SIMPAL: A Compositional Reasoning Framework for Imperative Programs

Lucas Wagner
Iowa State University
Ames, IA 50013, USA
lgwagner@iastate.edu

David Greve
Rockwell Collins
Cedar Rapids, IA 52498, USA
david.greve@rockwellcollins.com

Andrew Gacek
Rockwell Collins
Bloomington, MN 55438, USA
andrew.gacek@rockwellcollins.com

ABSTRACT

The Static IMPerative AnaLyzer (SIMPAL) is a tool [1] for performing compositional reasoning over software programs that utilize preexisting software components. SIMPAL features a specification language, called Limp, for modeling programs that utilize preexisting components. Limp is an extension of the Lustre synchronous data flow language. Limp extends Lustre by introducing control flow elements, global variables, and syntax specifying preconditions, postconditions, and global variable interactions of preexisting components.

SIMPAL translates Limp programs to an equivalent Lustre representation which can be passed to the JKind model checking tool to perform assume-guarantee reasoning, reachability, and viability analyses. The feedback from these analyses can be used to refine the program to ensure the software functions as intended.

KEYWORDS

assume-guarantee reasoning, model checking, lustre

ACM Reference format:
Lucas Wagner, David Greve, and Andrew Gacek. 2017. SIMPAL: A Compositional Reasoning Framework for Imperative Programs. In *Proceedings of SPIN, Santa Barbara, CA, USA, July 2017 (SPIN'17),* 4 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software reuse is the practice of using existing software to build new software. Reasons for resuing software are convenience, economy, and recognized service history of preexisting code. However, software is specifically engineered and tested to work for a specific purpose. If software is reused in a different context the overall assurance case for it is incomplete; the original claim for trusting the software is based on a specific usage.

This paper introduces the Static IMPerative AnaLysis (SIMPAL) tool. SIMPAL provides capabilities to model and analyze programs composed of preexisting software components. It allows users to specify precisely how components utilize and modify global variables, what preconditions a component is expecting to be invoked with, and what postconditions the component will provide back to the calling context. SIMPAL uses assume-guarantee analysis to ensure that each component correctly adheres to its specification. Additional analyses are also performed to identify unreachable or nonviable code in the program model. Analyses results are reported back to the user and property violations are accompanied by a sequence of inputs that demonstrate how to violate the component contracts specified in the new program. This information can be used to refine the program specification.

SIMPAL models interactions among program components in the context of a new program. To successfully use it one must generate specifications for preexisting components to be reused in a new program. Component specifications may be derived manually or automatically with tools. Once the components are modeled, the behavior of a new program using those components can be analyzed using SIMPAL. The proposed work flow for SIMPAL is shown in Figure 1.

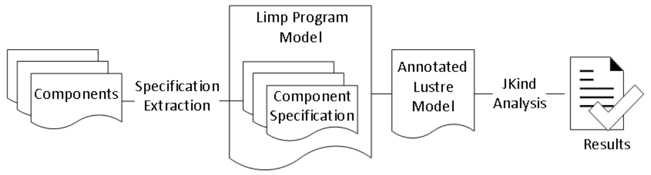


Figure 1: SIMPAL architecture flow.

2 BACKGROUND

Assume-guarantee reasoning [10] is a method of compositional verification that seeks to analyze a system of components without considering their internal design. Instead, each component is described by a contract that describes the assumptions the component expects the environment to adhere to when using it and a set of guarantees the component provides back. The AGREE [6] framework performs assume-guarantee reasoning over models written in the AADL language. AGREE can reason about systems and software architectures expressed in the Architecture Analysis and Design Language[8] (AADL). Behavioral aspects of an architectural component are captured in the AGREE annex to AADL. AGREE merges architectural and behavioral information into a Lustre[11] model which can be analyzed using the Kind 2[2] or JKind[9] model checking tools. While AGREE can be used to reason about software architecture modeled as threads and processes, it cannot reason about software behaviors.

