

SIMPAL: A Compositional Reasoning Framework for Imperative Programs

Lucas Wagner †

Contents

1	Acknowledgements.....	3
2	Introduction	4
3	Background	4
3.1	Lustre	4
3.2	Contract Based Reasoning	7
3.3	AADL, OSATE, and AGREE	8
4	Approach	8
4.1	Limp Specification Language.....	9
4.1.1	Motivating Example	9
4.1.2	Limp Type System	10
4.1.3	Functions.....	10
4.1.4	Local Procedures	11
4.1.5	External Procedures	11
4.1.6	Global Variables	12
4.1.7	Contract Specification	12
4.2	Translation to Lustre	13
4.2.1	Limp to Limb Transformations.....	14
4.2.2	Transforming Finalized Limb to a Control Flow Graph (CFG).....	16
4.2.3	Translating the CFG representation to Lustre.....	18
4.3	Analyses	27
4.3.1	Contract Verification.....	27
4.3.2	Reachability.....	31
4.3.3	Viability	31
5	Limitations.....	34
5.1	String semantics.....	34
5.2	Performance	34
6	Implementation	35
7	References	37

1 Acknowledgements

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

2 Introduction

Software reuse is the practice of using existing software to build new software. There are varying reasons for employing software reuse, many of them owing to the convenience, economy, and recognized service history of pre-existing code. However, reused software components can be viewed as custom designs, specifically engineered and tested to work in a single application. If software is reused in the wrong context, the overall assurance case (reviews, testing, service history, etc) is incomplete; the argument for trust was based on a specific use-case, not an arbitrary one.

This paper introduces a tool built to analyze programs composed of pre-existing existing software components using contract-based reasoning. This tool is called SIMPAL (Static IMPerative Analyzer). It features a domain specific language that can be used to specify a software program that is composed of pre-existing components. Once a program is specified, users can analyze whether a program obeys all of the preconditions required by the reused components, and that the resulting program obeys its intended postconditions. Further, it allows the user the ability to specify exactly how a component will utilize and modify global variables in the analysis. Additional analyses are performed to identify unreachable portions of the CFG, identifying potentially dead-code. Finally results are reported back to the user identifying a sequence of inputs that could violate the various contracts specified in the new program. This information can be used by the program developer to refine the software specification.

3 Background

The SIMPAL framework extends the concept of contract-based reasoning to imperative programs written in the Limp program specification language. Limp is an extension of the Lustre language, inheriting its type system and much of its syntax, with additional constructs added to support standard control flow mechanisms. The AGREE tool implements contract-based reasoning on hardware and software architectures specified in the Architectural Analysis and Design Language (AADL). This is accomplished by creating a Lustre model of the behavioral aspects of an AADL model and analyzing it.

SIMPAL extends this work by introducing extensions to the Lustre language that provide an imperative style program specification. It also performs contract-based verification. The following sections discuss the Lustre language, contract-based verification, and the approach implemented in the AGREE framework.

3.1 Lustre

The Lustre [1] synchronous dataflow language is designed for programming reactive systems. Programs written in the Lustre language continuously interact with their environment; they sample inputs and compute outputs on every time-step. This makes Lustre a natural fit for developing programs for automated control and monitoring, signal processing, and certain safety-critical applications, such as avionics software. Shown in Example 1 is a Lustre program that accepts three values that represent measurements provided from a sensor, and merges them into a single signal. This algorithm, and others like it, could be used to provide a fault tolerant measurements in a safety critical application.

```

1  const RATE : real = 0.05;
2  const MAX_ERROR : real = 0.1;
3
4  node max(a : real; b : real) returns (out : real);
5  let
6      out = if (a < b) then b else a;
7  tel ;
8
9  node min(a : real; b : real) returns (out : real);
10 let
11     out = if (a < b) then a else b;
12 tel ;
13
14 node middleValue(a : real; b : real; c : real) returns (out : real);
15 let
16     out = max(min(a,b), min(max(a,b),c));
17 tel ;
18
19 node saturation(x : real; signal : real) returns (out : real);
20 var
21     upper_limit, lower_limit : real;
22     property1 : bool;
23 let
24     property1 = x > 0.0;
25
26     upper_limit = x;
27     lower_limit = -x;
28
29     out = if (signal < lower_limit)
30         then lower_limit
31         else if (signal > upper_limit)
32             then upper_limit
33             else signal;
34 tel ;
35
36 node voter(a : real; b : real; c : real) returns (out : real);
37 var
38     equalizationA, equalizationB, equalizationC : real;
39     equalizedA, equalizedB, equalizedC : real;
40     centering : real;
41 let
42     equalizedA = a - equalizationA;
43     equalizedB = b - equalizationB;
44     equalizedC = c - equalizationC;
45
46     centering= middleValue(equalizationA, equalizationB, equalizationC);
47
48     out = middleValue(equalizedA, equalizedB, equalizedC);
49
50     equalizationA = 0.0 -> pre equalizationA +
51         (RATE * (saturation(0.5, (equalizedA - out)) - saturation(0.25,centering)));
52     equalizationB = 0.0 -> pre equalizationB +
53         (RATE * (saturation(0.5, (equalizedB - out)) - saturation(0.25,centering)));
54     equalizationC = 0.0 -> pre equalizationC +
55         (RATE * (saturation(0.5, (equalizedC - out)) - saturation(0.25,centering)));
56 tel ;
57
58 node abs_diff(in1 : real; in2 : real) returns (out : real);
59 var
60     diff : real;
61 let

```

```

58     diff = in1 - in2;
59
60     out =
61         if diff < 0.0
62             then -diff
63             else diff;
64 tel ;
65
66 node main(input : real; errorA : real; errorB : real; errorC : real) returns (out :
67 real);
68 var
69     pre_input : real;
70     input_change : real;
71     assert_input1, assert_input2, assert_errorA, assert_errorB, assert_errorC :
72 bool;
73
74     prop1 : bool;
75 let
76     pre_input = 0.0 -> pre input;
77     input_change = input - pre_input;
78
79     assert_input1 = (input >= 0.0 and input <= 20.0);
80     assert_input2 = (input_change >= 0.0 and input_change <= 1.0);
81     assert_errorA = (errorA <= MAX_ERROR) and (errorA >= -MAX_ERROR);
82     assert_errorB = (errorB <= MAX_ERROR) and (errorB >= -MAX_ERROR);
83     assert_errorC = (errorC <= MAX_ERROR) and (errorC >= -MAX_ERROR);
84
85     assert assert_input1;
86     assert assert_input2;
87     assert assert_errorA;
88     assert assert_errorB;
89     assert assert_errorC;
90
91     out = voter(input + errorA, input + errorB, input + errorC);
92     prop1 = abs_diff(out,input) <= (4.0 * MAX_ERROR);
93     --%PROPERTY prop1;
94 tel ;

```

Example 1 - Lustre program for a sensor fusion algorithm

This Lustre program is derived from work on analyzing an industrial triplex voter [2]. The triplex voter is an algorithm used to merge the measurements from three similar sensors into a single value, incorporating features to smooth the fused value to prevent drastic transient fluctuations in the output value. This behavior is defined in the node called **voter**, found on lines 36-53 of Example 1. The node **main** contains additional information that is used to reason about the relationship of the output of the voter and the three inputs. The property, named **prop1** on line 93, defines a proof obligation that says the difference between the output of the voter and the true value being measured is bounded by some constant term.

Example 1 is analyzable using tools developed to analyze Lustre programs. Kind 2 [3] [4] uses bounded model checking [5], k-induction [6], and property directed reachability [7] techniques to prove, or disprove safety properties of Lustre models. JKind is a reimplement of the Kind 2 model checker, developed in Java for portability and integration into tool frameworks.

3.2 Contract Based Reasoning

Contract-based reasoning is a compositional reasoning approach used to analyze systems composed of verified systems. Contract-based reasoning can be useful when a system is too complex to be analyzed as one piece of functionality. In a compositional approach each component of a system is analyzed or verified and the resulting artifacts are used when reasoning about the system as a whole. Figure 1 below shows a simple toy example containing components A, B, and C.

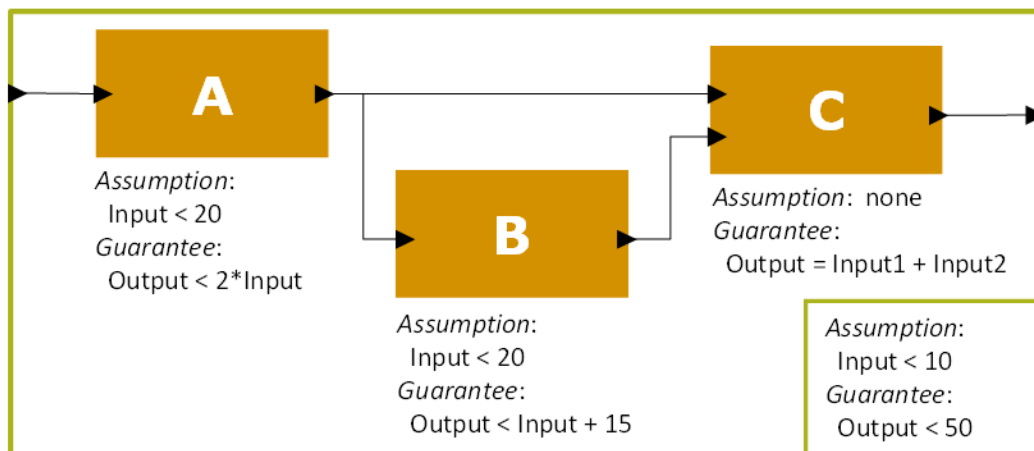


Figure 1 - A Toy Compositional Proof Example

Suppose component A was analyzed using model checking. In that analysis it was established that if the input to A is less than 20, it will always produce an output that is less than two times the input. This analysis reflects assumptions (or preconditions) made on the environment that A operates within and a guarantees (or postconditions) that it provides to that environment. The combination of the assumptions and guarantees associated with a component are collectively referred to as a contract. The contract for components A, B, C, and the system described in Figure 1 are in Table 1.

<u>Component</u>	<u>Assumptions</u>	<u>Guarantees</u>
A	$\text{Input}_A < 20$	$\text{Output}_A < 2 * \text{Input}_A$
B	$\text{Input}_B < 20$	$\text{Output}_B < \text{Input}_B + 15$
C	None	$\text{Output}_C = \text{Input}_{C1} + \text{Input}_{C2}$
System	$\text{Input}_{\text{System}} < 10$	$\text{Output}_{\text{System}} < 50$

Table 1 - Assumptions and Guarantees for the Toy Example in Figure 1

To properly analyze this system it is necessary to verify not only that the system meets its guarantees, but also that it calls each component in a way that satisfy its assumptions. Hence, each component assumption and system guarantee become proof obligations. These proof obligations are colored red in Table 1. However, because the components are verified we can make assumptions about their behavior. We assume each component will meet its stated guarantees. Similarly we can also assume that the system assumptions also hold. These assumptions are colored in blue in Table 1 and are enforced in analysis of the aforementioned proof obligations.

3.3 AADL, OSATE, and AGREE

The architectural analysis and design language (AADL) is a domain specific language used to describe the hardware and software architectural components of a cyber-physical systems. The Open Source AADL2 Tool Environment (OSATE) is an extension of the Eclipse XText framework that parses, validates, and analyzes AADL models. Additionally it is extensible, easily allowing developers to introduce custom analysis tools into the framework.

