defined a use-case of a distributed control system in coach/truck which makes use of car-2-car communication using WLAN 802.11p. This use-case will be used for the valuation of the developed temporal analysis techniques and the dataflow simulator. TNO is interested to analyse worst-case temporal behaviour as well as the probabilistic temporal behaviour of the system using the developed tools. This temporal behaviour is very important in the considered system because it affects the stability of the control system that will be used for the controlling the distance in a platoon of trucks. To relax the temporal requirements more robust state estimation and control techniques are under development in the cluster.

4.1. Analysis of industrial use cases and requirements

Use-case *TNO_UC01* is defined by TNO/VDL and its objective is to derive the worst-case and probabilistic temporal behaviour of a network of ECUs in truck platooning setup which is under development by TNO. One of the key challenges is the definition of a formal analysis model that captures the mix of time-triggered and event-driven processing. This model should unambiguously describe the temporal behaviour of the system despite shared resources (e.g. the communication network). Using the model it should be possible to compute the worst-case behaviour as well as derive the probability density function which describes the end-to-end latency.

Use-case 2 is defined by VDL/Verum and concerns the definition of monitors and/or controllers for ECUs in CAN networks. Monitors should raise alarms when communication with the environment does not adhere to defined protocol/behaviour. Controllers add active communication to this, and as such influence the network. The current proposal is the usage of a separate ECU for monitoring and control because after a long investigation and consultation of the ECU vendor, no suitable way was found to program the ECUs in C that are currently in use by VDL.

4.2. Proposed tools and methods

In the safe vehicle control cluster there are no plans to define one overall tool flow that makes use of developed tools. The main reason is that most of the tools that are under development address different aspects of a design and do not benefit from results of other tools. One exception is probably the work on Xenoclea and HAPI where the simulator can be (and is) used to falsify analysis results. In the future it is likely that exchange of data with a model checker like Uppaal will be added.

4.2.1.1. Xenoclea

Xenoclea is a temporal analysis tool developed by the UT/NXP. This tool is suitable for the analysis of task graph running on multiprocessor systems in which the processors are shared. Different schedulers are support including fixed priority preemptive. The task are allowed to communicate using finite FIFO buffers resulting in a cyclic dataflow analysis model. It is desirable that the accuracy of the tool is further extended as well as more expressive models should be supported to allow the analysis of a larger class of systems.

4.2.1.2. Dezyne

Dezyne is an event-driven modelling tool developed by Verum. The typical application of Dezyne in the automotive domain requires Dezyne models to be executed on an ECU in a CAN bus network. The CAN standard requires all nodes on the CAN network to be synchronized to sample every bit on the CAN network at the same time. Dezyne models with their discrete event-driven approach have to interface with the CAN oriented world. Such interfaces will developed.

Furthermore we investigate the possibilities to extend Dezyne with a formalism for functional verification, where the user can specify certain properties of the observable behaviour of a design. A specification language for such properties is needed, and tooling to check the design against these properties.

5. Consumer electronics cluster

Smart devices and components has been widely employed in recent years. New and previously unforeseen application areas for smart components emerge frequently. Although application areas are different, efficiency and scalability is the common and premium requirement of the smart components. Hence, the underlying embedded controllers mostly rely on multi-core architectures or allow concurrent programs.

To achieve better performance, programs tend to adopt techniques that increase the level of concurrency. However, highly concurrent programs enable enormous number of possible interleavings and different executions which make reasoning about the correctness of the program difficult and make the programs error-prone. Some concurrency bugs occur in subtle and/or rare conditions and it is difficult to detect them by standard testing methods. Hence, formal treatment (verification or validation) is crucial for mission-critical or safety critical components of the programs.

However, verifying a concurrent program by doing formal proofs and static analysis is not an easy task. The first difficulty arises because of the gap between the theory and practice. The widely accepted model for reasoning about the behaviors and semantics of concurrent programs is Sequential Consistency (SC) defined by Lamport (Lamport, 1979). In SC, all operations of an execution appear to be in some sequential order and operations from the same execution unit appear in this execution in the order specified by the program. SC is an elegant and powerful enough to describe concurrent programs but it is not realistic. Most modern hardware architectures (Intel x86, PowerPC, Arm …), including GPUs and programming language specifications (C, C++ 2011) allow "relaxations" which enable programs to produce more behaviors than SC ones.

The relaxations in hardware and programming language specifications are necessary for performance reasons. Relaxations are diverse and there are still significant examples of programming languages and hardware architectures that lack the formal semantic model. In the recent literature, relaxations are described by allowing statement reorderings in SC, imposing some relations among program actions as constraints on the executions or adding machines and data structures to SC programs for an operational description.

Reasoning techniques and proof tools are well-developed for programs running on SC semantics. The well-known Owicki-Gries (OG) reasoning (Gries, 1976) can be utilized for checking correctness of local assertions and Lipton's theory of reduction (Lipton, 1975) combined with abstraction techniques can be used for performing refinement proofs. However, there are no practically applicable tool or method for verifying programs running on relaxed semantics.