[1]The views expressed are those of the authors and do not reflect the official policy or position of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

Similar to AGREE, the Othello Contracts Refinement Analysis (OCRA) [3] tool performs compositional reasoning over systems described in the OCRA System Specification language (OSS), a textual format unique to the OCRA tools. OCRA allows users to model hybrid and discrete time systems. OCRA targets the NuXMV model checker [1] for discrete time systems and the HyCOMP [4] analysis tool for hybrid systems. Similar to the AGREE tool, OCRA performs assume-guarantee reasoning over discrete time but also extends the concept to hybrid systems.

Frama-C[13] a C source code analysis framework, utilizes assume-guarantee reasoning as part of its abstract interpretation [7] based Value analysis plug-in. The tool provides an option to use contracts to substitute behavior of called functions. Similarly, the SPARK 2014[14] tool set provides a utility, gnatprove[12] based on abstract interpretation, and utilizes assume-guarantee reasoning for subprograms.

3 LIMP

The Limp language is a domain specific language (DSL) for modeling programs constructed from preexisting components. Limp is an extension of Lustre. It has a similar syntax and type system, but incorporates concepts from imperative programming languages like C or Java to make modeling programs easier than it would be in Lustre. The following subsections discuss the key features of the Limp language.

3.1 Key Features of the Limp language

The Limp type system is similar to the Lustre type system in that it supports primitive types of integer, boolean, real, enumerations, and composite types record and array. Limp extends the Lustre type system by adding abstract types and strings. Abstract types can be used to model elements abstractly when the details of a type are not needed. String types are useful for reasoning about programs, but support for them in SIMPAL's analyses is limited, due to the JKind's lack of support for strings.

Limp provides a full complement of control flow constructs, including if-then-else statements, for and while loops, break and continue statements for early loop termination, and goto and label statements for arbitrary control flow. These constructs provide an enhanced specification of software programs than that provided by the Lustre language. Additionally, global variables are present in Limp to provide a richer language for describing programs.

Finally, Limp provides a specification language for a computational modules, referred to procedures. Procedures capture the signature of the computational module and contractual information of the module. A module's contract is composed of its preconditions (relationships it expects the environment to satisfy when it invokes the module), postconditions (relationships the module will provide back to the environment), and precise specifications for how the module uses (reads) and defines (writes) global variables. These elements provide complete information about how the program uses each component to adequately reason about it's behaviors.

3.2 A file writing example

Shown in Figure 2 is a simple Limp specification that describes a program that uses an existing component to write a file object. This program defines a data object called *File* (lines 1-5) as an abstract data object that contains fields *open*, *writes*, and *data* which refer to different characteristics of a file. The instantiation of the *File* data object is the global variable, *file* (line 7). Next, we see a constant *MAX_WRITES* (line 9) that defines the max number of times a *File* object can be written.

```
 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(data : int)
12        returns ()
13  attributes {
14        uses file.open;
15        defines file.data;
16        precondition pre = file.open;
17        postcondition post =
18              file == file{data := data};
19  }
20
21  procedure main(data : int) returns (success : bool)
22  attributes {
23        precondition pre = file.open and file.writes == 0;
24        postcondition post = success;
25  }
26  statements {
27        if(file.open) then {
28              while(file.writes < MAX_WRITES) {
29                    writeFile(data);
30                    file= file {writes := file.writes + 1};
31              }
32              success = true;
33        } else {
34              success = false;
35        }
36  }
```

Figure 2: Specification for a file writing program.

Next, an external procedure *writeFile* (lines 11-19) accepts a integer input named *data* and modifies the global variable *file* by updating its *data* field to the value contained *data* input variable. The procedure is allowed to read (line 14) the *open* field of the global variable *file* and write (line 15) the *data* field. The *writeFile* procedure has one precondition *pre* (line 16) that requires that the file to be written is open. It has a single postcondition *post* (line 17-18) that ensures that the global *file* variable's *data* field has been assigned to the value of the procedure's *data* input.

