D3.1
Modular reasoning for system validation and verification

Project number:	731453
Project acronym:	VESSEDIA
Project title:	Verification engineering of safety and security critical dynamic industrial applications
Start date of the project:	1st January, 2017
Duration:	36 months
Programme:	H2020-DS-2016-2017

Deliverable type:	Report
Deliverable reference number:	DS-01-731453 / D3.1 / 2.0
Work package contributing to the deliverable:	WP3
Due date:	June 2018 - M18
Actual submission date:	15th of October, 2018

Responsible organisation:	CEA
Editor:	Virgile Prevosto
Dissemination level:	PU
Revision:	2.0

Abstract:	This document describes the combination between high-level system reasoning with Diversity and low-level code verification with Frama-C
Keywords:	Diversity, Frama-C, symbolic execution, relational property, deductive verification

The project VESSEDIA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453.

Editor
Virgile Prevosto(CEA)

Contributors
Imen Boudhiba (CEA),
Boutheina Bannour (CEA),
Christophe Gaston (CEA)

Executive Summary

A major issue in assessing the security of a whole system is the lack of correspondence between high-level properties expressed at model level and lower-level verification tasks that need to be done on the implementation itself. The goal of Task 3.1 in the VESSEDIA project is to propose a solution bridging this gap for subsystems components that are implemented as software coded in C. More specifically, as described in this document, we propose to combine the analysis done at the model level by the Diversity tool, over UML sequence diagrams produced e.g. by Papyrus, with the verification that the implementation follows the execution paths identified by Diversity, through the use of a set of Frama-C plug-ins. The core of this method relies on the development made within Diversity to generate *relational properties*, a specific kind of formal properties over C programs, that a newly developed Frama-C plug-in, RPP, can understand and translates into more classical specifications that standard analysis plug-ins, namely WP, can verify. Finally, in this document we also present the first experiments that have been done with this toolchain over the 6LowPAN use case.

Contents

List of Figures

List of Tables

[1] `https://projects.eclipse.org/proposals/eclipse-formal-modeling-project`

Chapter 1

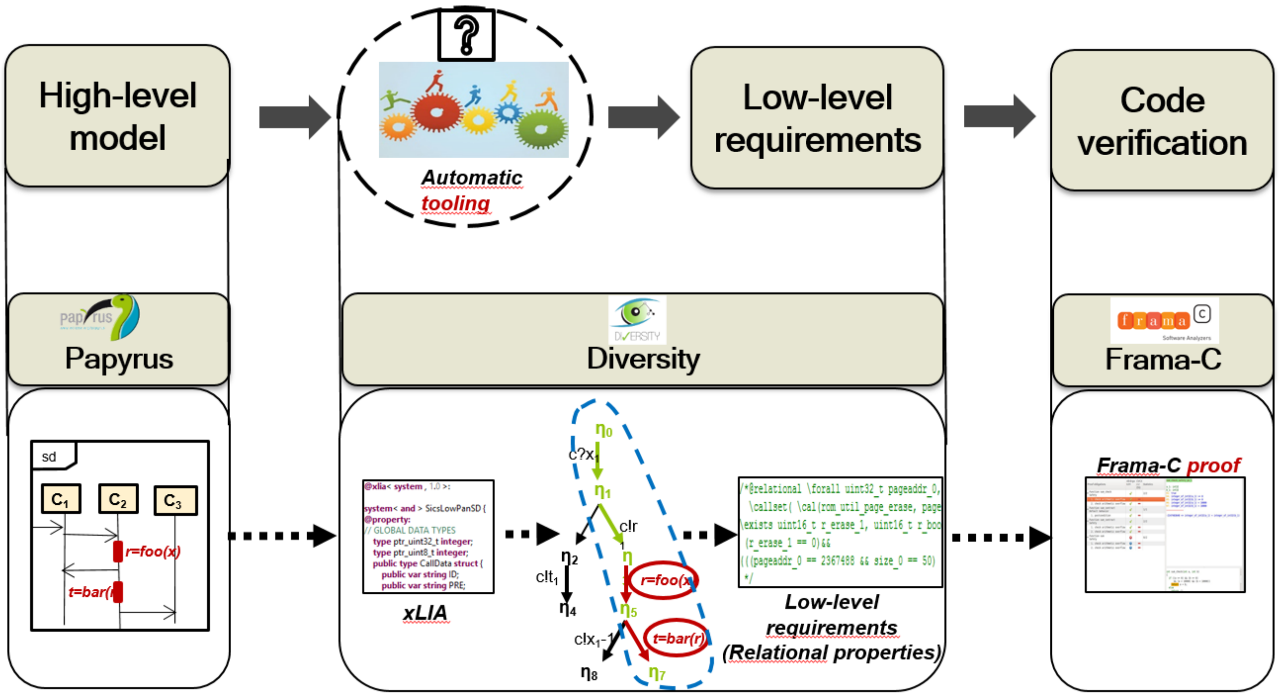
Introduction

This report describes work done inside Task 3.1 of the VESSEDIA project on modular reasoning for system validation and verification. Generally speaking, the goal of this task is to propose solutions to help users derive properties to be verified at code level from high-level properties expressed at the system level. Indeed, since systems are very often heterogeneous, it is not reasonable to try to find a unique verification technique for the whole system: some parts might be black boxes that can only be assessed through testing techniques, while some others might be software whose source code is known and which might be analysed using static analysis tools such as the one offered in the VESSEDIA project. It is therefore essential to boil down the problem of verifying systems to verifying, with different techniques if needed, its sub-systems. This question has been addressed in "design by contracts" approaches in a bottom-up manner, basing reasoning and verification technologies on compositional results whose goal is to ensure that system requirements can be deduced from properties of basic sub-systems. However, it is often difficult to identify the basic properties that should hold at sub-system level in order to have a given property hold at system level. Other approaches consider the question from a top-down perspective. Starting from system requirements and architecture, it is possible to derive local properties that should hold at sub-system level to ensure global correctness.

In the context of this task, we have focused on the use of the Diversity tool [13, 2], which forms the basis of the Eclipse Formal Modeling Project (E-FMP, [14]) for system-level reasoning, and of the Frama-C toolset [17] for code-level verification of properties inferred from Diversity. The proposed toolchain is summarized in figure 1.1.

Basically, we start from an UML model, e.g. designed with the Papyrus tool [15] using the meta-model developed as part of T1.2 and T1.3. This model, expressed as a sequence diagram, contains a certain number of function calls, as well as constraints expressed over the variables of the models. Diversity will then explore the sequence diagram and provide an execution tree, with path constraints indicating which conditions must be fulfilled for a given path to be taken. Now, in our context, some of the variables of the model are in fact the result of an internal computation (i.e. a function call). The generation of code level properties from the path constraints of Diversity is based on the idea that the implementation of function f must return results that are compatible with the path constraints in which a call to f is involved. Early work in this direction [10] has shown that when there is a single call involved in the path, it is possible to generate a partial function contracts in the ACSL specification language [5], that is at the core of the Frama-C platform.

However, when there are several function calls on the same path, contracts are not sufficient anymore. In particular, they fail to capture possible relations between the results re-



```
@xlia< system , 1.0 >:

system< and > SicsLowPanSD {
@property:
// GLOBAL DATA TYPES
  type ptr_uint32_t integer;
  type ptr_uint8_t integer;
  public type CallData struct {
    public var string ID;
    public var string PRE;
```

```
/*@relational \forall uint32_t pageaddr_0,
  \callset( \cal(rom_util_page_erase, page
\exists uint16_t r_erase_1, uint16_t r_boo
 (r_erase_1 == 0)&&
(((pageaddr_0 == 2367488 && size_0 == 50)
 */
```

Figure 1.1: Toolchain for verifying code-level properties assessing system-level requirements

turned by different calls. Fortunately, a Frama-C plug-in, RPP [8], is currently developed to allow expressing properties over several function calls. We have thus investigated the possibility of generating RPP properties instead of plain ACSL contracts.

The primary target of the experiments made in this direction has been extracted from the 6LowPan use-case of VESSEDIA. The chosen property focuses on the main functional requirements of the firmware update: basically, we want to ensure that the update of a node is considered successful if and only if all blocks of the new firmware are received, their checksums are verified, they are written in the corresponding slot of the flash memory, and the checksums of the memory blocks are also correct. A model of the node in UML has been developed and various path constraints have been generated by Diversity, giving rise to RPP properties. However, the corresponding 6LowPan code used many assembly parts and/or external functions, making it unsuitable to analyse directly. Analysis has thus been done on an equivalent pure C implementation. Equivalence with the actual code has been manually assessed with the help of LSC lab at CEA Tech List, who develops the firmware update code in the first place.

The report is structured as follows. First, we briefly present the Frama-C platform (chapter 2) and the Diversity tool (chapter 3). Then, chapter 4 describes how function calls are taken into account at the model level, and what kind of code-level properties can be derived from such a model. The RPP plugin of Frama-C is detailed in chapter 5. Finally, chapter 6 explains how the whole toolchain has been put into practice into the examples derived from the 6LowPan use-case.

Chapter 2

Frama-C toolbox

2.1 Frama-C kernel

Frama-C [17] is a platform dedicated to the analysis of C programs. It features a modular design, centered around a kernel that takes care of parsing C files and maintaining the resulting internal representation as an Abstract Syntax Tree (AST), as well as collecting the information emitted by the various analyzers in order to save it on disk, or conversely load the results of previous analyses. On top of this kernel, analyzers are built as plug-ins.

In addition, it is possible to provide, together with the code, formal specifications in the form of ACSL [5] annotations. The core components of ACSL are contracts and assertions. A contract let a user specify what a function requires from its caller (the pre-condition, a constraint about the program state when the execution of the function begins) and what it ensures in return (the post-condition, a formula characterizing the state of the program at the point where the function returns successfully). ACSL's assertions, like C's **assert**, state that a property should hold at a given program point. The language in which an ACSL **assert** can be expressed is however much richer than C.