AGREE is one such extension of the OSATE framework. AGREE defines an annex to the AADL language that allows users to specify behavioral contracts in system architecture models. Each layer of the architecture is then analyzed hierarchically from the bottom-up, verifying that each component correctly calls any subcomponents while preserving its stated guarantees as discussed in Section 3.2

4 Approach

The SIMPAL tool allows users to specify a program constructed from pre-existing software and custom functionality and use formal methods to reason about whether the pre-existing components were correctly used and whether the constructed program behaves as intended. SIMPAL is implemented in Xtext and uses a domain specific language called Limp (a portmanteau of Lustre Imperative) to capture the specification of a new program.

As the user writes the program the editing environment will identify errors (syntax, type-checking, unreferenced variables, incorrect contracts) for the user to fix in real-time. This ensures that the new program will interface correctly with the pre-existing components as they are defined. Once the user has written a program that passes all of the cursory checks the user can then analyze the model using model checking. In this analysis the Limp specification is translated into an equivalent representation in the Lustre language. This Lustre representation is based off of the control flow graph (CFG) of the original Limp specification and is executed as a state machine. Execution of the program is mapped across multiple Lustre time steps, one for each basic block in the CFG. The artifacts of the translation process are shown in Figure 2.

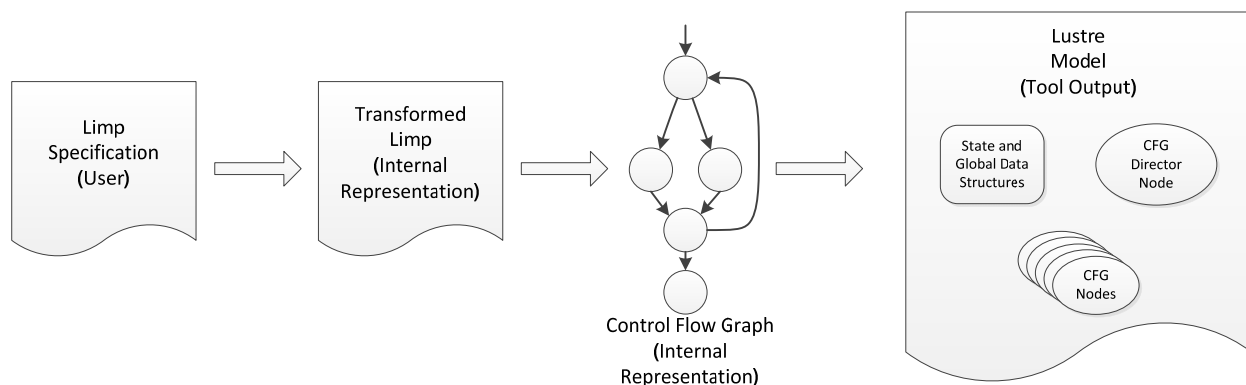


Figure 2 - Analysis internal and external artifacts

System and component contracts are gathered during translation and captured in the final Lustre model. These contracts are used to generate properties to be checked using the JKind [8] model checking tool. Proof violations are reported back to the user as counterexamples, a sequence of input assignments in the program that violate the system and component contracts.

4.1 Limp Specification Language

The Limp language is a domain specific language designed for specifying programs constructed from pre-existing components. Limp was designed to target the Lustre analysis language, yet unlike Lustre, it supports control flow constructs (if, while, for, break, continue, return), imperative execution semantics (allowing variables to be assigned more than once) and a property specification language designed for contract based reasoning.

The full grammar for the Limp language is shown in Appendix A.

4.1.1 Motivating Example

Example 2 shows a Limp specification of a program that accepts a record type named **File** and data to write to it. It returns a **File** record. In the context of this program a **File** is an abstract data object that contains fields open, writes, and data which refer to different characteristics of the file necessary to write to it. There is a constant **MAX_WRITES** which describes how many times a File object can be written as well.

There is an external procedure **writeFile** that accepts a **File** record and data and produces a **File** record that has the data written to the data field. The **writeFile** procedure has one precondition. This precondition **pre1** requires that the file to write to is open. It has a single postcondition **post1** that the file_out **File** record is the same as file_in, except that the writes field has been incremented by 1 and the data in the file is assigned to the data input argument.

Finally the **main** procedure has one input arguments and one output argument. The input is data to be written to the file. The output is a boolean that represents whether or not the file object was written successfully. This procedure modifies a global **File** record named file. The procedure has a single postcondition named **post1** (line 22 of Example 2). It states the procedure will always be successful. In addition the **main** procedure has a precondition, **pre1** (line 21 of Example 2) that expects the file to be open prior to executing the procedure. Together **pre1** and **post1** form the contract for this procedure, stating that if the global variable File is open, the procedure will always end successfully.

1	type record File = {
2	open : bool ,
3	writes : int ,
4	data : int
5	}
6	
7	global file : record File
8	
9	constant MAX_WRITES : int = 10
10	
11	external procedure writeFile(file_in : record File, data : int)
12	returns (file_out : record File)
13	attributes {

14	precondition pre1 = file_in.open;
15	postcondition post1 =
16	file_out == file_in{writes := file_in.writes + 1}{data := data};
17	}
18	
19	procedure main(data : int) returns (success : bool)
20	attributes {
21	precondition pre1 = file.open;
22	postcondition post1 = success;
23	}
24	statements {
25	if (file.open) then {
26	while (file.writes < MAX_WRITES) {
27	file = writeFile(file, data);
28	}
29	success = true;
30	} else {
31	success = false;
32	}
33	}

Example 2 - A simple File writing program

This example highlights several components of a Limp specification. In this case the **main** procedure represents a new software program, while the external procedure **writeFile** represents the pre-existing component to be reused. For the program to work correctly it is necessary to establish that the preconditions on the external procedure **writeFile** are preserved by the system and that the postconditions of the new program are upheld by both the components and glue-code.

4.1.2 Limp Type System

The Limp type system is similar to that of the Lustre language. This was an intentional decision made because the analysis target of Limp specification is the Lustre language and the expressive capabilities of the target language limit the expressive capabilities of the source language. For this reason the Limp type system is a minor extension of the Lustre type system. Types that can be described in Limp are:

- Natives (boolean, integer, real)
- Arrays
- Records
- Strings
- Abstract

The type system is fully compositional and composite types may contain other composite types. String and Abstract types are not present in Lustre and as such we have limited capacity for reasoning over them. During the translation from Limp to Lustre these types are abstracted into simpler representations and semantic information is lost. These transformations are done at the Limp level and are covered in Section 4.2.1.

4.1.3 Functions

The Limp specification language distinguishes between a Function call and a Procedure call. A function is used as a macro; it does not contain state, read or modify global variables, or allow users to specify preconditions, postconditions, uses or defines specifications. Further, functions are only allowed to

return a single value, and are not allowed to assign any variable more than once. Control flow elements such as loops and conditional statements are also not allowed. In Example 3 below, lines 1-4 demonstrate the declaration of a local function that is called on line 21, and line 6 demonstrates the declaration of an external function that is called on line 23.

1	function increment(in1 : int) returns (out : int)
2	equations {
3	out = in1 + 1;
4	}
5	
6	external function decrement(in1 : int) returns (out : int)
7	
8	procedure main (a : bool) returns (x : int)
9	var {
10	q : int ;
11	}
12	attributes {
13	postcondition post1 = x == 10;
14	}
15	statements
16	{
17	q = 0;
18	x = 0;
19	for (q=0;q<10;q=q+1;) {
20	if a then {
21	x = increment(x);
22	} else {
23	x = decrement(x);
24	}
25	}
26	}

Example 3 - A Limp program containing local and external functions

4.1.4 Local Procedures

Local procedures are the main computational element in Limp. They capture the inputs and outputs of the program being specified, the system level preconditions and postconditions, and pre-existing components that are invoked in the program through external procedure calls. Formal analysis is always considered through the entry-point of a local procedure. The inputs, locals and outputs of a local procedure define the state that is tracked in the compositional analysis. The syntax of a local procedure definition can be found on lines 19-33 of Example 2.

4.1.5 External Procedures

External procedures represent a pre-existing program that is invoked via a procedure call from within a local procedure. External procedures can be used to represent a pre-existing computational element. Similar to a local procedure, the user can specify the preconditions and postconditions that the external procedure is expected to satisfy. In addition, the user can specify the global variables (or portions of global variables) that are read and written by the external procedure. This allows for precision in our analysis of programs. The syntax of an external procedure definition can be found on lines 11-17 of Example 2.

4.1.6 Global Variables

Limp supports global variables of any valid Limp type. These variables can be read and written by both external and local procedures.

4.1.7 Contract Specification

Limp allows for the specification of a contract (preconditions and postconditions) for both local and external procedures. Further, it allows for the specification of use/define sets for global variables as well.

4.1.7.1 Local Procedure Contracts

Local procedures are the main computational element in Limp. Formal analysis is performed using a local procedure as the entry point for analysis. Considering this, preconditions and postconditions assigned at the level of a local procedure have a very distinct meaning. Preconditions are assumed by the analysis to be true and postconditions become proof obligations to be checked at termination of the program. In Example 2 the local procedure **main** has one precondition and one postcondition. This means that the analysis assumes that all initial assignments to the global variable **file** will have the field open set to true. At the completion of the program specified we expect the final variable assignments to satisfy the postcondition **post1**. If this is not the case the analysis will identify execution traces that violate each property.

4.1.7.2 External Procedure Contracts

External procedures are used to represent external computation. Like a local procedure, an external procedure has preconditions that its inputs are expected to satisfy and postconditions that its outputs will satisfy. However, when invoked from another program the roles of preconditions and postconditions are reversed. A component postcondition can be assumed to be satisfied by the component (verification that the actual component satisfies the postcondition is up to the provider) but only if its preconditions are met. Therefore all component preconditions become proof obligations.

In Example 2 the external procedure **writeFile** has a precondition **pre1** that must be satisfied by the caller. This means that the global variable **file** (the argument provided to the call to external procedure **writeFile**) must have the field open set to true. The analysis performed by this tool will verify that all preconditions are satisfied from the calling context. When considering the behaviors of the **writeFile** external procedure from within the **main** local procedure, we assume that the component meets its contract and assume that it satisfies its stated postconditions. Therefore when **writeFile** is completed we expect it will return a **File** record with the writes field incremented, the data field written to the value provided to the procedure, and the remaining elements set to the same values as the input File record, per postcondition **post1**.