Our aim in the scope of this project is to develop a proof system that enables refinement and linearizability proofs for programs running on weak semantics. Core of our method is to start from the most abstract specification of the program and reach to a concrete program in which all of the atomic actions of the program correspond to actual assembly level instructions, via a sequence of refinement proofs. We aim to leverage the power of already existing proof methods for SC for performing the refinement proofs. We transform original program *P* to another program *P'* by explicitly adding operational semantics of the relaxed memory model such that every relaxed execution of *P* is also an SC execution of *P'*. Hence, proving linearizability or refinement of the transformed program using the SC proof rules is sufficient for showing that the original relaxed programs refinement or linearizability.

We pick x86-TSO as our sample relaxed model since it describes the memory model of x86 Intel machines and a useful fragment of the C, C++ 2011 specification. Operationally, TSO can be described on SC as follows: each execution unit (thread or core) keeps a local stack for write (update) operations. Different from SC, write operations on global variables are not directly reflected to shared memory, but they are pushed to local stacks. These stacks can nondeterministically interfere into execution and perform pops. In addition, read operations on global variables first try to read from the local stack to check if there is a recent update on this variable that has not been reflected to shared memory yet. This model is simple enough to allow explicit program transformation, yet powerful enough to yield non-SC executions by delaying write operations.

5.1. Benchmark Data Structures and Synchronization Primitives

Static analysis methods are robust and used for formal proofs about programs. However, performing proofs is a tedious task that mostly needs human effort and intervention since most of the problems considered are undecidable by their nature. Hence, they are used for verifying mission-critical or safety critical components of the programs. We pick important data structures and synchronization primitives from the literature to show applicability of our approach. The examples we pick are widely employed in real industrial applications and their correctness is crucial for correct functioning of the programs. In this section, we will present, analyse and show importance of these benchmark examples.

Spinlock: Locks are important synchronization primitives that are used to protect critical sections and provide synchronization among threads. Spinlock is a widely used CAS based locking mechanism. For this example we aim to show that spinlock adheres to its atomic specification under relaxed semantics.

Double Checked Locking: An optimized concurrent software engineering paradigm for initializing objects or assigning values to objects based on locking. We developed a simple procedure that performs assignment using double checked locking, utilizing a spinlock. We aim to verify that the method adheres to its atomic specification.

Send / Receive Example: A simple synchronization mechanism that allows execution units to communicate through a shared flag variable. Using this variable in a careful way, an execution unit prevents another execution unit to read shared variables before it reflects all of its changes. We provide sample sender and receiver methods in this example and aim to show that values read by the receiver method is exactly the ones updated by the sender by providing local assertions.

Chase-Lev Deque (CLDeq) (Lev, 2005)**:** A double-ended queue (deque in short) is a concurrent data structure. A special thread called worker thread could perform *put* and *take* operations could insert and remove elements from the tail of the deque, respectively. Concurrently, other threads (called stealer threads) could perform *steal* operations that can remove elements from the head of the deque. Deque is a key data structure used in task-based concurrent platforms and its correctness is crucial for distribution of tasks. CLDeq is a high-performance, complicated deque implementation. It has been shown that non-SC behaviorus of CLDeq exist under TSO semantics. Hence, the usual deque specifications may not be satisfied by TSO executions of CLDeq. We aim to find tight enough (yet more relaxed than the original deque specs) atomic specifications and show that CLDeq refines these specifications.

5.2. Proposed tool flow and desired tool extensions

5.2.1. Global toolflow organization

BoogiePL (Mike Barnett, 2005) is an intermediate verification language that is used for describing proof obligations from various domains and Boogie is the tool developed by Microsoft Research to translate proof obligations in Boogie PL to SMT formulae and check their satisfiability using SMT solvers. CIVL (Qadeer, Tasiran, & Hawblitzel, 2015) is an extension to Boogie that uses a dialect of BoogiePL developed by Microsoft Research and MSRC lab at KU for reasoning about concurrent programs. It allows a sequence of refinement proofs for SC programs in a layered structure utilizing Lipton's reduction, abstraction techniques and OG reasoning. Our aim is to extend CIVL and Boogie tool set for our purposes.

Proposed method for refinement proofs on relaxed memory models contains the following steps:

- Input to the method is the finest grained concrete program *P* written in CIVL-like language. The language we use both extends and restricts the original CIVL language. Atomic actions of the program and the global variables must obey the restrictions provided in Section 5.3.1. CIVL language is extended to allow TSO memory model specific constructs like barriers.
- Apply program transformation on *P* to obtain equivalent SC program *P*`. Program transformation involves explicit modelling of thread buffers and global shared memory. In addition, the lowest level atomic actions in *P* are replaced with TSO counterparts in *P*` and nondeterministic drains are introduced. The transformed program is totally in the language of CIVL.
- Use CIVL to perform refinement proofs on *P*`. Using the layered proof structure of CIVL, obtain abstract programs by performing refinement proofs until reaching the desired abstract specifications. In addition to existing OG, reduction/abstraction proof techniques, we develop and apply new proof rules special to TSO related actions for simplifying proofs.
- (Optional) Use adapted version of BoogieASM on *P* to obtain executable assembly program for the desired platform. Since BoogieASM is in an experimental stage now and its development directions are

uncertain, we may skip employing BoogieASM and prefer to develop our own translation tool for this purpose in the future stages of the project.