Figure 2.1 gives a very simple example of ACSL annotations, with a contract for a function `swap` that takes as argument two valid pointers (pointers that can be safely dereferenced), and ensures that it swaps their contents, leaving the rest of the memory intact (this is the sense of the **assigns** clause).

```
/*@ requires validity: \valid(a) && \valid(b);
    assigns *a, *b \from *a, *b;
    ensures exchange: *a == \at(*b, Pre) && *b == \at(*a, Pre);
 */
 void swap(int* a, int* b) {
   int tmp = *a;
   *a = *b;
   /*@ assert tmp == \at(*a, Pre); */
   *b = tmp;
 }
```

Figure 2.1: Example of ACSL specification

Figure 2.2 gives a brief overview of the main plug-ins that are maintained at CEA Tech List. The ones that are most relevant for the activities of this task are marked in yellow and

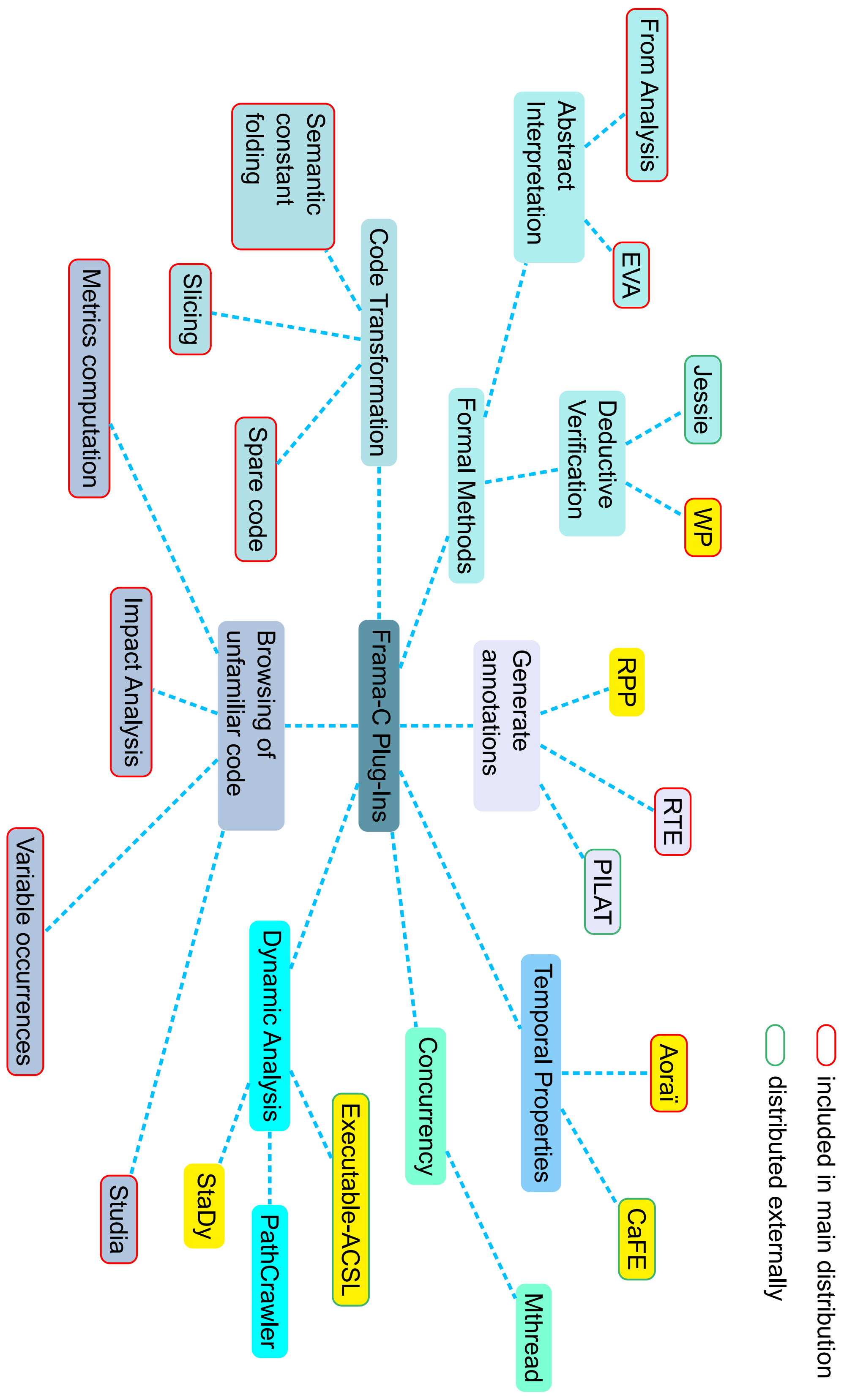


Figure 2.2: Overview of Frama-C plug-ins at CEA List

will be described in more details in the following chapters, but, from a very high-level point of view, these plugins can be grouped in a few broad categories as follows.

Semantic analysis: the most important plug-ins of the platform, EVA and WP, provide information about the semantics of the program under analysis, implementing respectively abstract interpretation and deductive verification techniques.

Dynamic analysis: although Frama-C is mainly dedicated to static analysis, some plug-ins, notably E-ACSL and PathCrawler, target concrete executions over instrumented code

Property specification: Function contracts and assertions offered by ACSL are not always the most convenient way to express properties that the code is supposed to verify. Some plug-ins propose other means to write these properties. This includes in particular temporal logic (LTL or CaRet) formulas with the Aoraï and CaFE plug-ins, RTE (generation of an assertion at each program point where an undefined behavior would appear if the assertion does not hold), or, as mentioned before, RPP.

Program transformation: it is possible to use the results of the analyses to perform various kind of transformations (constant propagation, dead-code elimination, ...)

Code browsing: Some plugins provide facilities for browsing through the code, especially for understanding the root causes of problems reported by the analyses.

2.2 Properties specification

As mentioned previously, the first targets for generating code-level properties from Diversity scenarios were pure ACSL contracts. This setting was not very satisfying for taking into account execution paths with multiple function calls. While it would have been possible to extend Diversity to directly generate a set of contracts also in this case, a better solution was to investigate whether some of the plug-ins dedicated to property specifications would offer a convenient target for Diversity. More specifically, three plug-ins were potential candidates: Aoraï [24], CaFE [12], and RPP [8].

2.2.1 Aoraï automata

Aoraï is historically the first plug-in of Frama-C that allowed users to express properties in another form than ACSL contracts. More precisely, the initial input language offered by Aoraï is Linear Temporal Logic (LTL [22]). LTL formulas allow specifying properties over a potentially infinite sequence of events, using *temporal operators*, in addition to boolean connectors:

- **X**p, indicating that p must hold at the *next* event;
- p**U**q, indicating that p must hold *until* q becomes true (which it will eventually);
- **F**p, indicating that p will hold at some point in the future, that can be defined as $true$ **U** p
- **G**p, indicating that p will always hold, that can be defined as $\neg($**F**$\neg p)$

In the context of Aoraï, the events that are tracked are function calls and return. For example, the following formula expresses the fact that it is not possible to access sensitive information before a proper authentication check:

```
 (!CALL(get_private_data))
     _U_ (RETURN(authentication) && authorized == 1)
```

Aoraï relies on an external tool to generate a Büchi automaton from a LTL formula. It is also possible to directly describe the property of interest as an automaton, whose transitions are guarded by the events of interest and conditions over the variables of the programs. The automaton has one initial state, in which it is at the beginning of the execution, and some accepting states, in which it must be at the end of the execution. An automaton corresponding to the formula above would be for instance:

```
%init S0;
%accept Sf;
%deterministic;

S0: CALL(get_private_data) -> fail
  | RETURN(authentication) && authorized == 1 -> Sf
  | other -> S0;

Sf: -> Sf;

fail: -> fail;
```

Given an automaton and an implementation, Aoraï will instrument the code to reflect the transitions taken by the automaton and generate ACSL contracts to ensure that the program ends in an accepting state. It is then up to other plug-ins to verify these ACSL annotations.

While there are some obvious similarities between Aoraï's automata and sequence diagrams used by Diversity, some technical difficulties prevent targeting these automata with Diversity directly. First, the diagrams only provide an abstraction of the call graph of the program, while Aoraï traces all call and return events and does not provide any mechanism to ignore some functions. Second, and more importantly, the constraints put on the model might rely on external variables that do not exist in the implementation. There is no possibility to introduce such variables in the guards of the automaton or in the LTL formula. In particular, this makes it impossible to specify a relation between the arguments and return values of two distinct calls, for instance to enforce that a `FILE f` obtained from `fopen` must be closed through a call to `fclose` with `f` as argument.

2.2.2 CaFE: CaRet Frama-C's Extension

One of the issues of using pure LTL for specifying program executions is that there are no built-in operators to speak directly of the call stack. In order to overcome that, the CaRet language [1] distinguishes between call, return and internal events in the execution trace, and proposes three sets of temporal operators:

- *general* operators (**X@N**) are the normal LTL operators;
- *abstract* operators (**X@A**) ignore the events that occur in inner calls;
- *past* operators (**X@P**) permits to go back on the call stack.

As an example, the following formula requires that any function that enters a critical section must release it before returning:

```
G@N (Call{enter_critical} ==> (!Ret U@A Call{exit_critical}))
```

While CaFE's input language makes it easier to abstract away the calls that are not present inside the model, it still lacks the possibility of using variables that are not present in the implementation, making it unsuitable in its current form for our purposes.

2.2.3 RPP, Relational Properties Prover

Relational properties are properties that, unlike function contracts, allow specifying relations that should hold between several function calls. Two simple examples are the monotony of a function `int f(int)`, i.e. the fact that for any `x` and `y` such that `x < y`, we have `f(x) < f(y)`. Relational properties can also involve distinct functions. For instance, given two functions for encrypting and decrypting a message

```
char* crypt(char *msg, long key);
char* decrypt(char *encrypted, long key);
```

it would be desirable to have for any `msg` and `key` that the following property holds:

```
strcmp(decrypt(crypt(msg,key), key), msg) == 0
```

RPP is a Frama-C plug-in that proposes an extension to ACSL for expressing relational properties and translates those properties into two sets of ACSL annotations. The first one is dedicated to prove the property using WP, while the second provides an axiomatic that allows WP to use the property as a hypothesis in other proofs. While RPP is relatively new and under heavy development, its relational properties, by allowing the introduction of fresh variables through universal quantification, are an appropriate target for the generation of code-level specifications through Diversity. We have thus based our work in task T3.1 on this setting. Because of its importance in this activity, RPP is described in more details in chapter 5

Chapter 3

Diversity

In this chapter, we present the DIVERSITY tool developed at CEA Tech List and used in the context of our project.

3.1 Introduction: what is DIVERSITY

DIVERSITY is a customizable model analysis tool based on symbolic execution. It is an open source tool (Eclipse license EPL) available in the Eclipse Formal Modeling Project [18]. It offers an extensible platform in order to take into consideration various formal analysis possibilities.