Finally, the *main* (lines 21-36) procedure has one input argument and one output argument. The input *data* is the data to be written to the global *file*. The output *success* is a boolean that represents whether or not the file object was written successfully. The procedure has a single postcondition named *post* (line 24) and it states the procedure will set the *success* variable equal to true in its final state. Also, the main procedure has a precondition *pre* (line 25) that expects the *open* field to be true, and the *writes* field to be 0, prior to executing the procedure. Together *pre* and *post* form the contract for this procedure. This contract states that if the global variable *file* is open prior to executing the *main* the procedure will always end with the *successful* flag being set to true.

4 ANALYSES

SIMPAL provides three analyses which are useful for evaluating whether or not a program specification meets its intended function. The first analysis is assume-guarantee reasoning. Its purpose is to ensure the specified program will satisfy its postconditions and always call preexisting components such that their preconditions are satisfied. Reachability analysis identifies dead code. Viability analysis identifies code that is unreachable under circumstances when all component preconditions and postconditions are satisfied.

All three analyses are supported by transforming a Limp specification into a Control Flow Graph (CFG) and then translating the CFG into an equivalent Lustre state machine. The CFG for Figure 2 is shown in Figure 3.

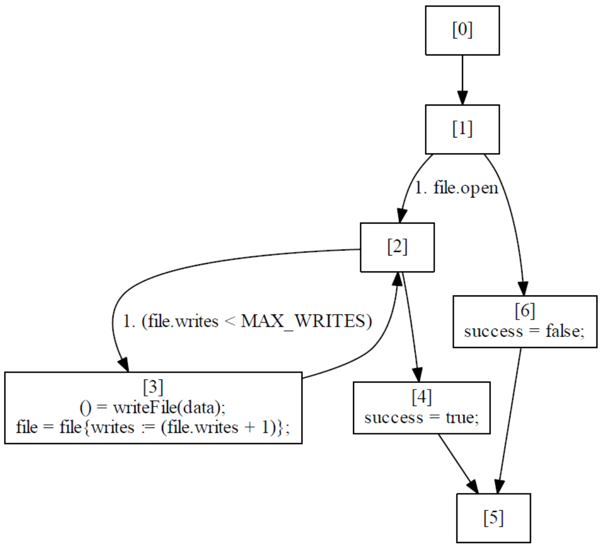


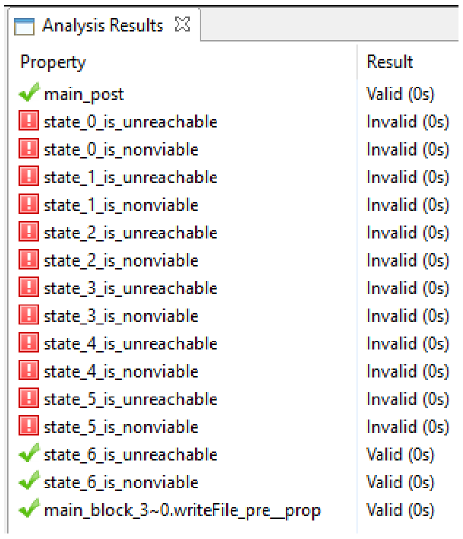
Figure 3: Control flow graph for Figure 2

The input, output, local, and global variables of the program are provided as inputs to each node of the CFG. The CFG node performs its computation and returns modified outputs, locals, and globals. SIMPAL annotates the resultant Lustre model of the program with properties that embody the three analyses. The annotated Lustre model is analyzed by JKind and each property evaluated by it. The following subsections discuss the three analyses and how properties are emitted in the Lustre model to perform each.

4.1 Assume-Guarantee Reasoning

SIMPAL produces a Lustre state machine that represents the Limp program. Assume-guarantee reasoning is performed by adding proof obligations to the Lustre model that determine if the specified program's final postconditions are satisfied and whether each called components preconditions are satisfied. It also adds assumptions that ensure the analysis only considers traces in which the program's preconditions hold as well as assumptions that ensure each called component's postconditions hold. Finally, it is necessary to assume that each component will only modify the portions of a global variable as defined by its defines specification.

