Static Flow Analysis of the Action Language for Foundational UML

Jean Malm, Federico Ciccozzi, Jan Gustafsson, Björn Lisper, Jonas Skoog*
School of Innovation, Design and Engineering – Mälardalen University
Västerås, Sweden
[name.surname]@mdh.se, *jonas.skoog.1992@gmail.com

***Abstract*—One of the major advantages of Model-Driven Engineering is the possibility to early assess crucial system properties, in order to identify issues that are easier and cheaper to solve at design level than at code level. An example of such a property is the timing behaviour of a real-time application, where an early indication that the timing constraints might not be met can help avoiding costly re-designs late in the development process.**

In this paper we provide a model-driven round-trip transformation chain for (i) applying a flow analysis to executable models described in terms of the Action Language for Foundational UML (Alf), and (ii) back-propagating analysis results to Alf models for further investigation. Alf models are transformed into the input format for an analysis tool that identifies flow facts, i.e., information about loop bounds and infeasible paths in the model. Flow facts can be used, for instance, when estimating the worst-case execution time for the analysed model. We evaluated the approach through a set of benchmark models of various size and complexity.

***Index Terms*—UML, Alf, model-based analysis, timing analysis, flow facts, SWEET, model transformation, back-propagation**

I. Introduction

Modelling languages are often used to describe the design of a software system under development. Besides showing the system at different levels of abstraction and through different viewpoints, models can also be analysed, to ensure that the resulting system will perform as expected [33], and used for automating the generation of code. In Model-Driven Engineering (MDE), models are the core engineering actors, which are used for multiple goals. If a modelling language is provided with a well-defined execution semantics, it is possible to achieve fully executable models, which can be simulated, analysed, executed, and from which even complete implementations can be automatically derived. To graphically design a detailed executable model can be cumbersome, therefore it is not uncommon for executable modelling languages to be supported by a so called action language, which is a textual notation to describe behaviours according to the language's execution semantics.

This is the case of the Action Language for Foundational UML[1] (Alf) defined by the Object Management Group to act as the surface notation for specifying executable behaviours within a wider model that is primarily represented using the graphical notations of the Unified Modeling Language (UML). Alf naturally leverages the UML metamodelling concepts and thereby can boost consistency-by-construction and ease model-based activities (e.g., analysis [11], simulation [9]). Indeed, one of the main benefits of executable models is that they disclose the opportunity of performing early, yet precise, analysis and verification of the modelled system. Late (i.e., code level) discovery of issues concerning system properties can lead to modifications that can cost 40 times more than if done at design phase [14]. Software developers spend around 50% of their development time on verifying program code, to an estimated total annual cost of $300 billion globally [5]. Early analysis and verification are therefore essential to cope with the exponential growth of modern software complexity and its effective development.

In this paper we focus on early analysis, more specifically static *flow analysis* of executable software models defined with UML and its action language Alf. A flow analysis identifies *flow facts*, i.e., constraints on the control flow of the system such as bounds for the number of loop iterations, or infeasible paths. These flow facts can then be used when estimating the *Worst-Case Execution Time* (WCET) [13], and to detect possible performance bottlenecks in the system (e.g., loops with a high number of possible iterations). Flow and WCET analyses are especially valuable for real-time and safety-critical systems, where correct timing is key. Static flow and WCET analyses offer the possibility to obtain early, approximate timing estimates on model level, before there is any code that can be run. This can help detecting timing problems early, thus avoiding late and costly system redesigns.

Besides WCET analysis, the results of a flow analysis can be used to enhance the designer's understanding of the model and its properties. For instance, flow analysis can discover dead code (or, better, behavioural paths) and possibly non-terminating loops.

In this paper, we present a round-trip model-based approach for static flow analysis of Alf executable models[2]. We provide an approach able to:

- Transform an Alf model to a format suitable for flow analysis.
- Perform the flow analysis using the SWEdish Execution Time tool[3] (SWEET), which can compute a wide variety of potentially very precise flow facts.
- Round-trip in terms of mapping the analysis results back to the Alf model for further investigation.

The remainder of the paper is organised as follows. Section II introduces executable models and static flow analysis. Section III reviews the literature related to our contribution, which in turn is described in Section IV. The paper is concluded by an evaluation in Section V and a summary of the presented contribution and future work in Section VI.

[1]http://www.omg.org/spec/ALF/

[2]Note that this paper is an extended version of the master's thesis of two of the paper's co-authors; see [26].

[3]http://www.mrtc.mdh.se/projects/wcet/sweet/

II. Background

In this section we provide a brief introduction of the main concepts related to our contribution: executable models and static program flow analysis.

A. Executable UML models and Alf

UML is the de-facto standard in industrial model-based development of software systems [22], and, more generally, empirically shown to be the most widely used architectural description language [25]. The standardisation of (i) the Foundational Subset For Executable UML Models (fUML), which gives a precise execution semantics to a subset of UML limited to composite structures, classes and activities (application models designed with fUML are executable by definition) [34], and (ii) a textual action language, Alf, to express complex execution behaviours, has made UML a full-fledged implementation quality language [32].