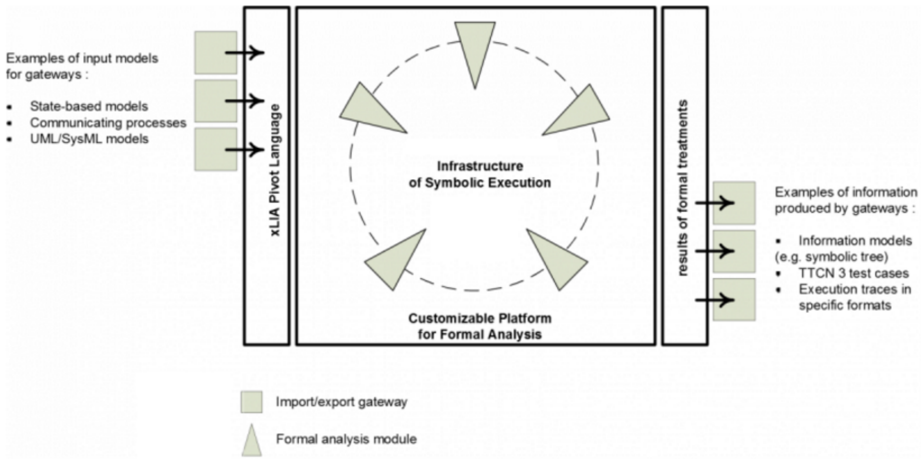

Examples of input models for gateways :
State-based models
Communicating processes
UML/SysML models
xLIA Pivot Language
Infrastructure of Symbolic Execution
Customizable Platform for Formal Analysis
results of formal treatments
Examples of information produced by gateways :
Information models (e.g. symbolic tree)
TTCN 3 test cases
Execution traces in specific formats
Import/export gateway
Formal analysis module

Figure 3.1: The symbolic execution platform of Diversity [1]

Namely, Diversity provides a common Symbolic Execution (SE) platform (Figure 3.1):

- generic enough to take into account the semantics of a wide range of models;
- extensible to allow customizing the basic symbolic treatments to implement specific Formal Analysis Modules (FAM), such as Model-Based Testing (MBT) algorithms, exploration strategies and heuristics;
- connected with many solvers, e.g., Z3[2], CVC4[3] and YICES[11] which can easily be used to implement new FAMs.

[2]https://z3.codeplex.com/
[3]http://cvc4.cs.nyu.edu/web/

DIVERSITY is composed of two distinct parts depicted in Figure 3.2: an Eclipse-based Graphical User Interface (GUI) and an SE kernel developed in C++. GUI. It provides a textual editor and a graphical editor for xLIA models. The latter is implemented as a UML/SysML specialization using the Papyrus technology. The GUI provides as well utilities for symbolic tree visualization and debug with different logging levels. Some of the existing FAMs of DIVERSITY have been associated with GUI forms in order to enter required user parameters. Note that DIVERSITY kernel EXE can be run in command line: it takes as input an XML-like user parameters file including the xLIA model location.

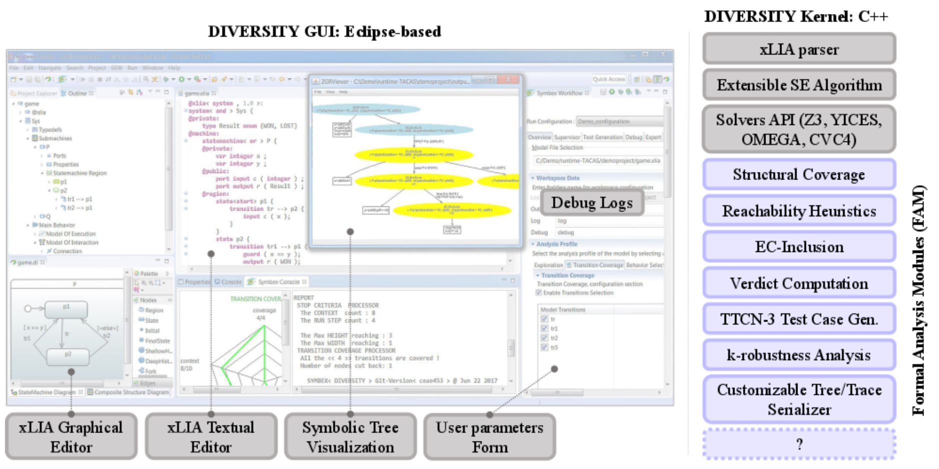


Figure 3.2: Diversity

3.2 Diversity entry language

Diversity has the ability to analyze a wide range of modeling languages. In fact, it provides a pivot language called $xLIA$ (eXecutable Language for Interaction & Architecture) which is a generic language introducing a set of communication and execution primitives allowing one to encode a wide class of dynamic model semantics, e.g., IOSTS/Timed IOSTS, UML/SysML, SDL, Communicating STS, abstractions of hybrid systems.
In particular, $xLIA$ supports classical automata syntax involving symbolic data and communication actions. In our case, we use $xLIA$ modeling language to encode transition systems where transitions are composed of a source and a target control state, and a sequence of instructions such as guards built from state variables (e.g. $x <= y$), some communication actions (receptions of values stored on state variables or emissions of values through some ports) (e.g. input $c(x)$ and output $r(OK)$), and variable updates denoted by classical assignments (e.g. $y := y + 42$).

3.3 Diversity services

Diversity is based on symbolic execution techniques. Symbolic execution (SE) consists in computing the semantics of a model in an abstract manner by constraint propagation on input symbolic values. The result of the symbolic execution is a tree-like structure where a path abstracts a class of concrete execution of the analyzed object. Symbolic trees are potentially infinite structures due to the possibility of repeatedly executing an unbounded loop for example. Classically, SE allows setting stopping criteria related to the tree size. Once such a symbolic tree is computed, SE provides post processing facilities to conduct formal analysis on the symbolic tree. In addition, the SE infrastructure has been designed to allow customized formal treatment on the fly during the construction of the symbolic tree.

These are descriptions of some available FAMs which will be used and assessed in the frame of the VESSEDIA project:

- EC-Inclusion: This FAM stops the symbolic execution of any Execution Context (EC), during pre-filtering, if it is included in another already computed EC in the symbolic tree.
- Reachability heuristics: This FAM computes a set of symbolic states that are the final states of some paths satisfying a coverage goal such as covering (successively) some transitions, states, I/O actions, function calls, or satisfying logical formulas on the model state variables.
- k-robustness analysis: this FAM is used to analyze the robustness of a design to some known cyber attacks, in terms of intrusion detection. The goal is to ensure that the design model is equipped with a watchdog able to emit a warning in less than k actions of the attacker (k is an integer specified by the designer of the detection mechanism endowed in the watchdog).
- Testing: beyond these few examples, one major application domain for DIVERSITY is model-based testing. Let us emphasize that the development of DIVERSITY has been mainly driven by needs issued from model-based testing, such as time modeling, customizable tool or selection criteria.

In the scope of VESSEDIA, a new FAM is being developed in DIVERSITY: its objective is to be used intertwined with the Reachability heuristics in order to infer code annotations for a cooperation of function calls which implement specific safe high level system scenarios.

Chapter 4

Relational properties inference from high-level model

4.1 Introduction

In this chapter, we propose a top-down approach to safety and security engineering that handles the combination of high-level system reasoning and low-level code verification. We are focusing on automatic generation of code annotations from a high-level requirements model enriched with function calls.

In fact, it may happen that some requirements specifying high-level security and safety properties occurring at the system model put constraints on the execution of called functions. In such a case we need to verify these requirements at code level, with the greatest rigor and accuracy. This implies that we need to translate high-level properties into low-level verification tasks.

To do so, we propose to automatically generate code annotations (in the format of relational properties) for functions called within the system model. The inference approach is achieved by exploring constraints coming from the model and transporting those system constraints at the functions level.

The proposed methodology is considered as an advanced formal verification framework which is implemented by combining a variety of tools: Papyrus (modeling), Diversity (model simulation and properties inference) and Frama-C (code analysis and proof). Furthermore, it presents a significant step towards bridging the gap between a model-based approach in which user-defined functions are abstracted away and a code-based approach in which small pieces of code are separately considered regardless of the way they are combined.

4.2 Proposed approach Overview

We now give a brief overview of the proposed approach for extracting relational properties on C functions from the system description. It consists in the following steps:

- System modeling taking into consideration safety and security properties. To achieve this we use UML sequence diagrams (as presented in D1.3) enriched with function calls. Then the sequence diagram will be automatically translated into an xLIA model (to be analyzed with Diversity).

- Generate properties which the code is required to realize in order to ensure that the system code operates as intended.
- Check the code correctness against inferred properties coming from the system high-level design (this will be detailed in the following chapter).