In the example shown in Figure 2 this means we must ensure the *main* procedure's postcondition *post* and the called procedure *writeFile*'s precondition *pre* are emitted as properties in the generated Lustre representation. In addition, the tool must ensure that the *main* procedure's precondition *pre* is asserted as well as the called procedure *writeFile*'s postcondition *post.* This means the global *file* must be open at the start of the execution of the *main* procedure and that the after execution of the *writeFile* procedure, the global *file*'s *writes* field will be incremented, and the *data* field set to the input value of *writeFile.* Further, the analysis must also ensure that the *writeFile* function only modifies the *writes* and *data* fields of the global variable *file.* SIMPAL's analysis results for the example shown in Figure 2 are shown in Figure 4.



Property	Result
✓ main_post	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.writeFile_pre__prop	Valid (0s)

Figure 4: Anaysis results for the example in Figure 2

Here can see the that the *main* procedure's postcondition *post* is valid as shown by the property *main_post* analysis result in Figure 4. Also, we can see that the call to the external procedure *writeFile* preserves its precondition *pre* and it is captured in the main_block_3~0.writeFile_pre__prop property in the results. This analysis informs us that our program always ends with the *successful* flag being set to true, given the *main* procedure's preconditions and *writeFile* component's postconditions.

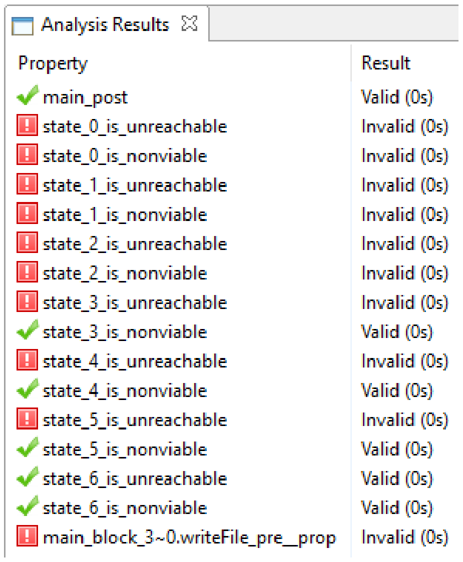
4.2 Reachability

Reachability analysis is useful for identifying code that cannot be exercised due to the way the program is constructed. This analysis is emitting properties that assert each node of the CFG can never be reached. If the property is valid (green check mark) we have proven the node is unreachable. If the property is invalid (red exclamation point) the tool will provide us with a single trace that shows how the node can be reached.

The analysis shown in Figure 4 shows us that nodes 0,1,2,3,4,5 from the CFG shown in Figure 3 are reachable, while node 6 is not. Careful analysis of the program confirms that if the *main* procedure's precondition is that the global variable *file*'s *open* field is always true, then the else branch of the if-then-else (the instruction on line 34) will never be executed.

4.3 Viability

A viable CFG node is one that is reachable while satisfying the contracts of the components contained within it. A nonviable node cannot ever be reached under the same conditions. To demonstrate this, consider a modification to the precondition *pre* (line 16) of the *writeFile* procedure of the example in Figure 2 to require that the *open* field of the global variable *file* now be false. This creates a situation where some CFG nodes are reachable, but not viable. The analysis results from this modification are shown in Figure 5.



Property	Result
✓ main_post	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	Valid (0s)
! state_4_is_unreachable	Invalid (0s)
✓ state_4_is_nonviable	Valid (0s)
! state_5_is_unreachable	Invalid (0s)
✓ state_5_is_nonviable	Valid (0s)
✓ state_6_is_unreachable	Valid (0s)
✓ state_6_is_nonviable	Valid (0s)
! main_block_3~0.writeFile_pre__prop	Invalid (0s)

Figure 5: Analysis demonstrating nonviable CFG nodes

Here we observe that this modification caused nodes 3, 4, 5 of the CFG to become nonviable. This is because the entrance into to the if-then-else statement (line 27) requires the global variable *file*'s *open* field to be true, yet the precondition to the *writeFile* procedure called on line 29 requires it to be false. This contradiction makes the nodes of the CFG contained within the if-then-else statement to be nonviable. Node 6 remains unreachable, and by definition, also nonviable.