Alf is a textual surface representation for UML modelling elements, whose execution semantics is given by mapping Alf's concrete syntax to the abstract syntax of fUML. The primary goal of Alf is to act as the surface notation for specifying executable behaviours within a model represented using the usual graphical notations of UML. Alf comes with an extended notation to represent structural modelling elements too. That is to say, it is possible to describe a UML model entirely using Alf. According to its specification, Alf has the following three levels of *syntactic conformance*:

- **Minimum conformance:** includes a subset of Alf for writing textual action language snippets in a graphical UML model, with the capabilities available in a traditional, procedural programming language;
- **Full conformance:** provides a complete action language for representing both behaviour and (partially) structure;
- **Extended conformance:** covers all Alf syntax.

In this work we focus on the following subset of the full conformance, which allows to describe a full-fledged functional model both structurally and behaviourally with Alf[4]:

- **Active Classes.** One active class that defines classes and functions in its body and using the *do*-block to represent the program entry-point;
- **Classes.** Classes support member variables, functions, and instantiations;
- **Basic Data Types.** The allowed basic data types are: Integer, Natural, Boolean, and Sequence;
- **Methods.** Supports all allowed data types as parameters as well as return value;
- **Arithmetic Expressions.** Standard arithmetic expressions with operands: addition, subtraction, multiplication, division, and modulus;
- **Boolean Expressions.** Boolean expressions with operands. Included operators are: AND, OR, NOT;
- **Relational Expressions.** Relational expressions on Integers and Naturals. Included operators are: LT, LEQ, GT, GEQ, EQ, NEQ;
- **Assignment Expressions.** Standard assignment of variables. Allowed assignments are: '=', '++', and '−−';
- **Sequences.** Creation of sequences and access to elements in sequences;
- **Branching Statements.** `if-else`, `while`, and `for` statements.

This subset has been identified by investigating the needs of existing industrial applications, in two different domains (telecom and factory automation), that we previously reverse-engineered and modelled with (f)UML and Alf: the AAL2 protocol in telecom and the self-orienting carrier robot in factory automation [6], [12].

In Listing 1 we can see an example of an Alf textual model including both structure and behaviour.

```
active class Controller{
  protected op(in b : Integer) {
    if (Robot.right.X == 0)
      Robot.left.vecRotateLeft(b);
   }
}
```

Listing 1: Example of ALF syntax

Alf has three prescribed ways to achieve *semantic conformance*, meaning how execution semantics is implemented, summarised as follows:

- **Interpretive execution:** Alf is directly interpreted and executed;
- **Compilative execution:** Alf is translated into a UML model conforming to the fUML subset and executed on the actual target platform according to the semantics specified in the fUML specification;
- **Translational execution:** Alf, as well as any surrounding UML concept in the model, is translated into an executable for a non-UML target platform and executed.

In this work we target translational execution towards an analysis tool, SWEET.

B. Static Program Flow Analysis and SWEET

Static program analysis means to analyse code for some interesting property, given a formal semantics for the code, rather than running it. The code can for instance define a model: static analysis can thus be applied already at model level to pinpoint possible problems.

In this work we focus on static *flow analysis*, applied to executable models. Such an analysis finds constraints on the execution flow, so-called *flow facts*. Examples include loop iteration bounds, and infeasible path constraints. Existing methods for program flow analysis work on quite a low level, and they target C or even machine code. They work best for code found in safety-critical applications with static memory allocation, without recursion, where the program flow is decided by arithmetic conditions.

Flow facts are usually expressed in a control-flow graph representation of the code, where the graph nodes are basic blocks. Each basic block `B` can be assigned a virtual counter variable `#B` that counts the number of executions of `B`. Flow facts can now be expressed as arithmetic constraints on the final values of these variables. For instance, if `H` is the header node for a loop, then the constraint $\#H \leq 100$ constrains the number of loop iterations to at most 100. Similarly the constraint $\#A + \#B \leq 100$ expresses that the basic blocks `A` and `B` can be executed together at most 100 times, which is a kind of infeasible path constraint.

Flow facts are primarily used in WCET analysis [13], as part of the timing analysis typically performed for safety-critical real-time systems. The WCET of a program is defined as its

[4]Note that structural concepts, such as classes and methods, can be described in terms of UML graphical diagrams, but for simplicity we use Alf only.

longest possible execution time when running uninterrupted on some given hardware. Given strong enough flow facts, and a timing model for the underlying hardware, WCET estimates can be computed. In the so-called IPET model [23], the WCET estimation is cast as an integer linear programming problem where a linear cost function, formed from the local execution time bounds for the basic blocks, is maximized subject to the linear constraints given by the flow facts. The IPET model prevails today due to its generality and flexibility.

A potential use of model-level flow facts is to compute approximate WCET estimates on the model level, using coarse estimates for the basic block execution times, in order to pinpoint possible timing problems early. This is an interesting topic for further research. Flow facts can also be used to identify dead code and infeasible paths. This information is interesting for instance when calculating coverage metrics. In addition, flow facts can help the developer understand the code and its properties better: for instance, unexpected flow facts may be due to some bug in the code.

We have used the SWEET flow analysis tool [24] to perform static flow analysis on executable models. SWEET is arguably one of the most precise flow analysis tools around [36], and it can compute a wide variety of flow facts from simple loop iteration bounds to more complex infeasible path constraints. It uses a variant of abstract interpretation called *abstract execution*, where the program is executed with *abstract states*. In these states numeric variables hold intervals rather than single numbers: thus, executing a statement with an abstract state represents several concrete executions, and the paths of abstract executions will always cover all concrete execution paths. Abstract execution is reminiscent of symbolic execution, but offers the option to merge different paths in order to avoid a path explosion [18]. This feature is important to obtain scalability of the analysis, at the cost of some loss of precision. SWEET can also compute approximate WCET estimates using simple timing models. In addition SWEET can perform some other analyses, including static backwards program slicing, and a value analysis to find bounds for the possible values of variables at different program points.