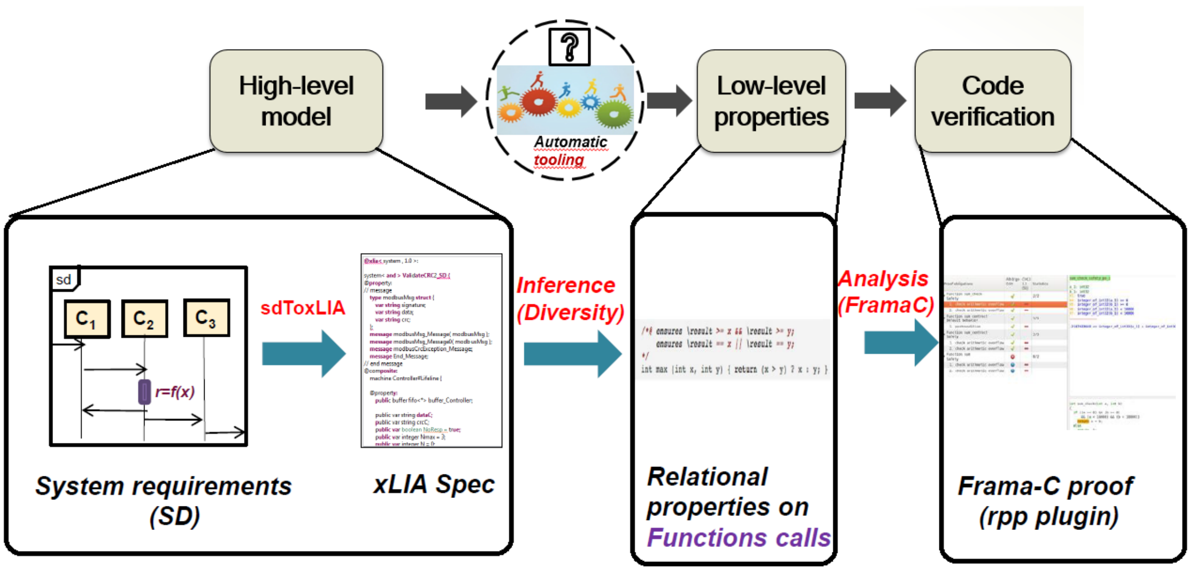


Figure 4.1: Approach overview

4.3 Papyrus tool for high-level requirements modeling

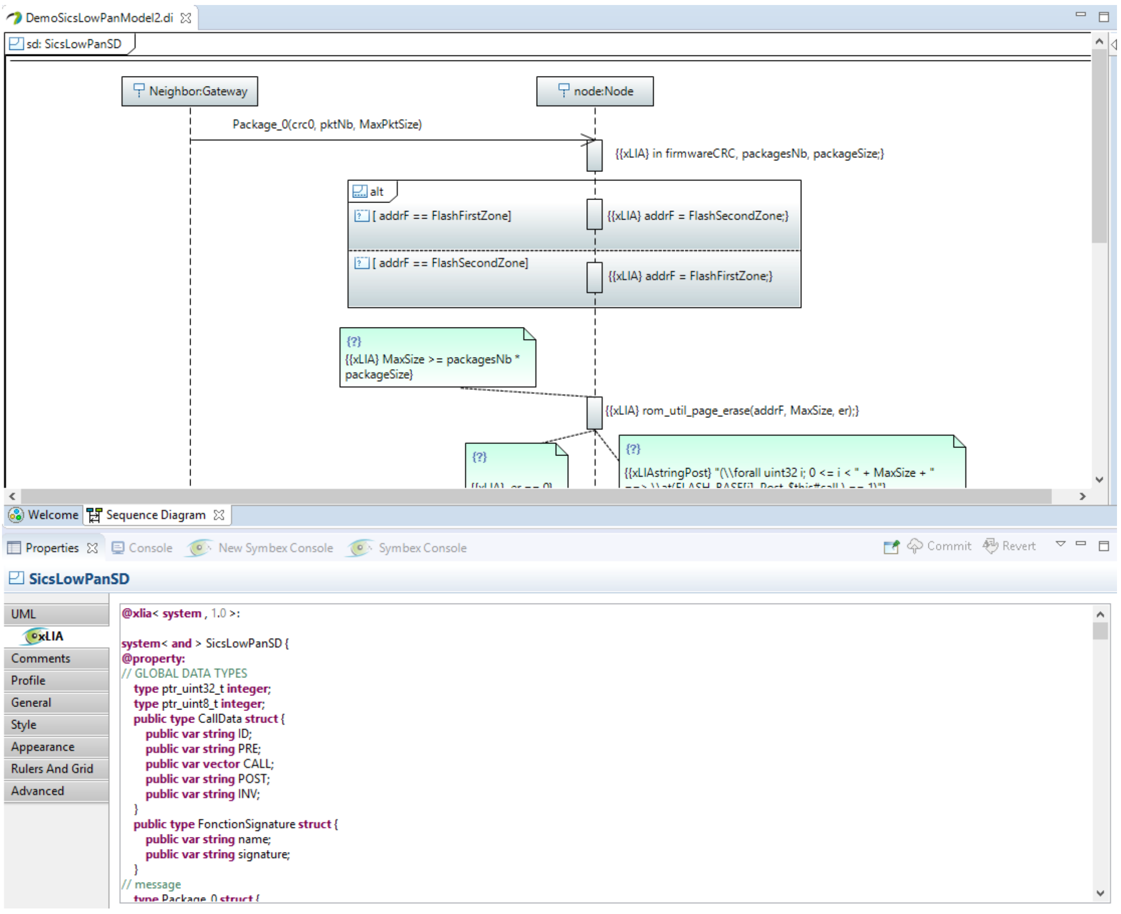
Papyrus is an UML standard Open Source modeling tool for embedded systems' design and specification and especially real time critical systems.

4.3.1 UML SD models

In the context of our work, we use Papyrus to specify systems as UML Sequence Diagrams. The complete specification of Sequence Diagrams in UML2 can be found at `http://www.omg.org/spec/UML/2.3/`. In addition, a refined version of this specification tailored to our needs in Vessedia is described in D1.3 In summary, a Sequence Diagram allows describing interactions between components of a system. Basically, Lifelines represent parts life cycle, communication between parts are represented by message exchanges and executions are represented by Execution specifications that can be Action or Behavior execution. In particular, User-defined functions can be called within the sequence diagram as behavior executions.

4.3.2 SD to xLIA translator

The UML sequence diagram semantics is implemented by translating it to an xLIA model (IOSTS), on which Diversity can be applied. xLIA allows representing the semantics of the sequence diagrams in a more compact way, better suited to symbolic execution and especially more modular (the regions of the combined fragment are translated into hierarchical / composite states). This last point will allow easier maintenance of this translator. The translation is a hierarchical translation that respects the semantics of the UML model. First, we perform model-to-model translation, from UML Sequence diagram to UML State Machine diagram. In a second step, we transform the UML State Machine into xLIA since both format are semantically close: both are based on the notion of automaton and transitions system. In addition, the results of this translation are viewable in the property view of the DIVERSITY GUI (see figure 4.2), which facilitates debugging.


DemoSicsLowPanModel2.di
sd: SicsLowPanSD
Neighbor:Gateway
node:Node
Package_0(crc0, pktNb, MaxPktSize)
{{xLIA} in firmwareCRC, packagesNb, packageSize;}
alt
[addrF == FlashFirstZone]
{{xLIA} addrF = FlashSecondZone;}
[addrF == FlashSecondZone]
{{xLIA} addrF = FlashFirstZone;}
{?}
{{xLIA} MaxSize >= packagesNb * packageSize}
{{xLIA} rom_util_page_erase(addrF, MaxSize, er);}
{?}
{?}
{{xLIAstringPost} "(\\forall uint32 i; 0 <= i < " + MaxSize + "
Welcome
Sequence Diagram
Properties
Console
New Symbex Console
Symbex Console
Commit
Revert
SicsLowPanSD
UML
xLIA
Comments
Profile
General
Style
Appearance
Rulers And Grid
Advanced

```
@xlia< system , 1.0 >:

system< and > SicsLowPanSD {
@property:
// GLOBAL DATA TYPES
  type ptr_uint32_t integer;
  type ptr_uint8_t integer;
  public type CallData struct {
    public var string ID;
    public var string PRE;
    public var vector CALL;
    public var string POST;
    public var string INV;
  }
  public type FonctionSignature struct {
    public var string name;
    public var string signature;
  }
// message
  type Package_0 struct {
```

Figure 4.2: SD to xLIA translator

4.4 New Diversity module for automatic relational properties inference

In this section, we focus on code annotations inference from UML sequence diagrams. More precisely, we start with a sequence diagram, translated to an xLIA model (IOSTS) calling functions and we show how to generate constraints on called functions code from path condition constraints.

4.4.1 Properties inference methodology

Path conditions (constraints gathered with SE and that characterize under which circumstances a given execution path might be taken) are not reduced to one function call. For that reason, we propose to generate code annotations in the format of a relational property (per path, or in other words per possible system behavior) which is a formula that, in the general case, may involve several function calls.

The inference approach is achieved by exploring constraints coming from the IOSTS symbolic execution and transporting those system constraints at the functions level. Our working hypothesis for the inference is that the concrete implementation of the functions should not add additional constraints to the system. In other words, if a path condition is satisfiable when considering the function calls as returning abstract values on which nothing is known beyond the patch condition itself, it must still be satisfiable when using an actual implementation for each function. In practice we generate the relational property using the following steps:

- identify the set of variables of the path condition related to the sequence of function calls occurring along the path;
- elicit the sub-formula from the path condition on which the calls depend;
- use the obtained constraints to formulate the relational property in a format acceptable by Frama-C (and more specifically its RPP plugin).

Example 4.4.1 *In order to explain clearly our approach, we focus on the toy example shown in figure 4.3. It is an $IOSTS$ describing a computational process calling two functions f and g with some input/output actions and some guards.*

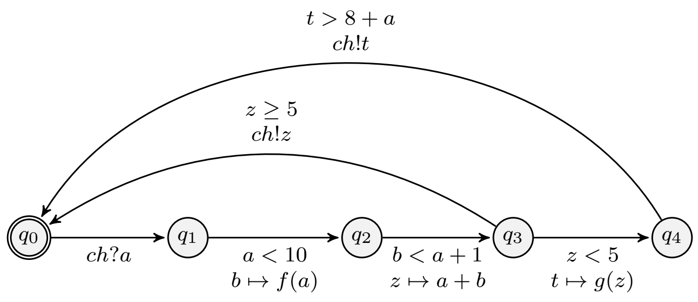


Figure 4.3: IOSTS

In the following we consider the IOSTS path $q_0 - q_1 - q_2 - q_3 - q_4 - q_0$. It includes two function calls (f and g) and its path condition is $a_1 < 10 \wedge b_1 < a_1 + 1 \wedge a_1 + b_1 < 5 \wedge t_1 > 8$, where a_1, b_1 and t_1 are free variables generated by the symbolic execution, corresponding respectively to the value received on ch during the first transition, the value returned by $f(a)$ and the value returned by $g(z)$ (or in terms of symbolic variables $g(a_1 + b_1)$).

Suppose that the user wants to make sure that this particular behavior of the given system is feasible. In order to reach the leaf of the considered path, the called functions within the path must satisfy some constraints that may be deduced from the path condition. These constraints will be automatically generated in the format of a relational property on different functions called within the considered path. In our example, the relational property corresponding to the given path is shown in figure 4.5. Let us note that function calls are explicitly

$$\begin{tikzcd}[row sep=small]
Init : (q_0, \top, [a \rightarrow a_0, b \rightarrow b_0, z \rightarrow z_0, t \rightarrow t_0]) \arrow[d, "ch?a_1"] \\
\eta_1 : (q_1, \top, [a \rightarrow a_1, \cdots]) \arrow[d, "\tau"] \\
\eta_2 : (q_2, a_1 < 10, [b \rightarrow b_1, \cdots], \{(f, a_1, b_1)\}) \arrow[d, "\tau"] \\
\eta_3 : (q_3, a_1 < 10 \wedge b_1 < a_1 + 1, [z \rightarrow a_1 + b_1, \cdots], \{(f, a_1, b_1)\}) \arrow[d, "\tau"'] \\
\begin{array}{c}\eta_4 : (q_4, a_1 < 10 \wedge b_1 < a_1 + 1 \wedge a_1 + b_1 < 5, \\ [t \rightarrow t_1 \cdots], \{(f, a_1, b_1), (g, a_1 + b_1, t_1)\})\end{array} \arrow[d, "ch?t_1"'] \\
\begin{array}{c}\eta_6 : (q_0, a_1 < 10 \wedge b_1 < a_1 + 1 \wedge a_1 + b_1 < 5 \wedge t_1 > 8, \\ [\cdots], \{(f, a_1, b_1), (g, a_1 + b_1, t_1)\})\end{array}
\end{tikzcd}$$

Figure 4.4: Symbolic path

specified in the $\backslash callset$ construct and $\backslash call(f, < args >)$, denoting the call $f(< args >)$ to f with arguments $< args >$. Each call is associated to an identifier (namely id_f and id_g in our case). $\backslash callresult$ takes a $call - id$ as parameter and refers to the value returned by the corresponding call.