5.2.2. Tool status and needed extensions

CIVL is a tool that can be used for verifying SC concurrent programs. Currently, it does not have any support or proof techniques for the relaxed memory models. Our aim is to extend CIVL/ Boogie toolset so that it becomes convenient for TSO programs and semantics.

5.2.2.1. CIVL \ Concurrent Boogie

There are two crucial extensions needed on CIVL to make it suitable for our method:

- A translator / compiler that will take the relaxed program *P* written in the extended CIVL language described previously and that will produce the SC program *P`* in CIVL language.
- Modifications in the CIVL source code to adopt our TSO specific proof rules. Our TSO proof rules include generalizations of existing mover analysis and reduction rules as well as specific rules tailored for certain TSO actions that may occur in the transformed program *P`*.

5.2.2.2. BoogieASM

BoogieASM is a tool that aims to generate executables from Boogie programs that are correct by construction. However, the tool is still in its infancy. It is not certain that what kind of semantic models will be supported by the tool and how it will be integrated to Boogie – CIVL. Depending on the future developments on BoogieASM, we aim to extend it for generating executables from Boogie programs that are verified through our methodology.

5.3. Interface definitions

5.3.1. Common Interface for CIVL and BoogieASM

In addition to the language allowed by CIVL, we require following restrictions / extensions on the input program *P*:

- For simplicity, we initially allow two kind of base types for global variables inside the programs: integers and pointers. Integers are already a base type in CIVL. If pointers will be used by the programs, there should be a definition of Boogie type with the exact name x*Pointer:int* in the program where *x* is another type name. If the defined type is just *Pointer*, we interpret it as the integer pointer. We allow composite types (records in Boogie) that may consist of base types or other composite types.
- Thread identifiers are assumed to be of type *Tid*. They must be passed as linear arguments to the methods.
- Dynamic memory allocation from explicitly defined memory is possible. To achieve this, the lowest level method *alloc* must be defined inside the program and it should be called for dynamic memory allocation inside the other methods.
- For reading from a global variable *x*, the programmer must define and use the lowest level atomic procedure *readX()*.
- For writing to a global variable *x*, the programmer must define and use the lowest level atomic procedure *writeX()*.
- For performing a compare-and-swap action, programmer must define and use the lowest level atomic procedure *CAS()*.
- The lowest level method name *barrier()* must be defined and used inside the methods for putting TSO fences inside the methods.

References

ASSUME Project Team. (2015). *Full Project Proposal Annex.* ASSUME.

CHAFEA, E. C. (2012, January 30). *Managing projects, Elaborating a Dissemination Plan*. Retrieved from http://ec.europa.eu/chafea/management/Fact_sheet_2010_06.html

Gries, S. O. (1976). An axiomatic proof technique for parallel programs. *Acta Informatica, 6(4)*, 319-340.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on, 100(9)*, 690-691.

Lev, D. C. (2005). Dynamic circular work-stealing deque. *In Proceedings of the seventeenth annual ACM Symposium on Parallelism in algorithms and architectures*, 21-28.

Lipton, R. J. (1975). Reduction: A method of proving properties of parallel programs. *Communications of the ACM, 18(12)*, 717-721.

Mike Barnett, e. a. (2005). A modular reusable verifier for object-oriented programs. *International Symposium on Formal Methods for Components and Objects*, 364-387.

Mutluergil, S. O., & Tasiran, S. (2016). A Mechanized Refinement Proof of the Chase-Lev Deque Using a Proof System. *International Conference on Networked Systems* (pp. 280-294). Springer International Publishing.

Qadeer, S., Tasiran, S., & Hawblitzel, C. (2015). Automated and modular refinement reasoning for concurrent programs. *International Conference on Computer Aided Verification* (pp. 449-465). Springer International Publishing.

T. Carle. Efficient compilation of embedded control specifications with complex functional and non-functional properties. PhD thesis. 2014. https://tel.archives-ouvertes.fr/tel-01088786/document

Koek, P. et. al. (2016, May 23-25) CSDFa: a model for exploiting the trade-off between data and pipeline parallelism. In: Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems

Kurtin, P. et. al. (2016, May 23-25) HAPI: An Event-Driven Simulator for Real-Time Multiprocessor Systems. In: Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems

Sridharan, V., Kaeli, D. Quantifying software vulnerability. Proceedings of the 2008 workshop on Radiation effects and fault tolerance in nanometer technologies, 2008.

Sridharan, V., Kaeli, D. Using hardware vulnerability factors to enhance AVF analysis. Proceedings of the 37th annual international symposium on Computer architecture (ISCA'10).

Shafique, M. and Rehman, S. and Aceituno, P. V. and Henkel, J. Exploiting Program-level Masking and Error Propagation for Constrained Reliability Optimization. Proceedings DAC 2010.

Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., Pouzet, M. - n-Synchronous Kahn Networks. In POPL 2006