5 CONCLUSIONS AND FUTURE WORK

This paper presents SIMPAL, a tool for modeling and performing assume-guarantee reasoning of programs built from preexisting components. It provides a domain specific language for modeling programs and performs analyses by translating the models to a Lustre representation which is analyzed using the JKind model checker.

Future work on SIMPAL will be focused on improving the performance of the underlying Lustre representation. First, the generated Lustre models map program execution over multiple Lustre execution steps. This can require large k-values for the underlying k-induction model checker to prove properties of interest. These large k-values generally drive up the time it takes to analyze properties. One solution to this would be to pursue loop-unwinding, such as that employed by the CBMC [5] model checker. Another approach to improve performance is the generation of lemmas that can aid the k-induction engine. JKind already performs invariant generation, but generating and emitting lemmas related to the structure of the Lustre model emitted by SIMPAL may provide additional information to the solver to reduce analysis times.

SIMPAL is an open source tool and is available for download at http://www.github.com/lgwagner/simpal.

6 ACKNOWLEDGMENT

This work was sponsored in part by the United States Department of Defense.

REFERENCES

[1] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 334–342. DOI: http://dx.doi.org/10.1007/978-3-319-08867-9_22
[2] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. 2016. *The Kind 2 Model Checker.* Springer International Publishing, Cham, 510–517. DOI: http://dx.doi.org/10.1007/978-3-319-41540-6_29
[3] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. 2013. OCRA: A tool for checking the refinement of temporal contracts. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 702–705. DOI: http://dx.doi.org/10.1109/ASE.2013.6693137
[4] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. 2015. HyComp: An SMT-Based Model Checker for Hybrid Systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Christel Baier and Cesare Tinelli (Eds.), Vol. 9035. Springer, 52–67. DOI: http://dx.doi.org/10.1007/978-3-662-46681-0_4
[5] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. *A Tool for Checking ANSI-C Programs.* Springer Berlin Heidelberg, Berlin, Heidelberg, 168–176. DOI: http://dx.doi.org/10.1007/978-3-540-24730-2_15
[6] Darren Cofer, Andrew Gacek, Steven Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. 2012. Compositional Verification of Architectural Models. In *Proceedings of the 4th International Conference on NASA Formal Methods (NFM'12).* Springer-Verlag, Berlin, Heidelberg, 126–140. DOI: http://dx.doi.org/10.1007/978-3-642-28891-3_13
[7] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM, 238–252.
[8] Peter H. Feiler and David P. Gluch. 2012. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language* (1st ed.). Addison-Wesley Professional.
[9] Andrew Gacek. 2014. The JKind model checker. (2014). http://loonwerks.com/tools/jkind.html
[10] Susanne Graf, Roberto Passerone, and Sophie Quinton. 2014. *Contract-Based Reasoning for Component Systems with Rich Interactions.* Springer New York, New York, NY, 139–154. DOI: http://dx.doi.org/10.1007/978-1-4614-3879-3_8
[11] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.
[12] Duc Hoang, Yannick Moy, Angela Wallenburg, and Roderick Chapman. 2015. SPARK 2014 and GNATprove. *Int. J. Softw. Tools Technol. Transf.* 17, 6 (Nov. 2015), 695–707. DOI: http://dx.doi.org/10.1007/s10009-014-0322-5
[13] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (2015), 573–609. DOI: http://dx.doi.org/10.1007/s00165-014-0326-7
[14] Ian O'Neill. 2012. *SPARK fi A Language and Tool-Set for High-Integrity Software Development.* John Wiley and Sons, Inc., 1–27. DOI: http://dx.doi.org/10.1002/9781118561829.ch1