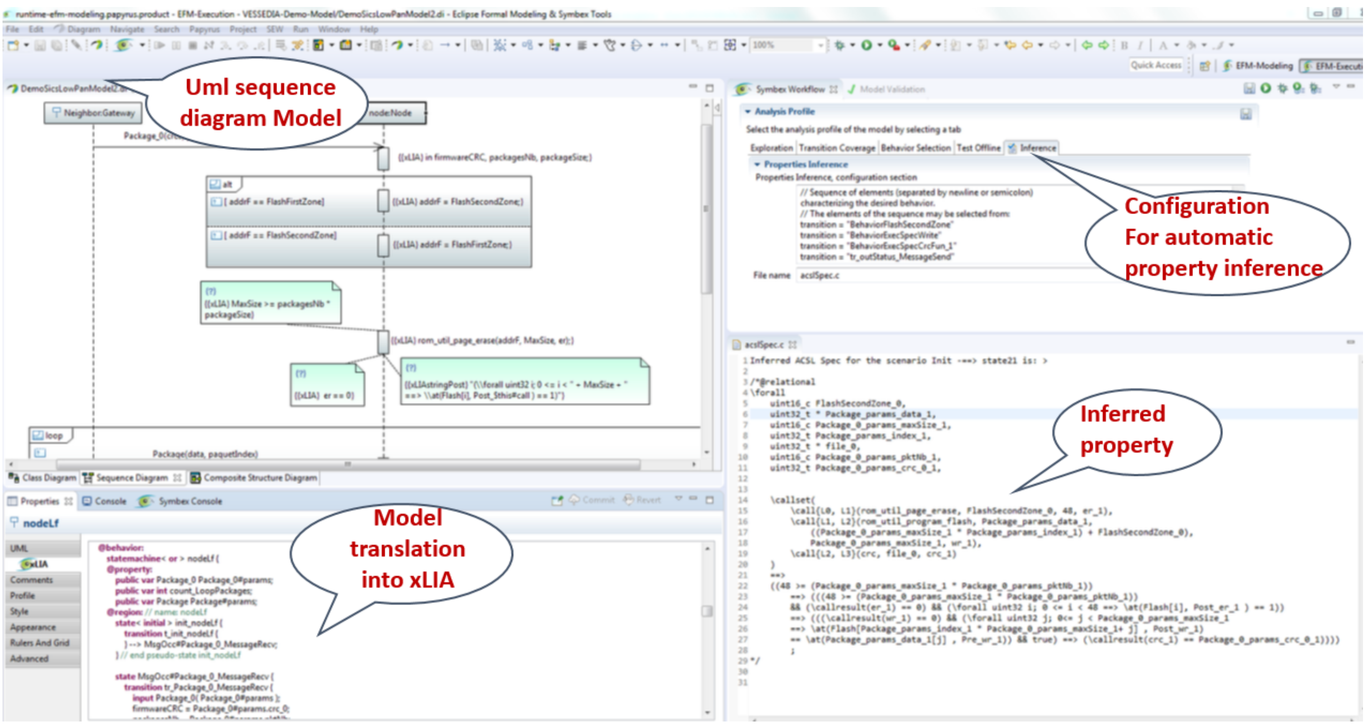
```
/*@relational
\forall
 integer $a_1$;
 \callset(
        \call(f, $a_1$, $id_f$),
        \call(g, $a_1$ + \callresult($id_f$), $id_g$)
 )
 ==>
 ($a_1 < 10$ ==>
        (\callresult($id_f$)$< a_1 + 1$ && $a_1$+ \callresult($id_f$)$< 5 ==>$
                  \callresult($id_g$) $> 8 + a_1$)
  );
*/
```

Figure 4.5: Inferred relational property for path feasibility

4.4.2 The inference module GUI

Our methodology is developed as a FAM in Diversity. The inference FAM has been associated with GUI forms in order to enter required user parameters. The user can configure the input model, the behavior for which they want to automatically generate a corresponding relational property and the output file that will contain the property.

The following figure shows the inference module within the GUI forms.



Select the analysis profile of the model by selecting a tab

Properties Inference, configuration section

```
// Sequence of elements (separated by newline or semicolon)
characterizing the desired behavior.
// The elements of the sequence may be selected from:
transition = "BehaviorFlashSecondZone"
transition = "BehaviorExecSpecWrite"
transition = "BehaviorExecSpecCrcFun_1"
transition = "tr_outStatus_MessageSend"
```

File name: acslSpec.c

```
@behavior:
  statemachine< or > nodeLf {
  @property:
    public var Package_0 Package_0#params;
    public var int count_LoopPackages;
    public var Package Package#params;
  @region: // name: nodeLf
    state< initial > init_nodeLf {
      transition t_init_nodeLf {
      } --> MsgOcc#Package_0_MessageRecv;
    } // end pseudo-state init_nodeLf

    state MsgOcc#Package_0_MessageRecv {
      transition tr_Package_0_MessageRecv {
        input Package_0( Package_0#params );
        firmwareCRC = Package_0#params.crc_0;
```

acslSpec.c

```
Inferred ACSL Spec for the scenario Init -==> state21 is: >

/*@relational
\forall
    uint16_c FlashSecondZone_0,
    uint32_t * Package_params_data_1,
    uint16_c Package_0_params_maxSize_1,
    uint32_t Package_params_index_1,
    uint32_t * file_0,
    uint16_c Package_0_params_pktNb_1,
    uint32_t Package_0_params_crc_0_1,


    \callset(
        \call(L0, L1)(rom_util_page_erase, FlashSecondZone_0, 48, er_1),
        \call(L1, L2)(rom_util_program_flash, Package_params_data_1,
            ((Package_0_params_maxSize_1 * Package_params_index_1) + FlashSecondZone_0),
            Package_0_params_maxSize_1, wr_1),
        \call(L2, L3)(crc, file_0, crc_1)
    )
    ==>
    ((48 >= (Package_0_params_maxSize_1 * Package_0_params_pktNb_1))
        ==> (((48 >= (Package_0_params_maxSize_1 * Package_0_params_pktNb_1))
        && (\callresult(er_1) == 0) && (\forall uint32 i; 0 <= i < 48 ==> \at(Flash[i], Post_er_1 ) == 1))
        ==> (((\callresult(wr_1) == 0) && (\forall uint32 j; 0<= j < Package_0_params_maxSize_1
        ==> \at(Flash[Package_params_index_1 * Package_0_params_maxSize_1+ j] , Post_wr_1)
        == \at(Package_params_data_1[j] , Pre_wr_1)) && true) ==> (\callresult(crc_1) == Package_0_params_crc_0_1))))
        ;
*/
```

Figure 4.6: Inference interface

Chapter 5

Relational Property Prover Plug-in

This chapter describes the RPP plug-in, that takes as input the formulas generated by Diversity, together with a C implementation of the UML model, and will try to prove that the implementation is conforming to the properties. We first present the input language of RPP in detail (section 5.1). The verification task can be decomposed in two steps: First, RPP generates a wrapper function, in the spirit of self-composition [3], together with a set of annotations whose proof amounts to proving that the relational property holds for the original functions. This is explained in section 5.2. At the same time, RPP is generating an ACSL axiomatic that makes it possible to use the relational property as a hypothesis when verifying other ACSL annotations. While this feature has not been used in the context of T3.1, we briefly describe it in section 5.3.

Once the annotations are generated, RPP delegates the proof task itself to the deductive verification plug-in of the platform, WP [4]. As with any deductive verification task, this might require writing additional ACSL annotations, notably loop invariants. This process is presented in section 5.4. Finally, for annotations that WP is unable to prove, it is possible to try generating counter-examples, i.e. sets of inputs for which the property does not hold. This relies on the StaDy [21, 20] plug-in, that executes the program over test cases generated by the PathCrawler [9] plug-in and checks whether one of the cases invalidates the property, as translated into C by the E-ACSL [23, 19] plug-in. While this possibility has not been used yet on the use case presented in chapter 6, we describe it in section 5.5. From a VESSEDIA perspective, it is also worth noting that it presents many similarities with the work done in T3.2 for combining static and dynamic analysis.

5.1 Relational Properties Language

RPP takes advantage of the ACSL extension mechanism proposed in Frama-C. Namely, a plug-in can register a new keyword (in our case `relational`) in order to introduce new clauses in a function contract beyond the ones that are already recognized. For historical reasons, it is not possible to extend global annotations this way, which means that a relational property over several functions must be written in the contract of the last function to be declared (in order for the others to be in scope).

A first kind of relational clause can be written when the functions involved in the property behave like pure mathematical functions, that do not perform any access (neither writing nor reading) to the global memory. In other words, they do not perform any global side-effect and their returned value only depends on the input parameters. In that case, a relational clause can be basically reduced to a standard ACSL predicate, with special terms of the form `\call(f,x_1,...,x_n)` to denote the result of a call to `f` with the corresponding arguments. As an example, the following clause specifies that f should be increasing.

```
relational inc:
  \forall int x,y; x <= y ==> \call(f,x) <= \call(f,y);
```

However, as soon as we are considering functions that may read from and/or write to locations in the global memory, this simple syntax is not sufficient. Indeed, we need to be able to refer in the relational formula to the values of these locations at the various pre- and post- states of the functions involved in the property. In order to take that into account, the additional construction **`\callset`** can be used to give a name to each of the calls, and use these names as a basis to refer to their respective pre- and post- state through the use of the standard ACSL contruction `\at(expr, Lab)`, which indicates that a given expression is to be evaluated in the state corresponding to the given label. In addition, `\callresult(name)` allows refering to the result return by the call identified by the corresponding name. As an example, the relational property over functions:

```
void encrypt(char *msg,unsigned char key, size_t length);
void decrypt(char *msg, unsigned char key, size_t length);
```

indicating that decrypting an encrypted message with the same key should give back the original content can be expressed as follows (where `same_content` is a predicate that compares the content of two buffers in two separate memory states).

```
relational
  \forall char *msg, *enc;
  \forall unsigned char k;
  \forall size_t n;
  \callset(\call(encrypt,msg,k,n,encryption),
           \call(decrypt,enc,k,n,decryption)) ==>
    same_content{Post_encryption, Pre_decryption}(msg, enc, 0, n)
    ==>
    same_content{Pre_encryption, Post_decryption}(msg, enc, 0, n);
```

This is this second form of annotation that is the target of Diversity.