SWEET analyses a format that is also called ALF (ARTIST2 Language for Flow Analysis); to avoid confusion with UML's Alf, we will refer to it as the *SWEET language*. Other formats can be analysed by translation into this language: currently, translators from C as well as some binary formats exist. The SWEET language is similar to a compiler intermediate format, and is designed to be able to represent both source code (C level) and binary code faithfully. It has procedures, a `store` statement for assignments, a `load` instruction to read from memory, a rich set of arithmetic-logical operators, and a `switch` statement to model different kinds of control statements, like if-then-else and conditional jumps. Data can be of different kinds, such as numerical, or addresses to data or code.

The SWEET language has *labels* that mark positions in the code. Code addresses are simply labels. The data memory model is based on *frames*, which represent non-overlapping memory areas. Each data address consists of a symbolic *base pointer* to a frame, and a numerical *offset* within the frame. Frames can represent, e,g., single variables, structs, arrays, objects, address tables, or low-level memory areas. Frames can be statically allocated, corresponding to global data. They can also be allocated within a so-called "`scope`" construct, corresponding to local, stack-allocated data. Finally frames can be allocated dynamically, corresponding to heap-allocated data, although the current version of SWEET does not handle this construct. See [16], [17] for details.

III. Related Work

Model-based analysis has been applied to UML before. The general approach is to either combine and analyse information from different model diagrams directly [37], or to transform them and take advantage of existing tools [27], [4].

In [20], the authors provide an approach for statically analysing UML use cases for assessing requirements engineering outcomes. Dynamic analysis through simulation or other execution means has been applied to UML too. For instance, in [3], the authors reverse engineer Java to fUML for performing different kinds of dynamic analysis on existing (Java) software.

Related to flow and timing analysis, in [29], the authors present a technique and a tool for model-checking operational (design level) UML models based on a mapping to a model of communicating extended timed automata. They consider structural and behavioural UML, but could not take fUML or Alf into account, since they had not been introduced yet. The lack of analysis techniques for action semantics of UML was partly addressed in [30], where the authors present a technique based on the static analysis of the dependencies between the different UML actions included in the behavioural schema. Nevertheless, the focus of that work was on consistency of action semantics specification; moreover, fUML and Alf were not introduced yet.

In [38], the authors describe another static analysis approach transforming a UML class model into a static model of behaviour, called a Snapshot Model, whose constraints can be verified using tools such as USE and OCLE. Planas et al. [31] present a lightweight and static verification technique to assess the executability (strong, weak, non-executable) of operations in executable UML models. The goal is to check the executability of operations without breaking the integrity of the structural model.

Baldovin et al. [2] transform a UML-MARTE model into to a specific textual format in order to be fed to a tool for WCET analysis. fUML and Alf are not considered. Another interesting work on WCET analysis on UML-MARTE is described in [15] and uses the Time Transition System (TTS) as verification language on models described with UML activities and composite structures and MARTE. The approach does not entail fUML nor Alf. In [28], the authors provide a translation from Alf to UPPAAL for model checking rather than static flow analysis. There is a prototype implementation of WCET analysis for specifically annotated UPPAAL models [19]. Nevertheless, this prototype does not scale and can only be used for small test examples. Ciccozzi et al. [8] perform timing analysis on C++ code automatically generated from Alf models. However, the timing analysis is done at code level, while in this work we focus on early analysis at model level (hardware-agnostic).

To the best of our knowledge, there is no approach dealing with static flow analysis of Alf (thereby according to the standard execution semantics of UML defined by fUML). This is supported by the results of a systematic review on execution of UML models that we have performed [10].

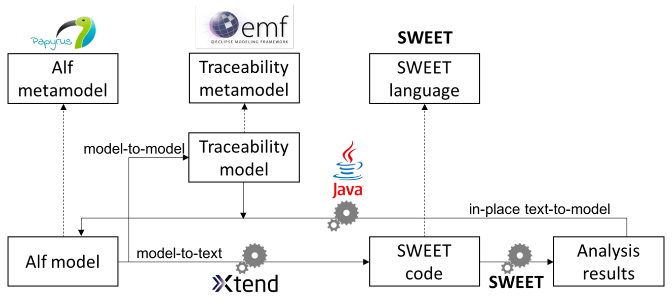

Fig. 1: Solution's architecture and workflow

IV. Flow analysis of Alf

We provide an automated solution for analysing Alf models in terms of their execution (data- and control-) flow. Fig. 1 depicts the architecture and workflow of the solution, which has been implemented as a set of Eclipse plugins.

Starting from an Alf model conforming to the Alf metamodel implementation provided in Eclipse Papyrus[5], an exogenous model-to-text transformation defined in Xtend[6] translates the Alf model into the SWEET's input format, i.e. a program conforming to the SWEET language's EBNF grammar.