4.1.7.3 Uses and Defines Specifications

External procedures may interact with global variables and the **uses** and **defines** mechanisms exist to specify this interaction. Limp provides specification constructs to precisely specify how a global is read or written from a local or external procedure. The **uses** construct is used to specify which portions of a global variable can be read by a procedure and similarly the **defines** construct is used to specify which portions of a global variable a procedure can write to. If a global variable is of a composite type (record

or array) the user can precisely specify each element that is used or defined. This allows for precise specification of how globals are read and written by the program. Example 4 shows an alternate version of the program from Example 2, with modifications made to the external procedure `writeFile`, which now utilizes the **uses** and **defines** construct. The syntax of the specification is shown on lines 16-18.

```

1  type record File = {
2      open : bool,
3      writes : int,
4      data : int
5  }
6
7  global file : record File
8
9  constant MAX_WRITES : int = 10
10
11 external procedure alternate_writeFile(data : int) returns ()
12 attributes {
13     precondition pre1 = file.open;
14     postcondition post1 = file.writes == (init file.writes) + 1;
15     postcondition post2 = file.data == data;
16     uses file;
17     defines file.writes;
18     defines file.data;
19 }
20
21 procedure main(data : int) returns (success : bool)
22 attributes {
23     precondition pre1 = file.open;
24     postcondition post1 = success;
25 }
26 statements {
27     if(file.open) then {
28         while(file.writes < MAX_WRITES) {
29             alternate_writeFile(data);
30         }
31         success = true;
32     } else {
33         success = false;
34     }
35 }

```

Example 4 - An alternate version of Example 2

In Example 4 the external procedure **alternate_writeFile** directly writes the global variable **file** rather than it being provided as an input argument. In addition to minor changes to the preconditions and postconditions to reflect this change, there are also **uses** and **defines** specifications. This specification shows that the **alternate_writeFile** reads the global variable **file** (the entire variable) and writes to its **writes** and **data** fields. This information is used in the formal analysis approach to preserve the frame condition; that is, it informs the analysis to only vary the portions of a global variable that an external procedure will change, rather than modifying the entire contents. Without this capability the analysis would require the user to explicitly state the frame condition, which is tedious for large data structures.

4.2 Translation to Lustre

The translation to Lustre is accomplished through a multi-step process. First the Limp specification is transformed to remove unsupported expressions to make the final Limp specification closer in

expressive capability to the Lustre language. Next the transformed Limp specification is converted into a control-flow graph (CFG) representation. Then the CFG representation is simplified through a series of CFG-to-CFG transformations. Finally the simplified CFG representation is translated into a state-machine representation in Lustre. This state-machine executes the Limp program over multiple Lustre steps, executing exactly one node of the CFG on each computation step.

4.2.1 Limp to Limp Transformations

The first part of translating Limp specifications to Lustre is transforming the user-specified Limp to a semantically equivalent form that can be translated into Lustre. The following sections describe each Limp transformation that is used in SIMPAL.

4.2.1.1 Remove Unspecified Constants

The Limp language allows for users to specify constants that are not assigned a literal value. This means that the value of the constant could be any valid assignment for the constant's type, but it does not change for any particular analysis run.

1	constant UNSPECIFIED_CONSTANT : int
2	
3	procedure main() returns (x : int)
4	attributes {
5	postcondition post = x >= UNSPECIFIED_CONSTANT;
6	}
7	statements {
8	x = 0;
9	while (x < UNSPECIFIED_CONSTANT) {
10	x = x + 1;
11	}
12	}

Example 5 - A small program utilizing an Unspecified Constant

Unspecified constants are translated into External Functions with zero inputs. Eventually these are translated into Lustre uninterpreted functions with zero inputs during the translation from the CFG to Lustre. The transformed version of the program found in Example 5 is shown in Example 6 below.

1	external function UNSPECIFIED_CONSTANT () returns (out : int)
2	
3	procedure main () returns (x : int)
4	attributes {
5	postcondition post = x >= UNSPECIFIED_CONSTANT ();
6	} statements
7	{
8	x = 0;
9	while x < UNSPECIFIED_CONSTANT ()
10	{
11	x = x + 1;
12	}
13	}

Example 6 -The program in example 3 after removing unspecified constants

4.2.1.2 Rename Lustre Keywords

Lustre language keywords that appear in Limp specifications would cause conflicts when the user attempts to run the analyzer. This pass renames all conflicting identifiers in a Limp specification.

4.2.1.3 Remove Elselfs

Elself statements are used to avoid deep nesting of If-Then-Else statements. They are simply syntactic sugar provided to the user for convenience. This transformation will remove Elself statements, instead replacing them with nested If-Then-Else statements. This is done to normalize the control flow structure prior to creating the CFG representation.

4.2.1.4 Remove Strings

Strings are not supported by the Lustre language. However the Limp language provides some support for strings. Users can write string literals, and equate two string values, but concatenation, evaluating substrings and similar operations are not supported. Prior to translating to Lustre each string is replaced with a unique integer literal. All operations over strings are replaced with semi-equivalent operations over integers.

1	procedure main(x : string, a : bool) returns (y : string)
2	attributes {
3	precondition pre1 = x <> "ABC";
4	postcondition post = y <> "ABC";
5	}
6	statements {
7	if a then {
8	y = x;
9	} else {
10	y = "ABC";
11	}
12	}

Example 7 - A simple program demonstrating use of strings

Example 7 is transformed into the program found in Example 8.

1	/*
2	Integer to String mapping :
3	0 -> "ABC"
4	*/
5	
6	procedure main (x : int, a : bool) returns (y : int)
7	attributes {
8	precondition pre1_ = x <> 0;
9	postcondition post = y <> 0;
10	} statements {
12	if a then {
13	y = x;
14	} else {
15	y = 0;
16	}
17	}

Example 8 - The transformed version of Example 7

4.2.1.5 Normalize Control Flow

The last step before creating the CFG representation of the Limp specification is to normalize control flow. This means all if-then-else, while, for, break, continue, and return constructs are normalized into a set of labels and goto statements. This “spaghetti-code” is unreadable to the human, but it greatly simplifies creation of the CFG. Example 9 shows a simple program and Example 10 shows its representation in Limp after normalizing control flow.

1	procedure main (a : bool, b : bool) returns (x : int)
2	var {
3	iterations : int;
4	}
5	attributes {
6	postcondition post1 = iterations > 0;
7	}
8	statements
9	{
10	x = 0;
11	iterations = 0;
12	if a then {
13	while(x < 10) {
14	x = x + 1;
15	iterations = iterations + 1;
16	}
17	}
18	}

Example 9 - A sample program utilizing control flow in Limp

1	procedure main (a : bool) returns (x : int)
2	var {
3	iterations : int;
4	}
5	attributes {
6	postcondition post1 = iterations > 0;
7	} statements
8	{
9	x = 0;
10	iterations = 0;
11	goto label_0 when a;
12	goto label_1;
13	
14	label label_0;
15	
16	label label_3;
17	goto label_4 when x < 10;
18	goto label_5;
19	
20	label label_4;
21	x = x + 1;
22	iterations = iterations + 1;
23	goto label_3;
24	
25	label label_5;
26	goto label_2;
27	
28	label label_1;
29	goto label_2;
30	
31	label label_2;
32	
33	label end;
34	}

Example 10 -Example 9 with normalized control flow

4.2.2 Transforming Finalized Limp to a Control Flow Graph (CFG)

The transformed Limp file has normalized control flow. This format contains only label, goto, and assignment statements. From this representation it is straightforward to identify the basic blocks that

make up the CFG nodes, as each basic block begins with an assignment statement and ends at a goto statement. The arcs of the CFG are constructed by following the goto statements.

4.2.2.1 Identifying Basic Blocks

A basic block is defined as a sequence of consecutive instructions that are always executed from beginning to end without branching execution. Basic blocks are determined by walking through the program, beginning a block on an assignment statement, and ending it once a goto statement or label statement is encountered. Once a goto statement is encountered a new basic block begins on the first assignment statement in the resulting label.

4.2.2.2 Constructing the CFG

After the basic blocks of the program are identified, the control flow graph captures execution of the program as set of basic blocks as serve as nodes connected by arcs that dictate their execution order. The CFG is constructed by walking through the normalized Limp specification from top to bottom, connecting the basic blocks via the unconditional and conditional goto statements encountered.

The program in Example 4 is translated into the CFG shown in Figure 3. The basic blocks of the program appear in the nodes of the CFG, labelled 0 – 6.

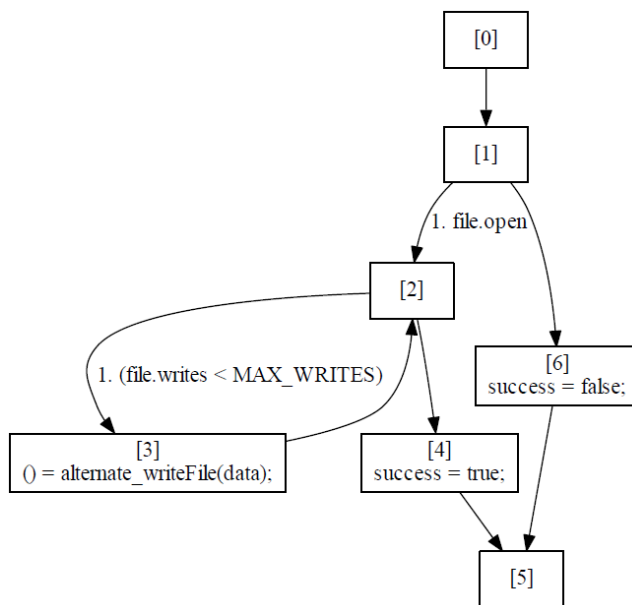


Figure 3 - The Control Flow Graph (CFG) for the program in Example 4

4.2.2.3 Generating Unique Variable Instances

The Lustre language only allows a variable to be assigned once in a given node. This presents problems when translating from Limp to Lustre because Limp does not have such restrictions. This is solved by applying a transformation that gives each variable inside of a basic block have a unique name. Figure 4 shows a simple basic block with variables assigned more than once.

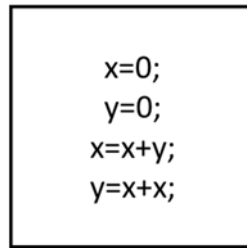


Figure 4 - A simple basic block with variables *x* and *y* assigned twice

In this example, we create new variables for each instance of variables *x* and *y*. This allows us to create a Lustre compliant representation while preserving the semantics of the block. The example Figure 4 becomes the basic block in Figure 5.

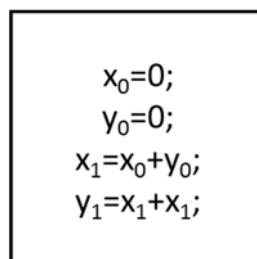


Figure 5 – The example in Figure 4 with unique naming

The instructions inside of this block are now Lustre compliant and the translation of them into Lustre is now trivial.

4.2.3 Translating the CFG representation to Lustre

Once the CFG representation has been created, and all of the basic blocks contained within it have been transformed to contain only unique variables, it is ready to be translated to Lustre. The translation is a multi-step process that must:

- Identify the entry point for the analysis
- Determine the state and global variables in the program
- Translate the referenced type, constant, function, and external procedure declarations
- Generate a node for each basic block in the CFG
- Generate a node that executes the CFG (called the CFG Director node)

4.2.3.1 Identifying the Main Local Procedure

In the SIMPAL framework analysis of a program always happens through the execution of a local procedure. This local procedure defines the input, output, and global variables that a program will read and write. If a program does not contain a local procedure there is no entry point for the analysis and thus an error will be reported by the editing environment to the user. If a program contains multiple local procedures then there are multiple potential entry points for analysis. In the case of multiple local procedures the following rule is used to identify the analysis entry point: if a local procedure is named “main” then it is the entry point for analysis, otherwise the last local procedure in a file is chosen as the entry point.