5.2 Annotation Generation for Verification

The idea beyond the generation of plain ACSL annotations for verifying relational properties is relatively simple. Basically, all that is needed is generating a wrapper function that will simulate the calls involved in the property and store the intermediate results into local variables, that can then be referenced to into an ACSL **`assert`**. The wrapper takes the universally quantified variables as parameters at the top of the formula, and if there are conditions over them, takes that as **`requires`** clause. Similarly, pre-conditions of the functions involved in the properties are lifted to the wrapper.

One key element of this technique, known as self-composition, is that each distinct call must operate on a separate portion of the global memory from the others in order to ensure a complete independence of the calls made by the wrapper. This is done by generating additional local variables, based on the **`assigns ... \from ...`** clauses provided in plain

ACSL contracts. It is therefore required that all functions involved in a relational property are equipped with such clauses, that give the set of locations that may be written to during a call (the **assigns** part, which is also required for doing plain deductive verification with WP), and for each of them the set of locations whose content may be read to compute the new value (the **\from** part).

The wrapper corresponding to the `encrypt/decrypt` example above is then the following:

```
/*@ requires independence: \separated(msg+(0 .. n), enc+(0 .. n));
    requires req_encrypt: \valid(msg+(0 .. n));
    requires req_decrypt: \valid(enc+(0 .. n));
*/
void relational_wrapper(
  char *msg, char* enc, unsigned char k, size_t n) {

/*@ assert req_encrypt: \valid(msg + (0 .. n)); */
// inlining of the body of encrypt

/*@ assert req_decrypt: \valid(enc + (0 .. n)); */

// inlining of the body of decrypt

/*@ assert relational_property:
      same_content{Here, Pre}(msg, enc, 0, n) ==>
      same_content{Pre, Here}(msg, enc, 0, n);
*/
}
```

The `independence` requirements ensures that `encrypt` and `decrypt` will operate on separate buffers, while the `req_encrypt` and `req_decrypt` requirements and assertions impose the pre-conditions of each functions over the corresponding arguments of the wrapper. Finally, the assertion at the end of the wrapper is the property that we want to check, now reduced to a standard ACSL annotation that can be proved by WP, as will be shown in section 5.4.

5.3 Axiomatic Generation

One of the originalities of RPP is that it is not meant only to verify relational properties, but it also generates ACSL annotations that make it possible to use them as hypotheses in other deductive verification activities, not necessarily related to other relational properties. Again, the key idea in this direction is quite simple: we generate into an axiomatic a declaration of an ACSL predicate for each function involved in the property, and an axiom corresponding to the property. Then, we relate each predicate with the corresponding C function by adding an additional post-condition indicating the pre- and post-states of the function must verify the predicate. Again, using predicates is required to deal with non-pure functions.

The axiomatics generated for our running example is the following:

```
/*@ axiomatic Relational_axiom_1 {
  predicate encrypt_acsl{pre, post}(char *msg,  key,  n)
    reads \at(*(msg + (0 .. n)),post), \at(*(msg + (0 .. n)),pre);
  predicate decrypt_acsl{pre, post}(int *enc,  key,  n)
    reads \at(*(msg + (0 .. n)),post), \at(*(msg + (0 .. n)),pre);

  lemma Relational_lemma
     {pre_encryption, post_encryption,
      pre_decryption, post_decryption}:
    \forall char *msg, *enc;
    \forall unsigned char k;
    \forall size_t n;
      encrypt_acsl{pre_encryption, post_encryption}(msg, k, n) ==>
      decrypt_acsl{pre_decryption, post_decryption}(enc, k, n) ==>
      same_content{post_encryption, pre_decryption}(msg, enc, 0, n)
      ==>
      same_content{pre_encryption, post_decryption}(msg, enc, 0, n);
  }
*/
```

The **reads** clauses are directly extracted from the **assigns ... \from ...** of the functions, and indicates all the memory locations in both of the states that are involved in deciding the truth value of the predicate. Then, the **lemma** states that for any four states that can be related respectively by `encrypt_acsl` and `decrypt_acsl`, the relational property hold.

Finally, the post-condition for the `encrypt` function is the following (the post-condition for `decrypt` is nearly identical, except that of course it calls the `decrypt_acsl` predicate).

```
ensures encrypt_acsl{Pre,Post}(msg, k, n);
```

5.4 Using WP to Prove Relational Properties

RPP does not prove anything by itself. Instead, once ACSL annotations have been generated, it relies on WP to perform a normal deductive verification on the new annotations. As is always the case with this technique, some additional annotations may be required though. In particular, all loops appearing in the body of one of the functions involved in the relational property must be equipped with **loop invariants**. These invariants are normal ACSL loop invariants that are meant to abstract the behavior of the loop, regardless of the numbers of steps that have been done. Similarly, the locations that may be modified during the execution of the loop must be given in a standard ACSL **loop assigns** clause. There is currently no mechanism to write relational property specific clauses for loops, that could for instance make an invariant dependent on the behavior of other calls involved in the property.

Once WP has run on the wrapper, RPP will lift its results to the initial relational property. More precisely, it takes advantage of the property status dependency mechanism of Frama-C [11] to indicate that:

- the relational property holds as soon as the last **assert** at the end of the corresponding wrapper function is verified;
- the lemma in the axiomatic also holds whenever the relational property holds
- the post-conditions introduced in the contracts of the functions to make the correspondence with the generated ACSL predicates always hold (more precisely, they depend

on the correction of RPP itself, something which has to be assessed outside of Frama-C).

5.5 Using StaDy to find Counter-Examples

When WP fails to prove some ACSL annotation, it is not always easy to understand the causes of the problem. Basically, three main issues can appear:

- the automated provers are not powerful enough for the kind of property that is given to them;
- some auxiliary annotations (notably loop invariants or loop assigns) are missing;
- there's a bug in the implementation, the specification, or both.

The terse Valid/Unknown answer provided by WP for each proof obligation that it has generated is generally not sufficient to decide in which case we are. A possible answer to this issue is providing the user with a *counter-example*: a set of inputs that falsifies the property. The StaDy plug-in [20] is meant to do that. More precisely, it takes advantage of the test-case generation plug-in PathCrawler [9], that structurally explores all paths of the function under analysis (up to a certain depth of course), and guides it to generate test cases that will activate the branch in which the annotation to be verified lies. If one of the generated test cases falsifies the annotation, it can be shown to the user to help them understand the situation that leads to the problematic behavior of the program. On the other hand, if no set of inputs falsifies the annotation, it is not possible to be sure that the annotation holds, unless PathCrawler can conclude that it has explored all possible paths of interest (e.g. if no loop is involved in the computation).

In addition to WP, RPP is thus also able to use StaDy as a backend and identify inputs for which the desired relational property does not hold. This has been experimented [7] on a small benchmark unrelated to the VESSEDIA use case described in chapter 6 but could probably be used on it as well if needed. However, counter-examples would be expressed over the implementation (C code), not on the UML model itself, as there is currently no mechanism to trace back the results of the code analysis up to the model.

Chapter 6

Example over 6LowPAN use-case

6.1 Quick reminder of 6LowPAN use-case

The 6LowPAN management platform aims at maintaining a good performance and sustainability of the 6LowPAN networks. This includes over-the-air firmware update operations on the 6LowPAN nodes, where the firmware update may be partial/modular or full. The platform comprises three functional components: the management server, the gateway, and the managed node.

- Management server: this component runs on Android OS. It is in charge of transmitting firmware updates and reboot requests to the Low power & Lossy Network (LLN) node.
- Gateway (GW): it runs on Embedded Linux OS. It is in charge of interconnecting the LLN network with a WAN to enable communication exchange between the management server and the LLN node.
- LLN node: it is an embedded hardware platform with a microcontroller in addition to one or more sensors and/or actuators. It runs on Contiki OS. The managed node runs some specific application layer program (e.g., transmitting environmental/physical information like temperature or position to the management server). In addition, the LLN node is in charge of forwarding/routing data packets in the LLN network.

6.2 SD: firmware update scenario modeling

According to its manual [16], the LLN node dynamically loads the new (piece of) firmware after it completes the reception of firmware update data from the management server. In this section, we propose an SD that models the firmware update scenario edited with Papyrus. In the following, we give a brief description of its main steps:

- Receive the first package which contains some meta-data like the number of packages, the package size, the crc code of the firmware. or etc.
- Load the address into the flash memory (which contains two partitions).
- Release of the flash memory area that will receive the firmware (modeled with the ”rom_util_page_erase” function call).

- Receive the rest (with the Loop CombinedFragment) of the firmware packages and write each received package to the flash memory (modeled with the ”rom_util_program_flash” function call).
- Integrity check (modeled with the ”crc” function call).
- Reception confirmation sent by the node to the gateway.

In the actual implementation of the use case, these functions include significant parts written directly in assembly code (`asm` directive) and not in C. As part of our experimental setup in this task, we have proposed equivalent C code to replace these parts. This methodology is quite standard for handling assembly code within a Frama-C-based verification task and requires of course a manual validation step for ensuring that the C transposition is indeed a faithful representation of the original assembly code. In the present case, this has been confirmed by working closely with the use-case owners.

Independently from the VESSEDIA project, work is done at CEA to use the BINSEC binary analyzer [6] to automatically generate C code from assembly and validate the equivalence. While this work is in a too early stage to be deployed on real-world use cases, it will probably be possible in the future to automate this stage as well.

