2012

Perpetual requirements engineering

Manuel Alfonso Peralta
Louisiana State University and Agricultural and Mechanical College

PERPETUAL REQUIREMENTS ENGINEERING

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Manuel Peralta
B.Sc., Pontificia Universidad Católica Madre y Maestra, 2003
M.Sc., Utah State University, 2007
December 2012

Acknowledgments

I would like to express my gratitude to the people and entities whose support had tremendous weight regarding the efforts that lead to the completion of this dissertation. First of all I would like to thank my family whose words of advice and encouragement helped me start and continue throughout this journey. I would also like to thank my advisor, Dr. Supratik Mukhopadhyay, who taught me that research often starts with ideas that seem impossible to realize in the face of the current state of the art. He, via example, also taught me that research is defined by bridging two or more fields which initially seem disconnected; research is made by marching fearlessly (and perhaps relentlessly) towards thorny problems and issues that most consider unsurmountable. Last but not least I would like to express my deepest appreciation to the government of the Dominican Republic and more specifically to the Science and Technology Ministry of Higher Education (MESCyT, its initials in Spanish) for providing me with the necessary financial support that helped me begin, continue, and successfully culminate this chapter of my life.

Table of Contents

List of Tables

List of Figures

Abstract

This dissertation attempts to make a contribution within the fields of *distributed systems*, *security*, and *formal verification*. We provide a way to formally assess the impact of a given change in three different contexts. We have developed a logic based on Lewis's *Counterfactual Logic*. First we show how our approach is applied to a standard sequential programming setting. Then, we show how a modified version of the logic can be used in the context of reactive systems and sensor networks. Last but not least we show how this logic can be used in the context of security systems.

Traditionally, change impact analysis has been viewed as an area in traditional software engineering. Software artifacts (source code, usually) are modified in response to a change in user requirements. Aside from making sure that the changes are inherently correct (testing and verification), programmers (software engineers) need to make sure that the introduced changes are coherent with those parts of the systems that were not affected by the artifact modification. The latter is generally achieved by establishing a *dependency relation* between software artifacts. In rough lines, the process of change management consists of projecting the transitive closure of the this dependency relation based on the set of artifacts that have actually changed and assessing how the related artifacts changed.

The latter description of the traditional change management process generally occurs after the affected artifacts are changed. Undesired secondary effects are usually found during the testing phase after the changes have been incorporated. In cases when there is certain level of criticality, there is always a division between production and development environments. Change management (either automatic, tool driven, or completely manually done) can introduce extraneous defects into any of the changed software life-cycle artifacts. The testing phase tries to eradicate a relatively large portion of the undesired defects introduced by change. However, traditional testing

techniques are limited by their coverage strength. Therefore, even when maximum coverage is guaranteed there is always the non-zero probability of having secondary effects prior to a change.

Chapter 1

Introduction

For Mission Critical Systems (MCS) applications, requirements dynamically change in a rapid, unpredictable, and continuous fashion. In applications such as those driving search and rescue missions, any operational expansion/contraction requires that systems be updated with minimum occurrence of secondary effects. For these application scenarios, any additional downtime resulting from upgrade of the control system leads to unacceptable disruption of service. As a result, being able to change these systems while preserving the correctness of both the changed and un-changed portions is very important. We need to develop techniques, tools, and methods that allow software engineers to specify change based on changing requirements; also, these methods and tools should be able to guarantee the correctness of the changes and the whole system after such changes have been applied.

Traditional software development methodologies assume that requirements are well understood and available, in the form of a formal or extensive specification, of the required system behavior. However, this assumption fails to hold for software that is meant to control MCS applications deployed in rapidly evolving scenarios. While it is possible to develop logically precise requirements for software computing mathematical functions, the behavior of a software system depends on extraneous factors that are not usually foreseen during its development. These include factors such as platform of deployment (e.g., the word length of the machine on which the software is run), the communication protocols used, the amount of memory available, etc. In software solving real world problems, such extraneous factors are compounded by those from the system's physical environment that expect the software to cope with changing business constraints. In some cases, given time and money, it is possible to get the original developers to update the software to meet the changed business requirements. However, updating the system through traditional methods

which consist in changing the code and then testing may introduce undesirable secondary effects. In these situations, the system must be phased out with millions of dollars in software development cost wasted.