Once the analysis terminates, the approach propagates analysis results back to the Alf model. To correctly map analysis results back to the specific model elements in the Alf model, we need to provide explicit trace links between Alf elements and corresponding generated SWEET code elements. For this purpose we specifically designed a traceability metamodel in EMF/Ecore[7], similarly to what we did in previous efforts [8]. A dedicated set of Xtend exogenous model-to-model transformation rules generate an instance of this metamodel – i.e. a traceability model – when generating SWEET code.

SWEET code can then be analysed through the SWEET tool, which generates a results log. At this point, in order for the modeller to be able to grasp at a glance the results of the analysis, they are back-propagated to the Alf model as structured lexical comments (using the `LexicalComment` concept provided by Alf). The back-propagation is carried out by an exogenous in-place text-to-model transformation defined in Java, which takes as input the analysis results and the traceability model previously generated, similarly to [8].

In the remainder of this section we describe in detail each step providing explanatory examples too.

A. *Modelling with Alf and transformation to SWEET language*

We target executable models defined in UML and Alf, and thereby adhere to the execution semantics brought by fUML. As aforementioned we exploit structural and behavioural modelling capabilities offered by Alf. In Listing 2 we introduce a compact version of one of the applications used for validation, as a running example. The Alf model represents *CollisionTest*, a software application that tests overlap between boxes in a 2D-space. A box is represented using its location, width and height. The application creates a sequence of boxes, and then tests all possible box pairs for overlaps by varying the value of the boxes' width. At the end of the execution, the `collisions` variable should hold the number of identified potential collisions.

```
active class CollisionTest {
  class HitBox {
    /* X, Y, height and width, all Integers */
    public Colliding(in o : HitBox) : Boolean {
      return ((this.X < o.X + o.w) && (this.X +
          this.w > o.X) && (this.Y < o.Y + o.h)
          && (this.Y + this.h > o.Y));}
    }
  }
do {
  /* Init code for b1-b4 omitted */
  let boxes : HitBox[] = HitBox[]{b1, b2, b3, b4};
  let i : Integer = 1; let j : Integer = 1;
  let collisions : Integer = 0;
  while(i < numOfBoxes) {
    j = i + 1;
    while(j <= numOfBoxes) {
      if(boxes[i].Colliding(boxes[j]))
        collisions++;
      j++;
    }
    i++;
  }
}
```

Listing 2: Running example in Alf

We currently support the Alf syntactical concepts listed in Section II-A.

Given an Alf model defined using a combination of those concepts, an Xtend model-to-text transformation automatically translates it into SWEET code. In the following subsections, we describe how the entailed Alf syntactical concepts are transformed to the SWEET language, referring to the running example. Alf and the SWEET language are placed at different abstraction levels. Being a high-level language, Alf abstracts away many details that are needed when performing our analyses; examples are the separation between big and little endian, or the specification of a fixed size for basic data types. These details are inferred by the model transformations when translating Alf to SWEET. As the SWEET language is quite verbose, we use some "macro instructions" in the following SWEET language snippets. These are specifically defined for this paper[8], and described in Table I. Let us see how the various Alf concepts are translated to the SWEET language.

a) Active Classes: We chose to support *active classes* as the topmost node element in the Alf model. This is in order to be able to execute Alf code in isolation using tools[9] implementing the Alf specification, and at the same time modelling a fully executable program, which is required by the analysis tool. Active classes consist of a declaration block and a behavioural block. The declaration block holds definitions of fields and additional classes, whereas the behavioural block contains functional code, which in this work is considered the entry point for the application.

An active class is transformed by going through declaration and behavioural blocks and translating them to SWEET (see Table II). Regarding declarations, classes are translated as described in the next section. Fields are transformed into SWEET global variables. As the SWEET input language separates the declaration of functions from their definitions,

[5]https://www.eclipse.org/papyrus/

[6]https://www.eclipse.org/xtend/

[7]https://www.eclipse.org/modeling/emf/

[8]For the interested reader, more info of SWEET language macros can be found in http://www.mrtc.mdh.se/Alf2SWEET.

[9]We used the implementation provided at: https://github.com/ModelDriven/Alf-Reference-Implementation.

TABLE I: SWEET code macros used in the description of the mappings

Macro	Explanation
`label_def` <lblName>	Defines an address label in the code, which can be used as a target for jump-type instructions.
`alloc_v` <size> <name>	Wrapper for function variable definition.
`load_inst` <size> <fName> <offset>	Loads amount of memory given by <size> from the memory address given by frame <fName> and <offset>.
`add_inst` <size> <op_1> <op_2>	Wrapper for addition instruction $op_1 + op_2$ of size <size>.
`addr_inst` <fName> <offset>	Evaluates to the address given by the frame <fName> and <offset>. Normally used in combination with storing and loading data.
`target_label` <lName>	Evaluates to the address of the label given by <lName>. Needed for jumps and function calls.