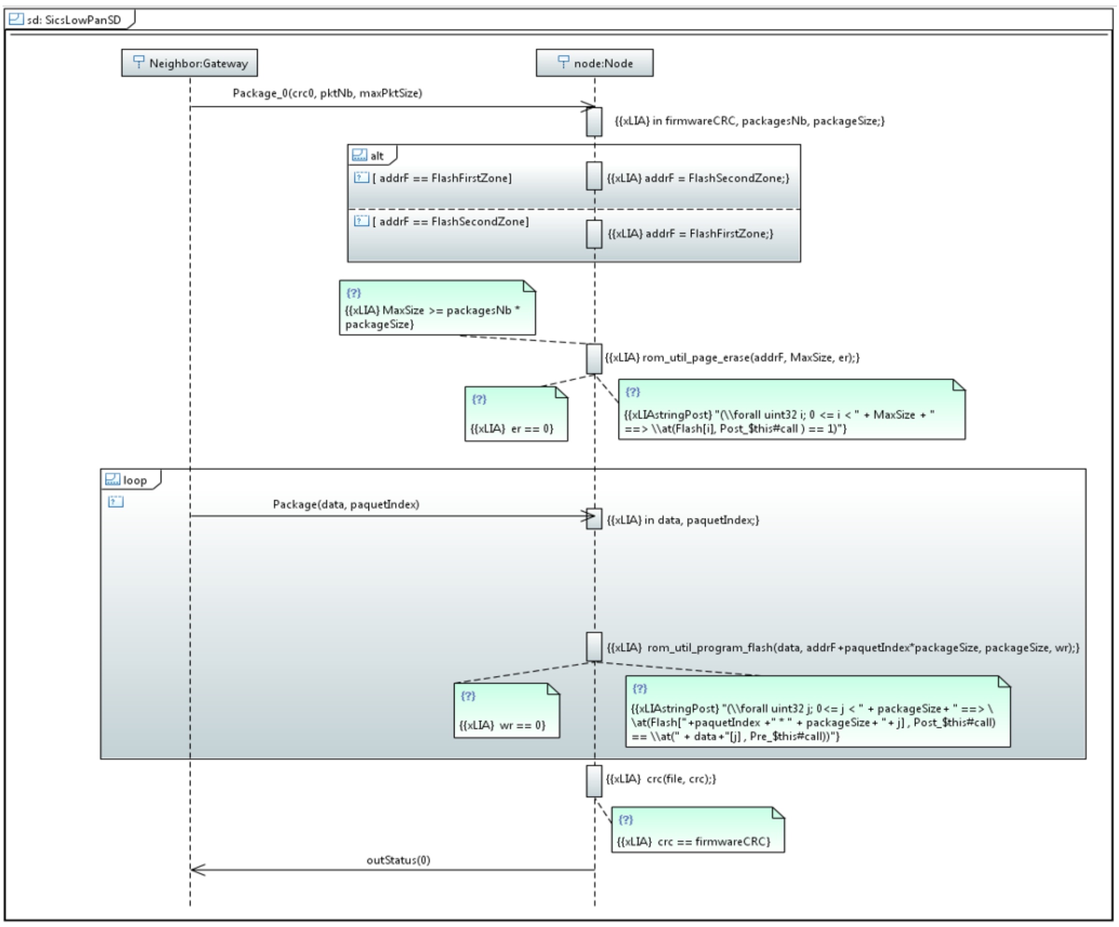
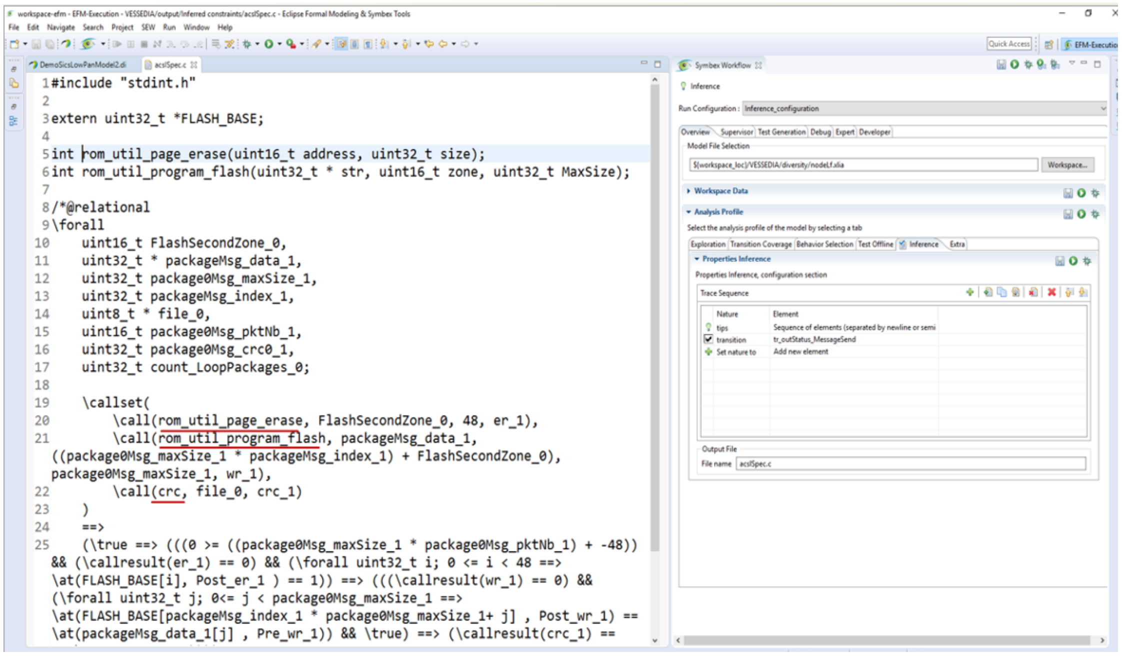

sd: SicsLowPanSD
Neighbor:Gateway
node:Node
Package_0(crc0, pktNb, maxPktSize)
{{xLIA} in firmwareCRC, packagesNb, packageSize;}
alt
[addrF == FlashFirstZone]
{{xLIA} addrF = FlashSecondZone;}
[addrF == FlashSecondZone]
{{xLIA} addrF = FlashFirstZone;}
{?}
{{xLIA} MaxSize >= packagesNb * packageSize}
{{xLIA} rom_util_page_erase(addrF, MaxSize, er);}
{?}
{{xLIA} er == 0}
{?}
{{xLIAstringPost} "(\\forall uint32 i; 0 <= i < " + MaxSize + " ==> \\at(Flash[i], Post_$this#call) == 1)"}
loop
Package(data, paquetIndex)
{{xLIA} in data, paquetIndex;}
{{xLIA} rom_util_program_flash(data, addrF+paquetIndex*packageSize, packageSize, wr);}
{?}
{{xLIA} wr == 0}
{?}
{{xLIAstringPost} "(\\forall uint32 j; 0<= j < " + packageSize + " ==> \\at(Flash["+paquetIndex +" * " + packageSize + "+ j] , Post_$this#call) == \\at(" + data+"[j] , Pre_$this#call))"}
{{xLIA} crc(file, crc);}
{?}
{{xLIA} crc == firmwareCRC}
outStatus(0)

Figure 6.1: Firmware update SD

6.3 Property inference and proof results

The modeled SD serves as a basis for the relational property generation. In the following we show the low-level property corresponding to the selected firmware update scenario in the form of a relational property that permits to check if no involved function would inappropriately return a success status even though an error (e.g. reception of a bad packet or write failure) occurred.


workspace-efm - EFM-Execution - VESSEDIA/output/Inferred constraints/acslSpec.c - Eclipse Formal Modeling & Symbex Tools
File Edit Navigate Search Project SEW Run Window Help
Quick Access
EFM-Execution
DemoSicsLowPanModel2.di
acslSpec.c
Symbex Workflow
Inference

```
#include "stdint.h"

extern uint32_t *FLASH_BASE;

int rom_util_page_erase(uint16_t address, uint32_t size);
int rom_util_program_flash(uint32_t * str, uint16_t zone, uint32_t MaxSize);

/*@relational
\forall
    uint16_t FlashSecondZone_0,
    uint32_t * packageMsg_data_1,
    uint32_t package0Msg_maxSize_1,
    uint32_t packageMsg_index_1,
    uint8_t * file_0,
    uint16_t package0Msg_pktNb_1,
    uint32_t package0Msg_crc0_1,
    uint32_t count_LoopPackages_0;

    \callset(
        \call(rom_util_page_erase, FlashSecondZone_0, 48, er_1),
        \call(rom_util_program_flash, packageMsg_data_1,
((package0Msg_maxSize_1 * packageMsg_index_1) + FlashSecondZone_0),
package0Msg_maxSize_1, wr_1),
        \call(crc, file_0, crc_1)
    )
    ==>
    (\true ==> (((0 >= ((package0Msg_maxSize_1 * package0Msg_pktNb_1) + -48))
&& (\callresult(er_1) == 0) && (\forall uint32_t i; 0 <= i < 48 ==>
\at(FLASH_BASE[i], Post_er_1 ) == 1)) ==> (((\callresult(wr_1) == 0) &&
(\forall uint32_t j; 0<= j < package0Msg_maxSize_1 ==>
\at(FLASH_BASE[packageMsg_index_1 * package0Msg_maxSize_1+ j] , Post_wr_1) ==
\at(packageMsg_data_1[j] , Pre_wr_1)) && \true) ==> (\callresult(crc_1) ==
```

Run Configuration : Inference_configuration

Overview | Supervisor | Test Generation | Debug | Expert | Developer

Model File Selection: ${workspace_loc}/VESSEDIA/diversity/nodeLf.xlia

Workspace Data

Analysis Profile

Select the analysis profile of the model by selecting a tab

Exploration | Transition Coverage | Behavior Selection | Test Offline | Inference | Extra

Properties Inference

Properties Inference, configuration section

Trace Sequence

Nature	Element
tips	Sequence of elements (separated by newline or semi
✓ transition	tr_outStatus_MessageSend
Set nature to	Add new element

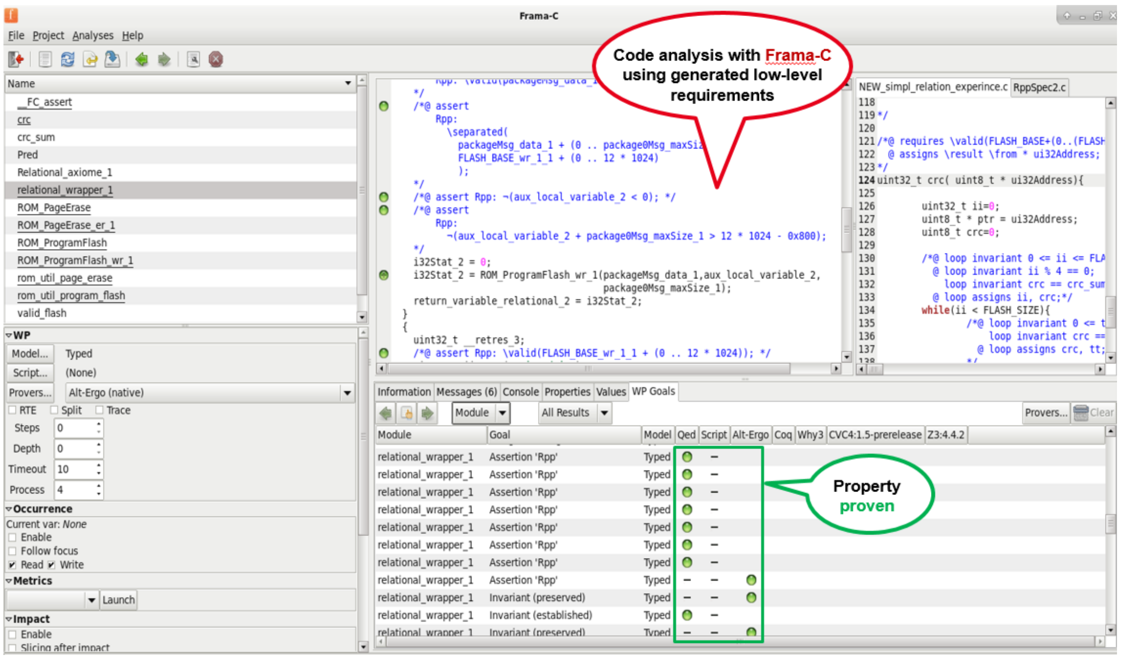
Output File

File name: acslSpec.c

Figure 6.2: Relational property inference

Let us note that the property annotates the functions called within the model. It will be used later for low-level verification with Frama-C in order to check if the functions' code is conforming to the paths conditions extracted by Diversity.