1.1 Problem Statement

Conventional approaches used in industry cope very poorly with an scenario in which software requirements change rapidly and constantly. Traditional testing approaches operate by first changing the system's source code (hopefully within a development environment separated from the live production environment), testing the changed system's modified fragments; testing the whole system in conjunction with the changed modules and committing the change. However, even the most robust testing procedures/methodologies are constrained by the coverage limit (citation). Even for relatively simple programs, it is not easy to foresee the impact of change without first changing the source code. It follows that for larger code bases (several thousands or millions of KLOC), the problem is amplified further especially if we take into account that systems of such dimensions tend to support critical operations. Since the state of the art (traditional change management within Software Engineering) does not provide a way in which we could *foresee* the effect of a change without incorporating the changed artifacts, we are in need of tools, processes, and methodologies by which the latter may be achieved.

In large software projects with dynamically changing requirements are usually encountered in the development of MCS applications, instead of manually performing development iterations every time a requirement changes, we need to implement techniques and strategies for automatic incremental update of system or subsystem components comprising the deployed software, in response to changing requirements. We need a requirements engineering paradigm that can give an idea on how the changes would affect the system in an *a-priori* manner.

Chapter 2

Related Work

The theory of counterfactuals allows us to reason about hypothetical situations. It has been used in Philosophy, Probability Theory, and Econometrics [34] for decision making in a hypothetical environment. In Physics, it has been used for reasoning about measurements in quantum mechanics [47], [33]. The main idea exposed by Fisler et al. in [14] is to gain knowledge regarding the effects of changing access control policies before actually making such changes. The work of Fisler et al. is similar to the one presented in this paper as it tries to find the effects of a change *a priori*. The work of Chockler et al. in [9] employs counterfactual reasoning also in the context of model checking. In this instance, the authors emphasize their work in coverage issues. They use counterfactual reasoning to enhance the coverage information. This work differs from ours given that they use counterfactual logic to explore alternative scenarios whereas, in our case, we explore a single alternative version given an initial version.

Also, in [18], Groce et al. use *counterfactual theory* to detect failures, isolate errors, and aid in repairing modifications of program code in the context of model checking. They construct a model of program executions and establish a metric among them. This metric lets them analyze faulty executions by examining those executions which lie at some distance from a given faulty execution. The work of [18] relates to ours in the sense that the authors define a notion of *distance* among possible execution traces in the same sense we implicitly define the number of transformation steps between program versions. In [18], however, the authors go beyond just defining a notion of proximity and actually define, given a system execution trace, the set of neighboring traces whereas our work only characterizes a single future version, given an initial actual version of some program. The work of Ren et al. in [40] exposes a tool (i.e. *Chianti*) that analyzes two different versions of a given program and a set of test cases for such program and determines which tests are

affected due to the changes that lead from one version to another. Furthermore, for each affected test, the tool determines a set of method-level changes that most probably affected the test. The work presented in this article differs from the work in [40] in the sense that we do not require a second version and a set of test cases. Our approach just needs the changes to be expressed in our logical calculus. The correctness that pertains to the changes is decided based on the current properties of the original program and the desired future properties.

The approach given by Guo et al. in [19] exhibits a method by which change impact analysis is modeled and verified in a distributed setting. This approach is based on model checking. Their model is in essence a network of state machines that communicate either via shared variables or queues. Changes are modeled as adding and/or deleting transitions from the composite state machine that represents the distributed system. The work in [19] is similar to ours in the context of two aspects: 1) the authors are formally representing change in a system; however, their approach targets distributed computations and is based on model checking whereas our approach targets sequential computation and it is based on theorem proving; 2) this approach prunes the global state space by using *partial order reduction* in order to infer the valid transitions when a change occurs; our approach deals with change at the source code level and the validity of the change is inferred by our logical calculus. In [50], Subramaniam et al. enhance the approach shown in [19]. The changes are still represented by adding and/or deleting transitions of a composite state machine. However, this work addresses the issue of test suite coverage when changes occur. This approach detects the affected tests based on whether or not these include the affected transitions. Using formal verification techniques similar to the ones presented in [19], the authors are able to reduce the total regression test suite based on which tests are relevant after a given change. Our approach goes in a different direction by formally characterizing the source code-change and determining if the changes to the current source code version are logically consistent with its properties and the future desired properties.