4.2.3.2 Identifying the State and Global Variables

Once the local procedure that is used as the entry point for analysis is identified, its inputs, locals, and outputs are captured in a global record structure that represents the state of the program for analysis. Similarly, the global variables that are read and written by the program are stored in a second global record structure. These global records are typed as user named types so they can be easily referred to in the translation.

4.2.3.3 Translation of Types, Constants, Functions, and External Procedures from Limp to Lustre

Limp declarations must be translated into Limp. For most of the Limp constructs this is a straightforward process. Translating External Procedures is a more complicated as they represent some additional features that are not native to the target Lustre language. The following sections describe the translation from Limp to Lustre on each type of Limp construct.

4.2.3.3.1 Types

The Limp type system is very similar to the Lustre type system. The only type extensions to the Lustre system are String and Abstract types, both of which are removed with preprocessing transformations. As a result, all the types contained in the model at the time of translation should be trivial. The complete type mapping is shown in Table 2.

Int → Int
Bool → Bool
Real → Real
String ¹ → Int
Abstract ¹ → Int
Array → Array ²
Record → Record ²
¹ String and Abstract types are removed during preprocessing transformations and replaced with Integer types and operations.
² Composite types in Limp are mapped to equivalent Lustre types and subtypes are recursively mapped into Lustre types, terminating when a primitive type is encountered.

Table 2 - Limp to Lustre type mapping

4.2.3.3.2 Constants

The earlier preprocessing pass, Remove Unspecified Constants, ensures that all constants remaining in the model after transformation are specified. These remaining constants are then easily translated into a Lustre constant. For the constant on line 9 of Example 2, the syntax in the Limp and Lustre representations is exactly the same.

4.2.3.3.3 Functions

Local functions in Limp have similar behavior as Lustre nodes, except they do not contain state. They are simply translated into equivalent Lustre nodes with some very minor syntactical differences. The local function **increment** from Example 3 is shown below in Example 11.

1	node increment(in1 : int) returns (out : int);
2	let
3	out = (in1 + 1);
4	tel;

Example 11 - Lustre node representing the increment local function from Example 3

4.2.3.3.4 Procedures

External procedures and local procedures that aren't the entry point of the analysis are translated into uninterpreted functions in Lustre. One aspect that must be captured in this translation is the portions of global variables that each procedure reads and writes, which is defined by the uses and defines specification. For the `alternate_writeFile` procedure on line X of Example 4, the resulting translation can be found in Example 12 below.

1	function alternate_writeFile(
2	data : int;
3	_file_IN : File
4) returns (
5	_file_writes_OUT : int;
6	_file_data_OUT : int
7);

Example 12 – Lustre uninterpreted function representing external function `alternate_writeFile` from Example 4

The external procedure ***alternate_writeFile*** uses, or reads, the global variable ***file***. This additional argument is added to the inputs of the uninterpreted function that represents ***alternate_writeFile*** in Lustre as shown on line 3 of Example 12. It also writes to the *writes* and *data* fields of the global variable ***file***. These fields are added as outputs to the uninterpreted function, shown in lines 5 and 6 of Example 12. As we process statements from the original program, we must update our procedure calls to reflect this new signature. This is discussed in Section 4.2.3.4.3.

4.2.3.4 Generating Lustre nodes from Basic Blocks

Once the state and global record types are defined the basic blocks can be seen as a sequential list of assignments that will update the state and global record types. This is captured as a Lustre node that accepts a record of the current state and global record types. In addition the preconditions and postconditions of any called procedure must be aggregated and passed as output arguments to be used in later analyses. Finally the uses and defines specifications must be handled at the site of the external procedure call to ensure only the specified portions of a global variable are read and written to. The full translation of basic block 3 of the CFG in Figure 3, is shown below.

1	node main_block_3(
2	trigger : bool;
3	state_in : main_state_type;
4	globals_in : main_global_type
5) returns (
6	state_out : main_state_type;
7	globals_out : main_global_type;
8	precondition : bool;
9	postcondition : bool
10);
11	var
12	data_0 : int;
13	success_0 : bool;
14	file_0 : File;

```

15  _file_IN : File;
16  _file_writes_OUT : int;
17  _file_data_OUT : int;
18  file_1 : File;
19  file_2 : File;
20  data : int;
21  success : bool;
22  file : File;
23  alternate_writeFile_pre1_ : bool;
24  alternate_writeFile_pre1__prop : bool;
25  alternate_writeFile_post1 : bool;
26  alternate_writeFile_post2 : bool;
27  let
28    data_0 = state_in.data;
29    success_0 = state_in.success;
30    file_0 = globals_in.file;
31    alternate_writeFile_pre1_ = file_0.open;
32    alternate_writeFile_pre1__prop = (trigger => alternate_writeFile_pre1_);
33    _file_IN = file_0;
34    _file_writes_OUT, _file_data_OUT = alternate_writeFile(data_0, _file_IN);
35    file_1 = file_0{writes := _file_writes_OUT};
36    file_2 = file_1{data := _file_data_OUT};
37    alternate_writeFile_post1 = (file_2.writes = (file_0.writes + 1));
38    alternate_writeFile_post2 = (file_2.data = data_0);
39    data = data_0;
40    success = success_0;
41    file = file_2;
42    state_out = main_state_type {data = data; success = success};
43    globals_out = main_global_type {file = file};
44    precondition = (alternate_writeFile_pre1__prop);
45    postcondition = (alternate_writeFile_post1 and alternate_writeFile_post2);
46    --%PROPERTY alternate_writeFile_pre1__prop;
47  tel;

```

Example 13 - Lustre node representing basic block 3 from Example 4

4.2.3.4.1 Capturing the State and Global Record Variables

The state and global record types are provided as inputs to a basic block and as outputs. The input values represent the state prior to the execution of the basic block, and the outputs are their modified versions after the execution of the basic block.

The inputs are unpacked from the record types as local variables in the Lustre node prior to processing any basic block statements. This is done to simplify namespace issues in the translation. This is accomplished on lines 28-30 in Example 13.

4.2.3.4.2 Processing preconditions and postconditions

Preconditions and postconditions must be processed correctly for the analysis to be accurate. First, the analysis must instantiate the preconditions and postconditions with variable instances from the calling reference. Instantiated preconditions for the call to `alternate_writeFile` are shown on line 31 of Example 13 and its instantiated postconditions are shown on lines 37-38.

All preconditions must be prepended with an implication containing a triggering signal from the node. This signal ensures the preconditions will only be evaluated when the basic block is actively being executed by the CFG Director node, which is described in Section 4.2.3.5. This triggering signal is essential to the analysis of component preconditions because the Lustre semantics execute the signal on

every time step and evaluating the property when the block is not active would lead to spurious counterexamples. This is shown on line 32 of Example 13.

Finally all the preconditions and postconditions are aggregated into a single variable to be passed to the CFG Director node. The postconditions are also aggregated. These aggregated signals are outputs to the node representing the basic block and are used by the CFG Director node for analysis. The aggregated signals are captured on lines 44 and 45 of Example 13.

4.2.3.4.3 Processing the Uses and Defines specifications

When an external procedure is called and it contains uses and defines specifications, several things must occur during the translation to ensure that it is handled correctly. First, the current value of the global variables that are used by an external procedure call must be passed into the uninterpreted function that represents the external procedure to be called. Just prior to the call, the portions of the global variables used by the procedure are extracted to local variables. These local variables are provided as arguments to the uninterpreted function that represents the procedure. In Example 13, this local variable is assigned in line 33 and appears as an input to the uninterpreted function `alternate_writeFile` in line 34.

Similarly the portions of the global variable that are written by the procedure are provided as outputs of the uninterpreted function. These variables must be captured and then merged into the global variable after execution of the uninterpreted function. Variables capturing the updated versions of *writes* and *data* fields of the global variable `file` from the external procedure call are written on line 34 of Example 13 as variables `_file_writes_OUT` and `_file_data_OUT`. These variables are then merged into the previous version of the global variable `file` on lines 35 and 36. This ensures that the `file` variable only changes aspects of the variable that are called out in the defines specification for the external procedure.

4.2.3.5 Generating the CFG Director Node

Once the basic blocks, or nodes of the CFG, are captured as Lustre nodes, the next step is to create a node that will drive the execution of the CFG as a state machine. This node is responsible for initializing the state and global record variables, calling the various nodes (or basic blocks) of the CFG in the correct order to successfully emulate the program, threading the state and global variable records through the execution of the CFG, and setting up system level analyses including contract verification, reachability, and viability analysis. Details of the analyses performed are provided in Section 4.3. Example 14 shows the CFG director node for the program in Example 4.

1	node main(
2	data__in : int;
3	success__in : bool;
4	file__in : File
5) returns (
6	data : int;
7	success : bool;
8	file : File
9);
10	var
11	state : int;

```

12  pre_state : int;
13  init_state : int;
14  final_state : int;
15  state_record : main_state_type;
16  pre_state_record : main_state_type;
17  init_state_record : main_state_type;
18  global_record : main_global_type;
19  pre_global_record : main_global_type;
20  init_global_record : main_global_type;
21  component_preconditions : bool;
22  component_postconditions : bool;
23  main_pre1_ : bool;
24  main_post1 : bool;
25  state_0_is_unreachable : bool;
26  state_0_is_nonviable : bool;
27  state_1_is_unreachable : bool;
28  state_1_is_nonviable : bool;
29  state_2_is_unreachable : bool;
30  state_2_is_nonviable : bool;
31  state_3_is_unreachable : bool;
32  state_3_is_nonviable : bool;
33  state_4_is_unreachable : bool;
34  state_4_is_nonviable : bool;
35  state_5_is_unreachable : bool;
36  state_5_is_nonviable : bool;
37  state_6_is_unreachable : bool;
38  state_6_is_nonviable : bool;
39  let
40    --%MAIN
41    state = (init_state ->
      (if (pre_state = 0) then 1 else
        (if (pre_state = 1) then
          (if file.open then 2 else 6)
        else
          (if (pre_state = 2) then
            (if (file.writes < MAX_WRITES) then 3 else 4)
          else
            (if (pre_state = 3) then 2 else
              (if (pre_state = 4) then 5 else
                (if (pre_state = 5) then pre_state else
                  (if (pre_state = 6) then 5 else
                    pre_state))))))));
42    pre_state = (init_state -> (pre state));
43    init_state = 0;
44    final_state = 5;
45    init_state_record = (main_state_type {data = data__in; success = false} ->
      (pre init_state_record));
46    pre_state_record = (init_state_record -> (pre state_record));
47    init_global_record = (main_global_type {file = file__in} ->
      (pre init_global_record));
48    pre_global_record = (init_global_record -> (pre global_record));
49    state_record,global_record,component_preconditions,component_postconditions =
      ((init_state_record, init_global_record, true, true) ->