The following figure then shows the result of the proof of the firmware update code with the generated property, using the RPP plugin of Frama-C. The green bullets show that the corresponding ACSL annotations (as generated by RPP from the initial relational property) are proven.



Frama-C
File Project Analyses Help
Code analysis with Frama-C using generated low-level requirements
Property proven

Name
- _FC_assert
- crc
- crc_sum
- Pred
- Relational_axiome_1
- relational_wrapper_1
- ROM_PageErase
- ROM_PageErase_er_1
- ROM_ProgramFlash
- ROM_ProgramFlash_wr_1
- rom_util_page_erase
- rom_util_program_flash
- valid_flash

WP
- Model... Typed
- Script... (None)
- Provers... Alt-Ergo (native)
- RTE, Split, Trace
- Steps 0
- Depth 0
- Timeout 10
- Process 4

Occurrence
- Current var: *None*
- Enable
- Follow focus
- ✓ Read ✓ Write

Metrics
- Launch

Impact
- Enable
- Slicing after impact

```
      */
    /*@ assert
        Rpp:
          \separated(
            packageMsg_data_1 + (0 .. package0Msg_maxSiz
            FLASH_BASE_wr_1_1 + (0 .. 12 * 1024)
            );
      */
    /*@ assert Rpp: ¬(aux_local_variable_2 < 0); */
    /*@ assert
        Rpp:
          ¬(aux_local_variable_2 + package0Msg_maxSize_1 > 12 * 1024 - 0x800);
      */
    i32Stat_2 = 0;
    i32Stat_2 = ROM_ProgramFlash_wr_1(packageMsg_data_1,aux_local_variable_2,
                                      package0Msg_maxSize_1);
    return_variable_relational_2 = i32Stat_2;
  }
  {
    uint32_t __retres_3;
    /*@ assert Rpp: \valid(FLASH_BASE_wr_1_1 + (0 .. 12 * 1024)); */
```

NEW_simpl_relation_experince.c | RppSpec2.c

```

*/

/*@ requires \valid(FLASH_BASE+(0..(FLASH
  @ assigns \result \from * ui32Address;
*/
uint32_t crc( uint8_t * ui32Address){

        uint32_t ii=0;
        uint8_t * ptr = ui32Address;
        uint8_t crc=0;

        /*@ loop invariant 0 <= ii <= FLA
          @ loop invariant ii % 4 == 0;
            loop invariant crc == crc_sum
          @ loop assigns ii, crc;*/
        while(ii < FLASH_SIZE){
                /*@ loop invariant 0 <= t
                    loop invariant crc ==
                  @ loop assigns crc, tt;
```

Information | Messages (6) | Console | Properties | Values | WP Goals

Module | All Results | Provers... | Clear

Module	Goal	Model	Qed	Script	Alt-Ergo	Coq	Why3	CVC4:1.5-prerelease	Z3:4.4.2
relational_wrapper_1	Assertion 'Rpp'	Typed	●	–					
relational_wrapper_1	Assertion 'Rpp'	Typed	●	–					
relational_wrapper_1	Assertion 'Rpp'	Typed	●	–					
relational_wrapper_1	Assertion 'Rpp'	Typed	●	–					
relational_wrapper_1	Assertion 'Rpp'	Typed	●	–					
relational_wrapper_1	Assertion 'Rpp'	Typed	●	–					
relational_wrapper_1	Assertion 'Rpp'	Typed	●	–					
relational_wrapper_1	Assertion 'Rpp'	Typed	–	–	●				
relational_wrapper_1	Invariant (preserved)	Typed	–	–	●				
relational_wrapper_1	Invariant (established)	Typed	●	–					
relational_wrapper_1	Invariant (preserved)	Typed	–	–	●				

Figure 6.3: Proof

Glossary

Abbreviation	Translation
ACSL	ANSI/ISO C Specification Language
AST	Abstract Syntax Tree
EC	Execution Context
EVA	Evolved Value Analysis
FAM	Formal Analysis Module
GUI	Graphical User Interface
GW	Gateway
IOSTS	Input-Output Symbolic Transition System
LLN	Low-power and Lossy Network
LTL	Linear Temporal Logic
MBT	Model-Based Testing
RPP	Relational Properties Prover
RTE	Run-time Error
SD	Sequence Diagram
SDL	Specification and Description Language
SE	Symbolic Execution
STS	Symbolic Transition System
UML	Unified Modeling Language
WAN	Wide Area Network
WP	Weakest Precondition
xLIA	eXecutable Language for Interaction and Architecture

Chapter 7

Conclusion

The work presented in this report shows a first step towards a complete toolchain to express property over a high-level model and have them checked on the concrete implementation at C level. First experiments have been conducted on the 6LowPAN use case of WP5 and are quite promising. It is also a good demonstration of the advantages of Frama-C's modular design and the use of ACSL as a common base for all plug-ins to exchange information, as the generation of ACSL annotations by RPP implies that it is then easy to directly re-use existing analysis methodologies (symbolized by WP and StaDy) for the verification of relational properties.

During the remainder of this task, we plan to strenghten the link between Diversity and RPP, notably by checking whether it is possible to deal with several scenarios at once in a single relational property. Another important aspect will be to target more complex properties, notably security-related one, expressed within the dedicated UML profile designed in WP1 for Papyrus, and over a larger body of code, directly extracted from the use cases, with as few stubs and assembly code as possible.

Time permitting, other experiments might be tried, in particular designing and implementing Aoraï and/or CaFE prototypes that could be used as target for Diversity, or, on a longer run, for Papyrus itself.

Finally, depending on the progress of external developments regarding the assembly/C correspondance (see section 6.2), it might be possible to replace the manually written C implementation that is currently used in our experiments by C code automatically generated from the actual assembly used by 6LowPAN. This is however a secondary objective from T3.1's perspective whose primary goal remain the generation of lower level specifications (RPP, Aoraï or plain ACSL inputs) from higher level model (sequence diagrams), regardless on the verification tasks that are performed afterwards on the implementation.

Bibliography

[1] Rajeev Alur, Kousha Etessami, and P Madhusudan. A temporal logic of nested calls and returns. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 467–481. Springer, 2004.

[2] Boutheina Bannour, Jose Pablo Escobedo, Christophe Gaston, Pascale Le Gall, and Gabriel Pedroza. Security weaknesses detection by symbolic analysis of scenarios. In *21st Asia-Pacific Software Engineering Conference, APSEC*, pages 367–374. IEEE, 2014.

[3] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[4] Patrick Baudin, François Bobot, Loïc Correnson, and Zaynah Dargaye. *WP Plugin Manual v1.0*, 2016.

[5] Patrick Baudin, Pascal Cuoq, Jean C. Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. `http://frama-c.com/acsl.html`.

[6] BINSEC: Formal Methods for Binary Code Analysis. `https://binsec.github.io/`.

[7] Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, Virgile Prevosto, and Guillaume Petiot. Static and Dynamic Verification of Relational Properties on Self-Composed C Code. In *Test and Proofs (TAP)*, 2018.

[8] Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, and Virgile Prevosto. RPP: automatic proof of relational properties by self-composition. In *TACAS*, volume 10205 of *LNCS*, pages 391–397, 2017.

[9] Bernard Botella, Mickaël Delahaye, Stéphane Hong Tuan Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams. Automating structural testing of C programs: Experience with PathCrawler. In *AST*, 2009.

[10] Imen Boudhiba, Christophe Gaston, Pascale Le Gall, and Virgile Prevosto. Model-based Testing from Input Output Symbolic Transition Systems Enriched by Program Calls and Contracts. In *Proceedings of ICTSS*, 2015.

[11] L. Correnson and J. Signoles. Combining analyses for C program verification. In *the 17th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2012)*, August 2012.

[12] Steven de Oliveira, Virgile Prevosto, and Saddek Bensalem. CaFE: un model-checker colaboratif. In *Approches Formelles pour le Développement de Logiciels (AFADL)*, 2018.

[13] Julien Deltour, Alain Faivre, Emmanuel Gaudin, and Arnault Lapitre. Model-based testing: An approach with sdl/rtds and diversity. In *System Analysis and Modeling: Models and Reusability*, volume 8769 of *LNCS*, pages 198–206, Cham, 2014.

[14] Eclipse Formal Modeling Project. `https://projects.eclipse.org/proposals/eclipse-formal-modeling-project`.

[15] Eclipse Modeling Project. `https://www.eclipse.org/modeling/`.

[16] Oscar Guillen, Bhargavi Nisarga, Luis Reynoso, and Ralf Brederlow. *Crypto-Bootloader – Secure in-field firmware updates for ultra-low power MCUs*. Texas Instrument, 2015.

[17] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015. `https://frama-c.com`.

[18] Arnaud Mathilde, Bannour Boutheina, and Lapitre Arnault. An illustrative use case of the diversity platform based on uml interaction scenarios. *Electron. Notes Theor. Comput. Sci.*, pages 21–34, 2016.

[19] Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov, and Julien Signoles. Instrumentation of annotated C programs for test generation. In *International Working Conference on Source Code Analysis and Manipulation, (SCAM)*, pages 105–114. IEEE, 2014.

[20] Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. How test generation helps software specification and deductive verification in Frama-C. In *TAP*, 2014.

[21] Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. StaDy: Deep Integration of Static and Dynamic Analysis in Frama-C. Technical report, 2014. `http://hal.archives-ouvertes.fr/hal-00992159`.

[22] A. Pnueli. The temporal logic of programs. *Proc. 18th IEEE Symp. on Foundations of Computer Science (1977)*, pages 46–77, 1977.

[23] Julien Signoles. *E-ACSL: Executable ANSI/ISO C Specification Language*, 2012. `http://frama-c.com/download/e-acsl/e-acsl.pdf`.

[24] Nicolas Stouls and Virgile Prevosto. *Aoraï Plugin Tutorial*. `http://frama-c.com/aorai.html`.