TABLE II: Translation of the Active Class and Class concepts

Alf	SWEET language
`active class` CollisionTest{ ... `class` HitBox { /* fields */ Colliding(other : HitBox) {<code>} } }	{ `alf` ... /* method declaration */ { `exports` ... { `lrefs` ... { `lref` 64 "HitBox.%baseConstructor" } { `lref` 64 "HitBox.Colliding" } } ⋮ { `funcs` { `func` { `label_def` "HitBox.Colliding" } { `arg_decls` { `alloc_v` "%this" 64 } { `alloc_v` "other" 64 } } ⋮ { `stmts` <code here> } } /* end func */ } /* end funcs */ } /* end program */

they are transformed separately. The behavioural block, once all statements and expressions are translated to SWEET (as shown later in the section), becomes the entry function of the analysis, and is therefore transformed into a SWEET function named "main".

b) Classes: An Alf class (or better, class instances) is translated in SWEET as a single memory frame, with a base address and the offset of each field stored in a lookup table[10]. Instantiation is done by calling a default function, which sets all fields to default values. Methods of the Alf class are transformed into SWEET functions, which take a pointer to the calling object as an extra argument.

In Table II we can see (an extract of) the transformation result from the Alf class `CollisionTest::HitBox` to SWEET. Function labels are defined for both a default constructor (“Hitbox.%baseConstructor” in SWEET), which is called when `HitBox` objects are created, and the `Colliding` member method (“Hitbox.Colliding” in SWEET). We can also see the translation of the definition of `Colliding` and its arguments.

[10]It is worth noting that fields representing class objects are stored using composition, i.e. the entire object is stored inside the allocated memory, not only its reference.

TABLE III: Translation of expressions

Alf	SWEET language
/*`this` is left of `o`*/ (`this`.X < o.X + o.width) && <operand2>	{ `and` /*&&*/ { `s_lt` /* < */ { `load_inst` <size> "this" <offset for X> } /* this.X */ { `add_inst` <size> { `load_inst` <size> "o" <offset for X> } { `load_inst` <size> "o" <offset for width> } } /* end add_inst */ } /* end s_lt */ { <transformed operand2>} } /* end and */

TABLE IV: Translation of assignments

Alf	SWEET language
j = i + 1	{ `store` { <address of j> } `with` { <translation of i + 1> } }

c) Basic Data Types: The SWEET language has numeric types that differ between signed/unsigned and in size. Alf's Boolean is transformed into an unsigned 1-bit value (e.g. `true` becomes { `dec_unsigned` 1 1 }). Natural becomes an unsigned type, while Integer a signed, and their size (the number after the sign) can be any positive number in bits.

d) Expressions: Alf expressions can be formed using constants or qualified names, representing for instances variables, in combination with unary or binary operators. In this category we address Alf arithmetic, boolean and relational expressions. The translation of a compound expression from the `Colliding` method is shown in Table III. Alf and SWEET natively support the same operators, so their translation is rather straightforward. Nevertheless, in SWEET we need to compute additional information, as for instance a variable's location in memory (e.g. `load_inst` <size> "this" <offset for X> in SWEET, representing `this`.X in Alf), as well as store and load of variable values. For compound expressions, we transform first the operands (possibly sub-expressions) and then apply the matching operator, as shown in the table.

e) Assignment Expressions: Assigning to a variable requires the target location address to be computed, transforming the expression computing the value to be stored and then storing it in the memory location. Table IV shows an example of assignment from the running example.

f) Methods: The translation of definition and call to an Alf function into SWEET is shown in Table V.

The definition of an Alf method is translated as follows. Each function gets a unique name label. The input parameters, as well as the calling object, are stored as a list of $(name, type)$ values. This list is transformed and inserted into the function header, which specifies name and size of all arguments and local variables (e.g. { `arg_decls` { `alloc_v` "%this" <refSize> } ...). The method body is then transformed and inserted in the statement part of the function in SWEET. SWEET functions must have explicit return statements, so, if not provided, the transformation generates one.

A call to an Alf function is translated as follows. Arguments are transformed and provided as arguments to the SWEET

TABLE V: Translation of methods

Alf	SWEET language
Colliding(in o: Hitbox) { /* code here */ }	{ func { label_def "HitBox.Colliding" } { arg_decls { alloc_v "%this" <refSize> } { alloc_v "o" <refSize>} } /* reserve memory space for local variables */ { stmts <code here> } } /* end func */
res = b1.Colliding(b2)	{ call { target_lbl "HitBox.Colliding" } { addr_inst "b1" } { addr_inst "b2" } /* args list */ result { addr_inst "res" } /* return value address */ } /* end call */

TABLE VI: Translation of sequences. Note that the first store instruction is used to store the size of the sequence in the first element index.

Alf	SWEET language
/* in CollisionTest do-block */ /* b1-b4 are HitBox objects */ let boxes : HitBox[] = HitBox[]{b1, b2, b3, b4}	/* in "main" func header */ { decls { alloc_v "b1" <HitBox size> } ⋮ { alloc_v "boxes" <(count + 1) * address size> } } /* End decls */ { stmts /* In function body */ { store { addr_inst "boxes" 0 } with { dec_unsigned <size> 4 } } /* boxes[0] ← 4 */ { store {addr_inst "boxes" <index 1 offset> } with { addr_inst "b1" 0 } } /* boxes[1] ← address to b1 */ ⋮ { store { addr_inst "boxes" <index 4 offset> } with { addr_inst "b4" 0 } } /* boxes[4] ← address to b4 */ } /* end of stmts */

call instruction. SWEET uses *call-by-value*, so when arguments represent either class objects or sequences, their location address in memory (e.g., addr_inst “b2”) is passed instead of the actual object (e.g., “b2”). The location to store the result is also explicitly stated (e.g. result { addr_inst "res" }).

g) Sequences: Sequences are collection types, representing an ordered list of elements. In Alf, sequences are created in two ways: with a concrete list of elements (as in the running example) or with an inclusive range between two integer values (e.g. Integer[]{1..5}). The total size of a variable needs to be defined in the function's header, so the range needs to be statically computable at transformation time. Much like class instances, the sequence is represented as a memory frame large enough to hold all elements as well as the sequence length, stored at the beginning of the frame.