```

	<pre> (if (pre_state = 0) then main_block_0((pre_state = 0), pre_state_record, pre_global_record) else (if (pre_state = 1) then main_block_1((pre_state = 1), pre_state_record, pre_global_record) else (if (pre_state = 2) then main_block_2((pre_state = 2), pre_state_record, pre_global_record) else (if (pre_state = 3) then main_block_3((pre_state = 3), pre_state_record, pre_global_record) else (if (pre_state = 4) then main_block_4((pre_state = 4), pre_state_record, pre_global_record) else (if (pre_state = 5) then main_block_5((pre_state = 5), pre_state_record, pre_global_record) else (if (pre_state = 6) then main_block_6((pre_state = 6), pre_state_record, pre_global_record) else (pre_state_record, pre_global_record, true, true)))))))); main_pre1_ = ((state = init_state) => file.open); main_post1_ = ((state = final_state) => success); state_0_is_unreachable = (pre_state <> 0); state_0_is_nonviable = (H(component_preconditions) => (pre_state <> 0)); state_1_is_unreachable = (pre_state <> 1); state_1_is_nonviable = (H(component_preconditions) => (pre_state <> 1)); state_2_is_unreachable = (pre_state <> 2); state_2_is_nonviable = (H(component_preconditions) => (pre_state <> 2)); state_3_is_unreachable = (pre_state <> 3); state_3_is_nonviable = (H(component_preconditions) => (pre_state <> 3)); state_4_is_unreachable = (pre_state <> 4); state_4_is_nonviable = (H(component_preconditions) => (pre_state <> 4)); state_5_is_unreachable = (pre_state <> 5); state_5_is_nonviable = (H(component_preconditions) => (pre_state <> 5)); state_6_is_unreachable = (pre_state <> 6); state_6_is_nonviable = (H(component_preconditions) => (pre_state <> 6)); data = state_record.data; success = state_record.success; file = global_record.file; assert component_postconditions; assert main_pre1_; --%PROPERTY main_post1; --%PROPERTY state_0_is_unreachable; --%PROPERTY state_0_is_nonviable; --%PROPERTY state_1_is_unreachable; --%PROPERTY state_1_is_nonviable; --%PROPERTY state_2_is_unreachable; --%PROPERTY state_2_is_nonviable; --%PROPERTY state_3_is_unreachable; --%PROPERTY state_3_is_nonviable; --%PROPERTY state_4_is_unreachable; --%PROPERTY state_4_is_nonviable; --%PROPERTY state_5_is_unreachable; --%PROPERTY state_5_is_nonviable; --%PROPERTY state_6_is_unreachable; --%PROPERTY state_6_is_nonviable; tel; </pre>
50	
51	
52	
53	
54	
55	
56	
57	
58	
59	
60	
61	
62	
63	
64	
65	
66	
67	
68	
69	
70	
71	
72	
73	
74	
75	
76	
77	
78	
79	
80	
81	
82	
83	
84	
85	
86	

Example 14 - CFG Director node for Example 4

Note to the reader: The H node, used on lines 53,55,57,59,61,63, and 65 of Example 14, is a simple boolean predicate that determines if the provided expression has been true for all of program execution. It stands for Historically and it is defined below in Example 15. It is used for viability analysis to enforce that the the local procedure preconditions and called external procedure postconditions are true, a prerequisite for viability.

1	node H(signal : bool) returns (holds : bool);
2	let
3	holds = signal -> signal and pre (holds);
4	tel;

Example 15 - The Historically node

4.2.3.5.1 Initializing the State and Global Records

The state record contains all of the inputs, locals, and outputs of the local procedure being used as the entry point for analysis. The state record's inputs are populated by assigning them to a nondeterministic input value generated by the JKind model checker. Locals and outputs (computed values) are assigned a default value for their type. These default types are shown in Table 3.

Int → 0
Real → 0.0
Bool → False
Composite Types → Composed of Primitive Defaults

Table 3 - Lustre Types mapped to default values

Line 47 of Example 14 shows the construction of the initial state record for Example 4. Once the state is initialized an expression that tracks the previous values of it is also generated. These are shown on line 46 of Example 14.

Global records are constructed similarly, except all globals are always populated with nondeterministic input variables. In our analysis we want to consider all potential values of a global variable at the beginning of a program, and treating globals in this manner preserves that capability. If the user would like to consider a subset of values for a global variable they can specify those assumptions as preconditions on the procedure itself. Line 49 shows the construction of the initial global record for Example 4. Similar to the state record, the previous value of the global record is also tracked, as shown on line 48 of Example 14.

4.2.3.5.2 Constructing the CFG State Machine Expression

The CFG consists of nodes that are the basic blocks of the program being specified, and the arcs consist of transition from basic block to basic block. Some arcs are guarded and have a condition that determines whether or not they are taken, others are unconditional. In the CFG Director node, each basic block must be executed in the correct sequence to emulate the CFG of the program. For the CFG found in Figure 3 (which is computed from Example 4) we must generate equations that keep track of the initial block to be executed, the block currently being executed, the previously executed block, and the final block being executed.

From the CFG in Figure 3 we can see that the initial block to execute is block 0. This is reflected on line 43 of Example 14. The block being executed is dependent the previous block executed and what

conditions, if any, must hold in the previous state and global records. This state transition relation is defined on line 41 of Example 14. The previously executed basic block is tracked throughout this process and it is derived from the expression found on line 42 of Example 14. Finally the final basic block must also be captured. This is useful for knowing when the program has terminated and is used when checking postconditions of the overall program. Line 44 show the assignment of the final basic block from Figure 3.

4.2.3.5.3 Threading the state and global variables through the CFG execution

Once the state machine that dictates the execution of the CFG is built the next step in the translation is to construct an equation that threads the current state and global variables through the calls to the various basic blocks. Similar to the state machine variables that are discussed in the previous section, we must track the initial, previous, and current state and global records. The generation of the initial value is discussed in Section 4.2.3.5.1. The previous value of the state and global records is trivial to compute. They are shown on lines 46 and 48 of Example 14, respectively.

To compute the current value of the state and global records we must identifying which basic block is currently being executed, call the Lustre node that represents that basic block with the previous values of the state and global records, and capture the output of executing that node in the current state and global records. The component preconditions and postcondition signals are also captured. This is shown on line 49 of Example 14. The reader should note that on the initial state no basic block is called, instead the current values of the state and global vectors are set to the initial values, and the `component_precondition` and `component_postcondition` signals are set to true.

4.2.3.5.4 Generating assertions and proof obligations

The next responsibility of the CFG Director node is to generate the assertions and proof obligations to perform contract reachability, and viability verification. Each type of analysis is discussed in detail in Section 4.3.

External procedure postconditions are lifted to the CFG Director node and asserted as true. This means that when a component is used the analysis only considers outputs that satisfy the external procedures postconditions. This is a key principle of contract based reasoning and it is discussed further in Section 4.3.1. Line 69 of Example 14 shows the assertion over the external procedure postconditions. Further any preconditions found in the entry point local procedure being analyzed are also asserted as true. This is another key principle in contract based reasoning. Line 70 illustrates the single precondition of Example 4 being asserted in the CFG Director node.

Preconditions of all external procedures called from the entry point are turned into proof obligations inside of the node that represents the basic block (see Example 13, Line 46) from which it is called. Additionally all postconditions of the entry point are turned into proof obligations (lines 51 of Example 14) and checked (lines 71 of Example 14) when the CFG is in its final state, which represents termination of the program.

Additional proof obligations representing the reachability and viability analyses are also generated and checked as properties. These are discussed in detail in Sections 4.3.2 and 4.3.3 respectively. The proof

obligations for viability and reachability are generated on lines 52-66 and checked as properties on lines 72-85 of Example 14.

4.2.3.5.5 Unpacking the state and global records

The last thing done in the CFG Director node is to unpack the state and global records into output variables. This isn't necessary, but does make it easy to refer to the original program variables for interpreting any counterexamples the tool may generate.

4.3 Analyses

The SIMPAL tool performs three analyses on programs specified in the Limp language:

1. Contract verification
2. Reachability
3. Viability

If a system is composed of pre-existing components, and only the components' preconditions and postconditions are known, is it possible to formally prove that the new program satisfies its postconditions? Contract verification is an approach to answering this question.

Further, one might want to reason about the reachability of execute certain portions of the specified program. Reachability analysis in SIMPAL either proves the existence of a trace that allows each basic block in the program to be reached or proves that no traces allow a basic block to be reached.

Lastly the concept of viability is an extension of reachability. Viability is reachability under conditions in which all the component preconditions and system postconditions are satisfied. Essentially, viability is the reachability of a portion of the program under intended conditions.

The approach taken by the SIMPAL tool is discussed in the following sections.

4.3.1 Contract Verification

Contract verification, also known as assume-guarantee reasoning, is a compositional reasoning approach that is useful for verifying that a program or system of pre-verified components behaves correctly. A contract for a component, or a system, is the set of assumptions (preconditions) it makes on the environment it operates in and the set of guarantees (postconditions) that it will provide to the environment it operates in. Contract verification in SIMPAL adopts the concept of assume-guarantee reasoning to a program specified in the Limp language.

For the program specified in Example 4, the local procedure **main** has the following contract, comprised of preconditions and postconditions. The precondition **pre1** states that prior to the execution of the program the file.open field is assumed to be set to true. The postcondition **post1** asserts that the success variable will always be true. This is shown in Table 4.

Preconditions: precondition pre1 = file.open;
Postconditions: postcondition post1 = success;

Table 4 - Contract for the main local procedure in Example 4

This program makes a single call to an external procedure, which is to `alternate_writeFile`. That external procedure has one precondition and two postconditions. The precondition `pre1` requires the `file.open` field to be true of the global variable `file` prior to calling the external procedure. The first postcondition specifies that the `writes` field of the global variable `file` will be one greater than the value of it before executing the procedure. The second postcondition specifies that the `data` field of the global variable `file` will be equal to the procedure input `data` after execution. This contract is shown in Table 5.

Preconditions:
precondition <code>pre1 = file.open;</code>
Postconditions:
postcondition <code>post1 = file.writes == (init file.writes) + 1;</code>
postcondition <code>post2 = file.data == data;</code>

Table 5 - Contract for the external procedure `alternate_writeFile`

In the analysis performed by SIMPAL, we want to verify that all of the components are called with variables that satisfy their respective preconditions. This proves that the specified program always honors the preconditions of the external procedures it invokes. Every basic block contains proof obligations that check whether procedure calls satisfy the stated preconditions of the called procedure. This is shown on line 46 of Example 13. The triggering signal ensures the analysis only evaluates the property when the basic block is actively being executed.

Secondly we want to verify that the new program always honors its postconditions. These properties are specified inside of the CFG Director, as discussed in Section 4.2.3.5.4. These properties must only be checked when the program has terminated. A program is terminated when the current basic block being executed is the final basic block of the CFG. Line 53 of Example 14 shows the postcondition of the **main** local procedure. The reader may note that the expression is prepended with an implication (*state = final_state*.) This makes the property trivially true when the state is not the final state which prevents spurious counterexamples.

To correctly perform contract verification assumptions must be made to only consider inputs that satisfy the entry point's preconditions and the components postconditions. Since nothing is known about the internals of the called procedures, each procedure's postconditions can be assumed to be true. It is assumed that the specifier has already verified that each component satisfies its postconditions. This analysis makes no attempt to verify the internals of any called procedure. Line 49 of Example 14 captures all of the component postconditions of the basic block being executed. It is not necessary to reason about individual postconditions, thus they are aggregated into a single signal.

Additionally we must assume that the preconditions of the entry point are true. This occurs on line 70 of Example 14. The reader may note that this precondition is prepended with an implication (*state = init_state*.) This makes the assertion only hold for the initial state of the program, which is prior to any execution of program statements.

4.3.1.1 The File Example

Example 4 demonstrates a simple example in which the local procedure `main`, specifies a program in which a global object, `file`, is written repeatedly until the `MAX_WRITES` value has been reached. In our analysis of this program we must verify that all calls to the external procedure `alternate_writeFile` satisfy its precondition, and that the system always satisfies its single postcondition. The results of this analysis are shown in Figure 6 below.

Property	Result
✓ main_post1	Valid (0s)
! state_0_is_unreachable	Invalid (0s)
! state_0_is_nonviable	Invalid (0s)
! state_1_is_unreachable	Invalid (0s)
! state_1_is_nonviable	Invalid (0s)
! state_2_is_unreachable	Invalid (0s)
! state_2_is_nonviable	Invalid (0s)
! state_3_is_unreachable	Invalid (0s)
! state_3_is_nonviable	Invalid (0s)
! state_4_is_unreachable	Invalid (0s)
! state_4_is_nonviable	Invalid (0s)
! state_5_is_unreachable	Invalid (0s)
! state_5_is_nonviable	Invalid (0s)
✓ state_6_is_unreachable	Valid (0s)
✓ state_6_is_nonviable	Valid (0s)
✓ main_block_3~0.alternate_writeFile_pre1_prop	Valid (0s)

Figure 6 - Contract verification analysis results of Example 4

As we can see the postcondition of local procedure `main`, called `main_post1` is valid. Similarly the precondition of external procedure `alternate_writeFile`, called `alternate_writeFile_pre1_prop` is also valid.

Note to the reader: The remaining properties shown in Figure 6 relate to the reachability and viability of each basic block in the original program. These are discussed in-depth in Sections 4.3.2 and 4.3.3.

4.3.1.2 Modified File Example

For the purposes of demonstrating an invalid property we introduce a modified version of Example 4 in which the local procedure `main`'s precondition is commented out. This allows the global variable `file`'s open field to be both true and false in the analysis. Example 16 shows this change to Example 4 on line 24.

```

1  type record File = {
2      open : bool,
3      writes : int,
4      data : int
5  }
6
7  global file : record File
8
9  constant MAX_WRITES : int = 10
10

```

```

11  external procedure alternate_writeFile(data : int)
12      returns ()
13  attributes {
14      precondition pre1 = file.open;
15      postcondition post1 = file.writes == (init file.writes) + 1;
16      postcondition post2 = file.data == data;
17      uses file;
18      defines file.writes;
19      defines file.data;
20  }
21
22  procedure main(data : int) returns (success : bool)
23  attributes {
24      //precondition pre1 = file.open;
25      postcondition post1 = success;
26  }
27  statements {
28      if(file.open) then {
29          while(file.writes < MAX_WRITES) {
30              alternate_writeFile(data);
31          }
32          success = true;
33      } else {
34          success = false;
35      }
36  }

```

Example 16 - A modified version of Example 4

Performing contract verification on this example yields the results shown in Figure 7.

Property	Result
main_post1	Invalid (1s)
state_0_is_unreachable	Invalid (0s)
state_0_is_nonviable	Invalid (0s)
state_1_is_unreachable	Invalid (0s)
state_1_is_nonviable	Invalid (1s)
state_2_is_unreachable	Invalid (1s)
state_2_is_nonviable	Invalid (1s)
state_3_is_unreachable	Invalid (1s)
state_3_is_nonviable	Invalid (1s)
state_4_is_unreachable	Invalid (1s)
state_4_is_nonviable	Invalid (1s)
state_5_is_unreachable	Invalid (1s)
state_5_is_nonviable	Invalid (1s)
state_6_is_unreachable	Invalid (1s)
state_6_is_nonviable	Invalid (1s)
main_block_3~0.alternate_writeFile_pre1_prop	Valid (0s)

Figure 7 – Contract verification analysis results of Example 16

The precondition of the external procedure **alternate_writeFile**, **pre1**, is still valid. This is because the main program only calls this external procedure when the global variable file's open field is true, thus always satisfying the precondition. However now we see that the local procedure's **main** postcondition, **post1**, is invalid. This is because the If branch (lines 27-30 of Example 16) of the If-Then-Else statement is

never entered. It is inside this branch, on line 30, that the success variable is set to true. Therefore the postcondition **post1** is not satisfied in circumstances when the global variable **file's open** field is false.

Similar to Figure 6, Figure 7 contains properties related to reachability and viability that are explained in Sections 4.3.2 and 4.3.3.

4.3.2 Reachability

Reachability analysis in the SIMPAL framework is a very simple analysis that generates properties that attempt to prove that a given basic block in the CFG is not reachable. For Example 4, the CFG is shown in Figure 3. A basic block is reachable if it can be executed. For each basic block in the CFG, we generate a property that asserts it cannot be reached. Lines 52, 54, 56, 58, 60, 62, and 64 of Example 14 demonstrate the generated reachability properties for Example 4.

These properties attempt to prove that each basic block is unreachable. If the property is proven, then the basic block is unreachable. However, if the property fails, the accompanying counterexample trace will demonstrate at least one sequence of inputs that will lead to the execution of that block.

4.3.2.1 Reachability of the File Example

In Example 4, the local procedure **main's** precondition asserts that the global variable **file's open** field is always true. Careful inspection shows that the Else branch of the If-Then-Else is never executed if the open field is never true. The analysis results shown in Figure 6 show that the basic block 6 (the block that executes the Else branch of the If-Then-Else statement) is never executed, hence the unreachability property proves.

4.3.2.2 Reachability of the Modified File Example

Example 16 represents a modified version of Example 4 which removes the local procedure **main's** only precondition, **pre1**, which asserts that the global **file's open** field is always true. This allows the field to be both true and false. The results of this is now the Else branch of the If-Then-Else is reachable. The analysis of this modified program shown in Figure 7 above reflects this.

4.3.3 Viability

Viability is an extension of reachability. When the user specifies a program composed of multiple external components, one might like to check that every line of the program is reachable. However, there is also an implicit intention by the user to only call a component in ways that satisfy its preconditions. Viability combines the concept of reachability with this implicit intention; that is, it checks that each basic block is reachable under conditions when the local procedure's preconditions and all external postconditions hold. Lines 53, 55, 57, 59, 61, 63, and 65 show the generated viability properties of Example 4.

Similar to reachability, the viability properties attempt to prove that each basic block is nonviable. If the property is proven, then the block is unreachable under conditions when the system calls the basic block with its preconditions satisfied. If the property fails then the accompanying counterexample trace demonstrates at least one sequence of inputs that leads to the execution of that block under conditions

where its preconditions are satisfied. It is worth noting that an unreachable basic block, is by definition, also nonviable. In other words, viability is a subset of reachability.

4.3.3.1 Viability of the Modified File Example

The modified File example, shown in Example 16, makes all the basic blocks from Example 4 reachable. As stated in the previous section, an unreachable block is also nonviable. However, a reachable block can be nonviable, depending on the preconditions required by a given external procedure. The analysis of the modified File example shows that every basic block has at least one trace that allows it to be executed with its preconditions satisfied. This is shown in Figure 7 above.

4.3.3.2 Viability of the Modified Modified File Example

Consider a Limp specification that is a small modification of the already modified File example, shown in Example 2. In this modified example the precondition to external procedure **alternate_writeFile** is slightly changed from the expression *file.open* to *not file.open*. This is demonstrated in Example 17.

```

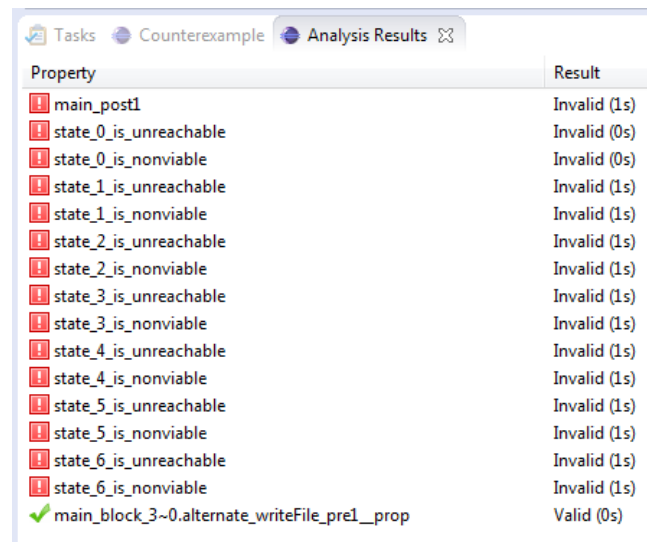
1  type record File = {
2      open : bool,
3      writes : int,
4      data : int
5  }
6
7  global file : record File
8
9  constant MAX_WRITES : int = 10
10
11 external procedure alternate_writeFile(data : int)
12     returns ()
13 attributes {
14     precondition pre1 = not file.open;
15     postcondition post1 = file.writes == (init file.writes) + 1;
16     postcondition post2 = file.data == data;
17     uses file;
18     defines file.writes;
19     defines file.data;
20 }
21
22 procedure main(data : int) returns (success : bool)
23 attributes {
24     //precondition pre1 = file.open;
25     postcondition post1 = success;
26 }
27 statements {
28     if(file.open) then {
29         while(file.writes < MAX_WRITES) {
30             alternate_writeFile(data);
31         }
32         success = true;
33     } else {
34         success = false;
35     }
36 }

```

Example 17 – The Modified Modified File Example

This means that if the If branch of the If-Then-Else statement is executed, the new precondition of **alternate_writeFile** will never be satisfied. It is in direct contradiction to the test condition of the If-

Then-Else statement. Basic block 3 of this program is nonviable, as shown in the analysis results shown in Figure 8.



The screenshot shows a software window titled 'Analysis Results' with a tabbed interface. The 'Analysis Results' tab is active, displaying a table with two columns: 'Property' and 'Result'. The table lists 17 properties, each preceded by a red exclamation mark icon, except for the last one which has a green checkmark icon. The results are either 'Invalid' or 'Valid' with a time value in parentheses.

Property	Result
! main_post1	Invalid (1s)
! state_0_is_unreachable	Invalid (0s)
! state_0_is_nonviable	Invalid (0s)
! state_1_is_unreachable	Invalid (1s)
! state_1_is_nonviable	Invalid (1s)
! state_2_is_unreachable	Invalid (1s)
! state_2_is_nonviable	Invalid (1s)
! state_3_is_unreachable	Invalid (1s)
! state_3_is_nonviable	Invalid (1s)
! state_4_is_unreachable	Invalid (1s)
! state_4_is_nonviable	Invalid (1s)
! state_5_is_unreachable	Invalid (1s)
! state_5_is_nonviable	Invalid (1s)
! state_6_is_unreachable	Invalid (1s)
! state_6_is_nonviable	Invalid (1s)
✓ main_block_3~0.alternate_writeFile_pre1__prop	Valid (0s)

Figure 8 - Analysis results of the Modified Modified File example

5 Limitations

This section discusses some inherent limitations in the SIMPAL reasoning framework.

5.1 String semantics

The underlying analysis language, Lustre, does not support string types. As described in Section 4.2.1.4, strings are removed prior to the translation from Limp to Lustre and replaced with integers. In this translation the ability to equate two strings is preserved, but we are unable to reason about the concatenation of strings or analyze substrings of a string type. Currently the targeted analysis tool, JKind, does not support string types. However ongoing research to develop SMT theories may improve reasoning over string types. Once this capability is sufficiently mature and integrated into the JKind tool the Limp translation can be updated to support this capability.