Unlike local function sequences, sequence class fields are stored as references, so, when indexing sequences through a class object, extra dereferencing needs to be done in SWEET. Class objects in sequences are also stored as references.

In Table VI, we can see the translation of the initialization of a sequence in the main function of the running example.

TABLE VII: Translation of conditional branches

Alf	SWEET language
while(i<numOfBoxes){ /* Loop body */ }	{ label_def "main.WhileStmt_0" } { switch { s_lt <operands size> { load_inst "main.i" } { load_inst "main.numOfBoxes" } } { target { dec_signed 1 { minus 1 } } { target_lbl "main.WhileStmt_0_start" } } { default { target_lbl "main.WhileStmt_0_end" } } } { label_def "main.WhileStmt_0_start" } /* loop body code here */ { jump_inst { target_lbl "main.WhileStmt_0" } } { label_def "main.WhileStmt_0_end" }

h) Branching statements: Alf statements for branching, such as for-, while- and if-statements, are translated by combining SWEET's conditional and unconditional jump instructions. Table VII shows the outer while-statement in the running example's active class behaviour and its translation to SWEET.

They are translated into a switch statement, where a conditional expression is evaluated and based on the result the execution jumps to another memory location. Jumping requires a target label, so these are generated (e.g., { label_def "main.WhileStmt_0" }). If the expression is evaluated to true, execution should move to the conditional block (e.g., { target ... { target_lbl "main.WhileStmt_0_start" } }). If it is false, it should skip the block (e.g., { default { target_lbl "main.WhileStmt_0_end" } }).

Loops require an additional label before the evaluation of the condition, as it should jump back to evaluate the condition again once the conditional block is executed (e.g., jump_inst { target_lbl "main.WhileStmt_0" }).

B. Analysis with SWEET and back-propagation to Alf model

Once the Alf code has been translated, it can be analysed using the SWEET tool. We focused on flow analysis where the generated flow facts provide lower and upper bounds on the number of executions at specific program points. Once the system has been analysed, the results are mapped back to the source model in a format understandable by the modeller.

Each relevant program point and variable in the resulting code is given a unique ID label, based on its type and location in the code. SWEET uses these labels when presenting the resulting flow facts. After the analysis, the results are parsed and back-propagated to the Alf model through an in-place text-to-model transformation. More specifically, ID labels for interesting program points are recomputed and used to look up the results, which are then properly injected into the model, according to the trace links in the traceability model, as structured lexical comments.

The final presentation format can be seen in Listing 3. The number of executions of the while loops is completely dependent on the constant number of boxes, and therefore deterministic. To avoid just having a static configuration of boxes, the width of box B1 has been manually overriden and

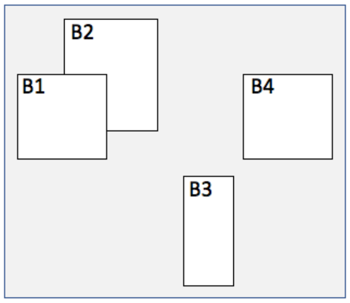

Fig. 2: Placement of boxes in the example

set to unknown in the analysis. This causes it to test all non-negative widths at the same time. In Fig. 2, we can see that, by changing its width only, `B1` would overlap with a maximum of two boxes. This fact is reflected in line 21 of Listing 3. The final step of the tool has inserted some lexical comments stating the number of times that the execution reached the program point. For the running example, we can use this information to verify that the code does what is expected in terms of loop execution, while also verifying the functionality of the `Colliding` function. The analysis found an execution path that never entered the if-statement, as well as one that entered it up to twice, which is what we would expect given the properties of the boxes used.

The benefit of presenting the analysis results as decoration of the Alf model rather than as a separate log file is that the relation between the result and the source is directly visible. When dealing with complex software systems, the possibility to investigate analysis results at model level can be crucial for the engineer to identify and correct possible anomalies in the design that would lead to functional errors or extra-functional issues at code level.

```
1  let collisions : Integer = 0;
2  let i : Integer = 1;
3  let j : Integer = 1;
4  let index : Integer = 1;
5  index = 1;
6  /* Note that the value of widths[1] has been
       overriden in the analysis, in order to test
       all possible values of it. */
7  for(HitBox b : boxes)
8  {
9     /* Flowfacts: Program point passed: 4 times  */
10    b.Set(Xs[index], Ys[index], widths[index],
          heights[index]);
11    index++;
12 }
13 j = 1;
14 i = 1;
15 while(i < numOfBoxes /* numOfBoxes = 4 */) {
16    /* Flowfacts: Program point passed: 3 times */
17    j = i + 1;
18    while(j <= numOfBoxes) {
19       /* Flowfacts: Program point passed: 6 times
            */
20       if(boxes[i].Colliding(boxes[j])) {
21          /* Flowfacts: Program point passed: 0-2
               times */
22          collisions++;
23       }
24       j++;
25    }
26    i++;
27 }
```

Listing 3: Back-propagated analysis results

V. Evaluation

The transformation between Alf and SWEET has been validated through transformation unit testing [35], where actual (generated SWEET code) and expected results were compared. The test programs were designed already from the beginning as a test suite to be used for regression testing, whenever new features (in terms of Alf concepts) were added. The test suite was run on multiple examples of the specific modelled constructs under test.