5.2 Performance

The target analysis tool, JKind, is a bounded model checker extended to prove properties over reactive systems using a technique called k-induction. Bounded model checking is not able to prove properties, only show the absence of property violations for a finite depth. K-induction extends this by performing a corresponding inductive proof that establishes the absence of property violations over an arbitrary depth of the model. Coupling the results of bounded model checking (to establish validity of the base case) with proof of correctness of the arbitrary depth allows the tool to inductively prove the property of interest.

K-induction is sensitive to increasing values of k. This is mitigated by introducing lemmas that allow properties to be proven using smaller values of k. The translation of Limp to Lustre maps an entire Limp program into a state machine that executes the program over multiple Lustre steps. The number of basic blocks in a Limp program dictates the total number of steps to reach termination of the function call. This means that using this approach on large programs could result in large k-values necessary to prove properties by induction. As the k-value needed to prove a property increase, the tool is more likely to return indeterminate results, which means it is unable to prove or falsify a given property.

6 Implementation

The SIMPAL tool is implemented in the Xtext framework. Xtext is a framework for developing domain-specific languages. By providing a language description in the Xtext grammar specification language, users can quickly and easily build Eclipse based development environments to parse, analyze, and process their domain-specific languages. The SIMPAL framework provides an environment for parsing, validating and analyzing Limp specifications.

The Limp integrated development environment (IDE) is shown in Figure 9.

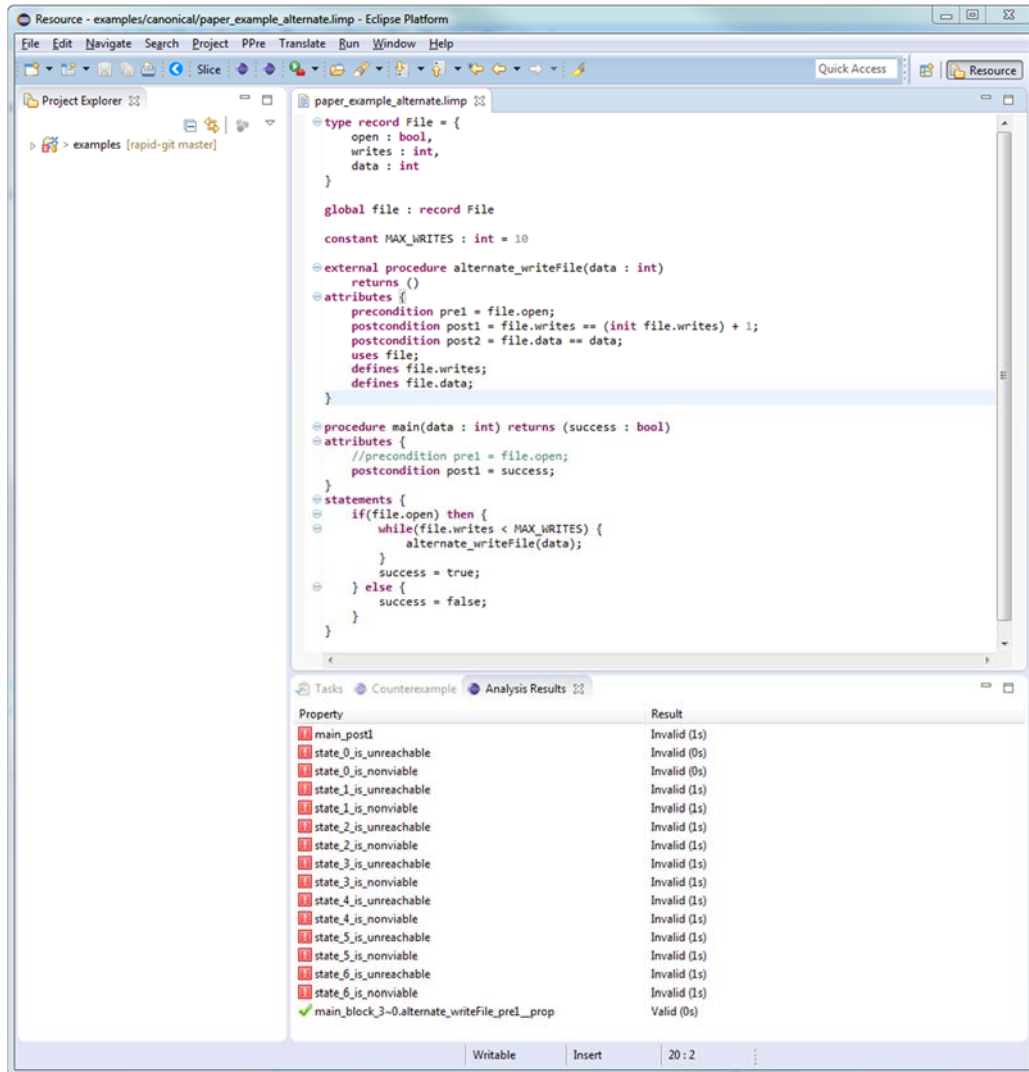
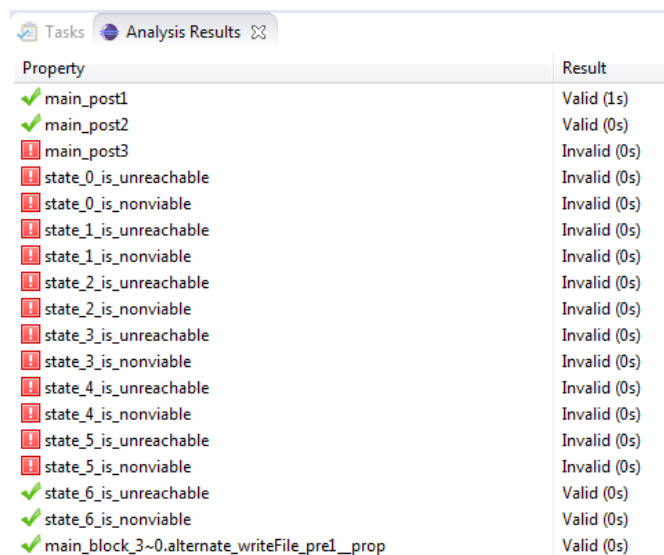


Figure 9 - The Limp IDE

The integrated development environment provides capability to analyze a program and immediately give the user feedback as they edit the file. For example, type checking is done on the fly. However, higher level analyses can also be performed. For example, the tool will highlight an external procedure that contains no outputs and modifies no global variables, notifying the user that the procedure basically performs a null-operation.

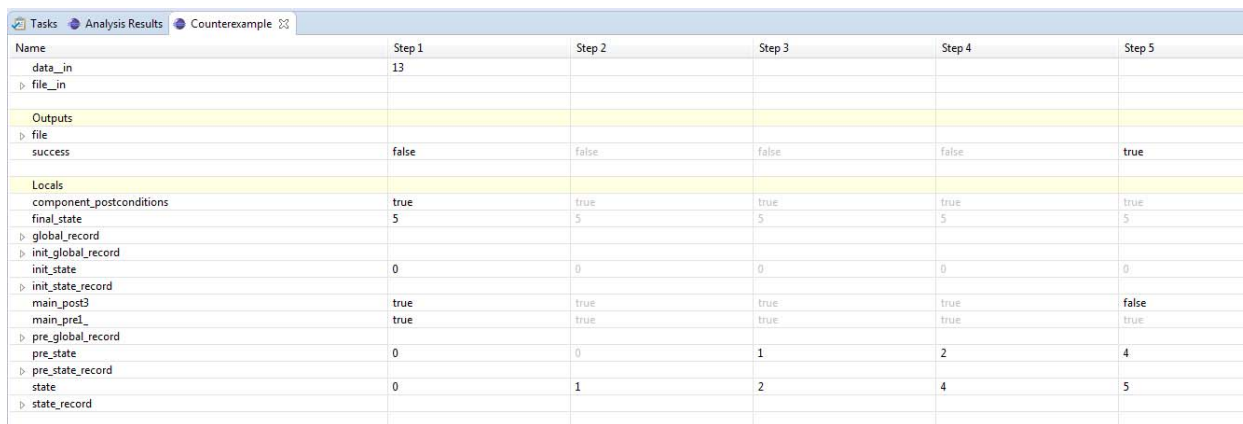
Further, when a program is analyzed the results are displayed conveniently for the user in a tab labelled Analysis Results. This tab is shown in Figure 10.



Property	Result
main_post1	Valid (1s)
main_post2	Valid (0s)
main_post3	Invalid (0s)
state_0_is_unreachable	Invalid (0s)
state_0_is_nonviable	Invalid (0s)
state_1_is_unreachable	Invalid (0s)
state_1_is_nonviable	Invalid (0s)
state_2_is_unreachable	Invalid (0s)
state_2_is_nonviable	Invalid (0s)
state_3_is_unreachable	Invalid (0s)
state_3_is_nonviable	Invalid (0s)
state_4_is_unreachable	Invalid (0s)
state_4_is_nonviable	Invalid (0s)
state_5_is_unreachable	Invalid (0s)
state_5_is_nonviable	Invalid (0s)
state_6_is_unreachable	Valid (0s)
state_6_is_nonviable	Valid (0s)
main_block_3~0.alternate_writeFile_pre1__prop	Valid (0s)

Figure 10 - Analysis Results Tab

Further, if a property has failed, such as main_post3, shown in Figure 10, the user can easily right click on it and display the counterexample trace inside of the IDE. This capability is shown in Figure 11. It is very useful for understanding why a property has failed and provides assignments to each variable in the program that the user can conveniently view, even collapsing large record and array objects for the user's convenience.



Name	Step 1	Step 2	Step 3	Step 4	Step 5
data_in	13				
file_in					
Outputs					
file					
success	false	false	false	false	true
Locals					
component_postconditions	true	true	true	true	true
final_state	5	5	5	5	5
global_record					
init_global_record					
init_state	0	0	0	0	0
init_state_record					
main_post3	true	true	true	true	false
main_pre1	true	true	true	true	true
pre_global_record					
pre_state	0	0	1	2	4
pre_state_record					
state	0	1	2	4	5
state_record					

Figure 11 - Viewing the counterexample trace in the Eclipse IDE

7 References

- [1] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, pp. 1305-1320, 1991.
- [2] M. Dierkes, "Formal Analysis of a Triplex Sensor Voter in an Industrial Context," in *FMICS'11 Proceedings of the 16th International Conference on Formal Methods for Industrial Critical Systems*, 2011.
- [3] G. Hagen and C. Tinelli, "Scaling up the formal verification of Lustre programs with SMT-based techniques," in *Proceedings of the 8th Internal Conference on Formal Methods in Computer-Aided Design*, Portland, 2008.
- [4] T. Kahsai and C. Tinelli, "PKind: A parallel k-induction based model checker," in *Parallel and Distributed Methods in Verification*, 2011.
- [5] E. Clarke, A. Biere, R. Raimi and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Formal Methods in System Design*, vol. 19, pp. 7-34, 2001.
- [6] L. de Moura, H. Rueß and M. Sorea, "Bounded Model Checking and Induction: From Refutation to Verification," in *Computer Aided Verification (CAV) 2003*, Springer Berlin Heidelberg, 2003, pp. 14-26.
- [7] A. Bradley, "SAT-based model checking without unrolling," in *Proceedings of 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011.
- [8] A. Gacek, "JKind Github homepage," Rockwell Collins, [Online]. Available: <https://github.com/agacek/jkind>. [Accessed 29 May 2015].
- [9] S. Quinton, S. Graf and R. Passerone, "Contract-Based Reasoning for Component Systems with Complex Interactions," Verimag, 2010.