The approach has been validated by processing a set of Alf models of varying size and complexity[11].

We tested the performance (i.e. in terms of execution time) of the two most demanding operations, that is to say (i) translation of Alf to SWEET and (ii) flow analysis, on the *CollisionTest* application. To test the scalability of the two operations, we considered three versions of the application, composed of 72, 1000 and 10231 LoC respectively. The resulting execution times are shown in Table VIII[12].

The end-to-end execution times (in ms) were recorded using Java's `System.NanoTime` API for the translation from Alf to SWEET, and the built-in `time` command in the Ubuntu subsystem for Windows 10 (Ubuntu 14.04.5 LTS) for the SWEET analysis, averaged over 20 runs each. This was run on a Intel Core i7.7820HQ CPU @ 2.90 GHz processor.

TABLE VIII: Performance of the prototype implementation

Alf LoC	SWEET LoC	Alf $\rightarrow$ SWEET ET	SWEET ET
72	726	6.43	101.29
1000	13310	67.25	2029.76
10231	138891	610.14	68644.62

The current major limitation of the proposed approach is the limited coverage of modelling concepts translated to SWEET. Among them, two relate to most high-level languages and are recursion[13] and dynamic memory allocation. Concerning recursion, since SWEET does not provide support for analysing it, recursive behaviours would need to be rewritten into iterative ones. This is a well-known activity, which entails studying the behaviour, convert recursive calls into tail calls, add a one-shot loop around the entire behaviour's body, and eventually convert tail calls into continue statements. Such an activity could be partially automated too. When it comes to dynamic memory allocation, SWEET does not currently provide out-of-the-box support for it. To entail it without disrupting the analysis itself, a possibility could be to "mimic" dynamic allocation using structures currently available in SWEET, such as allocating a shared memory frame for dynamic data and fill it in during the analysis run.

Additionally, in order for the analysis to produce a result, it needs to terminate so it is not suitable for analysing non-terminating problems.

As hinted by "The Power of Ten – Rules for Developing Safety Critical Code" by NASA [21], recursion, dynamic memory allocation, and non-termination should be avoided

[11]For the interested reader, distributable Alf models, related generated SWEET code, and annotated analysis results are available at http://www.mrtc.mdh.se/Alf2SWEET.

[12]Due to memory optimisations performed by the Eclipse IDE, the first translation performed in a given 'instance' takes between 1200 and 1600 ms longer. As this is a known phenomenon, we excluded it from the sampling.

[13]Although not explicitly covered by either the fUML or the Alf specifications, there are ways to express recursion in UML.

a-priori in real-time safety-critical systems, which represent those benefiting the most from the kinds of analysis targeted by our approach. Additionally, the subset of Alf syntax covered by our transformation chain is expressive enough to model full-fledged industrial applications both in telecom and factory automation domains, as shown in [6], [12].

VI. Conclusion

In this paper we presented a round-trip model-based approach for static flow analysis of Alf executable models.

With our solution we showed that it is feasible to perform static flow analysis on executable Alf models using a translational approach. Moreover, the analysis makes it possible to identify possible anomalies early in the development process, rather than at implementation phase.

As future enhancement, we plan to investigate the possibilities to, on the short term, extend the coverage of entailed Alf concepts, and, on the long term, adapt the SWEET analysis to run on Alf models, properly described, directly. We plan also to enclose the analysable subset of Alf into a specific profile (or similar) for static flow analysis. Regarding the analysis itself, our next step will be to infer timing information in the Alf model in order to be able to use flow facts to derive WCET estimations and identify performance bottlenecks in the modelled system. An interesting option is to attempt the methodology for identifying timing models in [1] for this purpose. Moreover, we plan to investigate how to provide analysis results in Alf models as proper lexical tokens, rather than with lexical comments. A possibility would be to provide something similar to the Alf statement annotation mechanism. Additionally, we are already investigating the possibility to directly compile Alf models to more expressive languages, such as the LLVM intermediate representation, to enable other types of analysis, such as memory access and memory allocation analysis, provided by LLVM and related tools [7].

Acknowledgements

This work is supported by the KK-foundation through the MOMENTUM project[14], and the ITEA3 TESTOMAT[15] project through Vinnova – Sweden's innovation agency.

References

[1] P. Altenbernd, J. Gustafsson, B. Lisper, and F. Stappert. Early execution time-estimation through automatically generated timing models. *Real-Time Systems*, 52(6):731–760, Nov. 2016.

[2] A. Baldovin, A. Zovi, G. Nelissen, and S. Puri. The concerto methodology for model-based development of avionics software. *Lecture Notes in Computer Science*, pages 131–145, 2015.

[3] A. Bergmayr, H. Bruneliere, J. Cabot, J. Garcia, T. Mayerhofer, and M. Wimmer. fREX: fUML-based Reverse Engineering of Executable Behavior for Software Dynamic Analysis. In *Procs of MiSE*, 2016.

[4] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *ERTS 2008*, Toulouse, France, Jan. 2008.

[5] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software. *Univ. Cambridge, Tech. Rep*, 2013.

[6] F. Ciccozzi. On the automated translational execution of the action language for foundational UML. *Software & Systems Modeling*, pages 1–27, 2016.

[7] F. Ciccozzi. UniComp: a semantics-aware model compiler for optimised predictable software. In *Procs of ICSE-NIER*, 2018.

[8] F. Ciccozzi, A. Cicchetti, and M. Sjödin. Round-trip Support for Extra-functional Property Management in Model-driven Engineering of Embedded Systems. *Inf. Softw. Technol.*, 55(6):1085–1100, 2013.

[9] F. Ciccozzi, J. Feljan, J. Carlson, and I. Crnković. Architecture optimization: speed or accuracy? Both! *Software Quality Journal*, pages 1–24, 2016.

[10] F. Ciccozzi, I. Malavolta, and B. Selic. Execution of UML models: a systematic review of research and practice. *Software & Systems Modeling*, pages 1–48, 2018.

[11] F. Ciccozzi, M. Saadatmand, A. Cicchetti, and M. Sjödin. An Automated Round-trip Support Towards Deployment Assessment in Component-based Embedded Systems. In *Procs of CBSE*, 2013.

[12] F. Ciccozzi, T. Seceleanu, D. Corcoran, and D. Scholle. UML-based development of embedded real-time software on multi-core in practice: lessons learned and future perspectives. *IEEE Access*, 4:6528–6540, 2016.

[13] R. W. et al. The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[14] D. Galin. *Software quality assurance: from theory to implementation*. Pearson Education India, 2004.

[15] N. Ge, M. Pantel, and B. Berthomieu. A Flexible WCET Analysis Method for Safety-Critical Real-Time System using UML-MARTE Model Checker. 2016.

[16] J. Gustafsson, A. Ermedahl, and B. Lisper. ALF (ARTIST2 Language for Flow Analysis) specification. Technical report, Mälardalen University, Västerås, Sweden, Jan. 2011.

[17] J. Gustafsson, A. Ermedahl, B. Lisper, C. Sandberg, and L. Källberg. Alf – a language for wcet flow analysis. In N. Holsti, editor, *Procs of WCET*. OCG, June 2009.

[18] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Procs of RTSS*, 2006.

[19] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET Analysis of Multicore Architectures using UPPAAL. In *Procs of WCET*. OGS, 2010.

[20] J. H. Hausmann, R. Heckel, and G. Taentzer. Detection of Conflicting Functional Requirements in a Use Case-driven Approach: A Static Analysis Technique Based on Graph Transformation. In *Procs of ICSE*, 2002.

[21] G. J. Holzmann. The power of 10: rules for developing safety-critical code. *Computer*, 39(6):95–99, 2006.

[22] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical Assessment of MDE in Industry. In *Procs of ICSE*. ACM, 2011.

[23] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, Dec. 1997.

[24] B. Lisper. SWEET – a tool for WCET flow analysis (extended abstract). In *Procs of ISOLA*. Springer-Verlag, 2014.

[25] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Trans. Software Eng.*, 39(6):869–891, 2013.

[26] J. Malm and J. Skoog. Towards automated analysis of executable models. Master's thesis, Mälardalen University, School of Innovation, Design and Engineering, 2017.

[27] P. C. Mehlitz. Trust your model - verifying aerospace system models with java pathfinder. *2008 IEEE Aerospace Conference*, 2008.

[28] Z. Micskei, R.-A. Konnerth, B. Horváth, O. Semeráth, A. Vörös, and D. Varró. On Open Source Tools for Behavioral Modeling and Analysis with fUML and Alf. In *Procs of OSS4MDE*, 2014.

[29] I. Ober, S. Graf, and I. Ober. Validating timed uml models by simulation and verification. *International Journal on Software Tools for Technology Transfer*, 8(2):128–145, Apr 2006.

[30] E. Planas, J. Cabot, and C. Gómez. Verifying Action Semantics Specifications in UML Behavioral Models. In *Procs of CAiSE*, 2009.

[31] E. Planas, J. Cabot, and C. Gomez. Lightweight and static verification of UML executable models. *Computer Languages, Systems & Structures*, 46:66 – 90, 2016.

[32] B. Selic. The Less Well Known UML. *Formal Methods for Model-Driven Engineering*, 7320:1–20, 2012.

[33] I. Sommerville. *Software engineering*. Addison-Wesley, 8 edition, 2006.

[34] J. Tatibouët, A. Cuccuru, S. Gérard, and F. Terrier. Formalizing Execution Semantics of UML Profiles with fUML. In *Procs of MoDELS*. 2014.

[35] A. Tiso, G. Reggio, and M. Leotta. Unit Testing of Model to Text Transformations. In *Procs of AMT*, page 14, 2014.

[36] R. von Hanxleden et al. WCET tool challenge 2011: Report. In *Procs of WCET*, 2011.

[37] R. Wille, M. Gogolla, M. Soeken, M. Kuhlmann, and R. Drechsler. Towards a Generic Verification Methodology for System Models. *Procs of DATE*, 2013.

[38] L. Yu, R. B. France, and I. Ray. Scenario-Based Static Analysis of UML Class Models. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Procs of MODELS*, 2008.

[14]http://www.es.mdh.se/projects/458-MOMENTUM

[15]http://testomatproject.eu/