Appendix A - Limp DSL Grammar

```
grammar com.rockwellcollins.atc.Limp hidden(WS, SL_COMMENT, ML_COMMENT)

generate limp "http://www.rockwellcollins.com/atc/Limp"

import "http://www.eclipse.org/emf/2002/Ecore" as.ecore

Specification:
    (declarations+=Declaration)*
;

Declaration:
    Import
    | Comment
    | ExternalFunction
    | ExternalProcedure
    | LocalFunction
    | LocalProcedure
    | ConstantDeclaration
    | GlobalDeclaration
    | TypeDeclaration;

Comment:
    comment=SEMANTIC_COMMENT
;

terminal SEMANTIC_COMMENT : '/'# -> '#/';

Import:
    'import' (importURI=STRING)
;

ExternalFunction:
    'external' 'function' (name = ID) '(' inputs=InputArgList ')' 'returns' '('
output=OutputArg ')'
;

ExternalProcedure:
    'external' 'procedure' (name = ID) '(' inputs=InputArgList ')' 'returns' '('
outputs=OutputArgList ')'
    (attributeBlock=AttributeBlock)
;

LocalFunction:
    'function' (name = ID) '(' inputs=InputArgList ')' 'returns' '(' output=OutputArg ')'
    (varBlock=VarBlock)
    'equations' (equationBlock=EquationBlock)
;

LocalProcedure:
    'procedure' (name = ID) '(' inputs=InputArgList ')' 'returns' '('
outputs=OutputArgList ')'
    (varBlock=VarBlock)
    (attributeBlock=AttributeBlock)
    'statements' (statementblock=StatementBlock)
;

TypeDeclaration:
    {TypeAlias} 'type' (name=ID) '=' (type=Type)
```

```

|      EnumTypeDef
|      RecordTypeDef
|      ArrayTypeDef
|      AbstractTypeDef
;

VarBlock:
    {SomeVarBlock} 'var' '{' (locals+=LocalArg)* '}'
|
    {NoVarBlock}
;

EnumTypeDef:
    'type' 'enum' (name=ID) '=' '{' (enumerations+=EnumValue) (','
enumerations+=EnumValue)* '}'
;

EnumValue:
    {EnumValue} (name=ID)
;

RecordTypeDef:
    'type' 'record' (name=ID) '=' '{' (fields+=RecordFieldType) (','
fields+=RecordFieldType)* '}'
;

ArrayTypeDef:
    'type' 'array' (name=ID) '=' (baseType=Type) '[' size=(INT) ']'
;

AbstractTypeDef:
    'type' 'abstract' (name=ID)
;

RecordFieldType:
    (fieldName=ID) ':' (fieldType=Type)
;

ConstantDeclaration:
    'constant' (name=ID) ':' (type=Type) ('=' (expr=Expr))?
;

GlobalDeclaration:
    'global' (name=ID) ':' (type=Type)
;

VariableRef:
    InputArg
|
    LocalArg
|
    OutputArg
|
    ConstantDeclaration
|
    GlobalDeclaration
|
    EnumValue
;

InputArgList:
    {InputArgList} ((inputArgs+=InputArg) (',' inputArgs+=InputArg)*)?
;

InputArg:
    (name=ID) ':' (type=Type)
;

```

```

LocalArg:
    (name=ID) ':' (type=Type) ';'
;

OutputArgList:
    {OutputArgList} ((outputArgs+=OutputArg) (',' outputArgs+=OutputArg)*)?
;

OutputArg:
    (name=ID) ':' (type=Type)
;

Type:
    {VoidType} 'void'
    |
    {BoolType} 'bool'
    |
    {IntegerType} 'int'
    |
    {RealType} 'real'
    |
    {StringType} 'string'
    |
    {EnumType} 'enum' (enumDef=[EnumTypeDef])
    |
    {RecordType} 'record' (recordDef=[RecordTypeDef])
    |
    {ArrayType} 'array' (arrayDef=[ArrayTypeDef])
    |
    {AbstractType} 'abstract' (abstractDef=[AbstractTypeDef])
    |
    {NamedType} (alias=[TypeAlias])
;

AttributeBlock:
    {SomeAttributeBlock} 'attributes' '{' (attributeList+=Attribute)* '}'
    |
    {NoAttributeBlock}
;

Attribute:
    Precondition
    |
    Postcondition
    |
    Define
    |
    Uses
;

Precondition:
    'precondition' (name=ID) '=' (expr=Expr) ';'
;

Postcondition:
    'postcondition' (name=ID) '=' (expr=Expr) ';'
;

DefineUseRef:
    (referenceExpr=Expr)
;

Define:
    'defines' (elements+=DefineUseRef) (',' elements+=DefineUseRef)* ';'
;

Uses:
    'uses' (elements+=DefineUseRef) (',' elements+=DefineUseRef)* ';'
;

StatementBlock:
    {StatementBlock} '{' (statements+=Statement)* '}'
;

```



```

Statement:
    VoidStatement
|   AssignmentStatement
|   IfThenElseStatement
|   WhileStatement
|   ForStatement
|   GotoStatement
|   LabelStatement
|   {BreakStatement} 'break' ';'
|   {ContinueStatement} 'continue' ';'
|   {ReturnStatement} 'return' ';'
;

VoidStatement:
    (expr=Expr) ';'
;

AssignmentStatement:
    (ids=IdList) '=' (rhs=Expr) ';'
;

IfThenElseStatement:
    'if' (cond=Expr)
    'then' (thenBlock=StatementBlock)
    (else=Else)
;

Else:
    {ElseBlock} 'else' (block=StatementBlock)
|   {ElseIf} 'else' (ifThenElse=IfThenElseStatement)
|   {NoElse}
;

WhileStatement:
    'while' (cond=Expr) (block=StatementBlock)
;

ForStatement:
    'for' '(' (initStatement=AssignmentStatement) (limitExpr=Expr) ';'
    (incrementStatement=AssignmentStatement) ')' (block = StatementBlock)
;

LabelStatement:
    'label' (name=ID) ';'
;

GotoStatement:
    {GotoStatement} 'goto' (label=[LabelStatement]) ('when' (whenExpr=Expr))? ';'
;

EquationBlock:
    {EquationBlock} '{' (equations+=Equation)* '}'
;

Equation:
    VoidStatement
|   AssignmentStatement
;

IdList:

```

```

        (ids+=[VariableRef]) (',' (ids+=[VariableRef]))*
;

Expr:
    IfThenElseExpr
;

IfThenElseExpr returns Expr:
    ChoiceExpr (=>({IfThenElseExpr.condExpr=current} '?' (thenExpr=Expr) ':'
    (elseExpr=Expr)))?
;

ChoiceExpr returns Expr:
    {ChoiceExpr} 'choice' '(' (first=Expr) ',' (second=Expr) ')' //analysis cannot be run
    if a spec contains this
    |
    ImpliesExpr
;

ImpliesExpr returns Expr:
    OrExpr (=>({BinaryExpr.left=current} op='=>') right=ImpliesExpr)?
;

OrExpr returns Expr:
    AndExpr (=>({BinaryExpr.left=current} op='or') right=AndExpr)*
;

AndExpr returns Expr:
    RelationalExpr (=>({BinaryExpr.left=current} op='and') right=RelationalExpr)*
;

RelationalOp:
    '<' | '<=' | '>' | '>=' | '==' | '<>'
;

RelationalExpr returns Expr:
    PlusExpr (=>({BinaryExpr.left=current} op=RelationalOp) right=PlusExpr)?
;

PlusExpr returns Expr:
    MultExpr (=>({BinaryExpr.left=current} op=('+' | '-')) right=MultExpr)*
;

MultExpr returns Expr:
    UnaryExpr (=>({BinaryExpr.left=current} op=('*' | '/')) right=UnaryExpr)*
;

UnaryExpr returns Expr:
    AccessExpr
    |
    {UnaryNegationExpr} 'not' (expr=UnaryExpr)
    |
    {UnaryMinusExpr} '-' (expr=UnaryExpr)
;

AccessExpr returns Expr:
    TerminalExpr (=>({RecordAccessExpr.record=current} '.' field=ID
    |=>({RecordUpdateExpr.record=current} '{' field=ID ':=' value=Expr
    '}'
    |=>({ArrayAccessExpr.array=current} '[' index=Expr
    (=>({ArrayUpdateExpr.access=current} ':=' value=Expr)? ']')
    )*)
;

```

```

FunctionRef:
    ExternalFunction
    ExternalProcedure
    LocalFunction
    LocalProcedure
;

TerminalExpr returns Expr:
    '(' Expr ')'
    {BooleanLiteralExpr} (boolVal=BOOLEAN)
    {IntegerLiteralExpr} (intVal=INT)
    {IntegerWildCardExpr} '*' //analysis cannot be run if this expression is present
    {RealLiteralExpr} (realVal=REAL)
    {StringLiteralExpr} (stringVal=STRING)
    {InitExpr} 'init' (id=[VariableRef])
    {SecondInit} 'second_init' (id=[VariableRef]) //analysis cannot be run if this
expression is present
    ArrayExpr
    RecordExpr
    {IdExpr} (id=[VariableRef])
    {FcnCallExpr} (id=[FunctionRef]) '(' exprs=ExprList ')'
;

ArrayExpr:
    'array' (arrayDefinition=[ArrayTypeDef]) '[' (exprList+=Expr) (',' exprList+=Expr)*
    ']'
;

RecordExpr:
    'record' (recordDefinition=[RecordTypeDef]) '{' (fieldExprList+=RecordFieldExpr) (','
fieldExprList+=RecordFieldExpr)* '}'
;

RecordFieldExpr:
    (fieldName=ID) '=' (fieldExpr=Expr)
;

ExprList: {ExprList}
    ((exprList+=Expr) (',' exprList+=Expr)*)?
;

terminal INT returns ecore::EBigInteger: ('0'..'9')+;
terminal BOOLEAN: 'true' | 'false';
terminal REAL: INT '.' INT;

terminal ID : '^?('a'..'z'|'A'..'Z'|'_' ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*);
terminal STRING :
    ''' ( '\\ ' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'|'|'|'\\' */ | !('\\'|''))
)* ''' |
    """ ( '\\ ' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'|'|'|'\\' */ | !('\\'|''))
)* """
;
terminal ML_COMMENT : '/*' -> '*/';
terminal SL_COMMENT : '//' !('\n'|\r)* ('\r'? '\n')?;
terminal WS : (' |\t|\r|\n')+;

terminal ANY_OTHER: .;

/* This section captures all of the AST objects that we include in the language, but do not
* allow the user to write. They are used to make transformations to the code and simplify
* our tasks.

```

```
*/  
ExtendedType returns Type:  
    {TupleType} '(' (typeList+=Type) (',' typeList+=Type)* ')'  
;  
ExtendedExpr returns Expr:  
    {FreshVariable} '@'(value=INT) '@'  
;
```