The work presented in [38] uses symbolic execution to establish whether or not too source code versions are equivalent. In the negative case the proposed approach generates the *deltas* which characterize the input values that induce the behavior difference between the two versions. Our approach, being based on proof theoretic method, relies mostly on the syntactic nature of change and hence we consider two versions identical as long as they have equivalent logical characterizations. The latter also means that we are only interested on cases where our two versions (the actual and the potential version) are logically different.

Significant research has been performed in massively distributed environment-aware computing (also known as "swarm computing" [13], [30]), in particular for creating and reasoning about swarm programs. Most of these works have been focused on developing programming paradigms, tools, and languages for swarm computing. EnviroTrack [1], an object-based distributed middleware system, raises the level of programming abstraction for distributed s by providing a convenient and powerful interface to the application developer geared towards tracking the physical environment. Menezes et al. [30] study different abstractions in the field of swarms. However, none of these works are concerned with the problem of developing formal methods for building sensor-based systems that provide provable guarantees of meeting their requirements.

Roemer et. al. [42] survey middleware challenges in the area of wireless sensor networks. According to [42], adaptability and data-centric communication should be important issues in coordinating services in networks that involve wireless sensors. We augment the desirable properties of coordination frameworks for wireless sensor networks stated in [42] with the capability of intelligent data/service fusion.

The work shown in [11] uses Datalog rules to express security policies. Their approach is based on model checking. The model used is a composite state machine. The states in this automaton are relational structures and the transitions are labeled by events and policy rules. The transitions of this automaton are restricted to be those that preserve the security policy's integrity (i.e. avoid a security breach). Furthermore, their approach requires two policies as inputs. To replace a security

policy, the old and the new policies need to be linked by the *policy containment* relationship which amounts to the new policy subsuming the decisions that pertain to the old one. The semantics for our agent-based networks are also based on security automatons whose transitions formally guarantee that security and safety properties are respected. Additionally, our approach does not require two versions of the access control matrix. Instead, we express the new version in terms of the desired changes. Besides, while this approach heavily relies on model theory, ours is mainly proof-theoretical. It is widely known that the size of the model is a decisive factor when considering the space-efficiency of model-checking. In many real-world systems the size of the model may be so large a model-checking-based approach may turn out to be prohibitive.

Ginsberg [15] pioneered the application of counterfactuals for reasoning about change in Artificial Intelligence. In particular, [15] applies the logic of counterfactuals to hardware design problems. Our work, however, deals with reasoning about change in terms of security and safety in sensor networks where arbitrary changes can compromise the system.

The approach followed in [10] targets large customized Enterprise Resource Planning (ERP) systems. They analyze two different customization code versions. They apply static analysis in the form of program slicing to examine two subsequent code versions. The change impact is denoted by the difference in the outputs when the two versions are analyzed using a given set of inputs, i.e., there is a change if the two versions are not *observationally equivalent*. This approach, similar to the work reviewed in the previous paragraphs, still requires two different versions. Although this approach is clearly applicable to large scale ERP systems, it lacks the ability to manipulate a representation of the source code (security model in our case) and deduce whether or not there would be a security breach if the changes were incorporated.

Chapter 3

Literature Review

Zhang, Ji; Cheng, Betty H.C. "Model-Based Development of Dynamically Adaptive Software". *Proceedings of the International Conference of Software Engineering (ICSE'06) 2006*. pp. 371-380. [54]

The authors propose an approach for dynamic software update which could be described as generic. This approach's contribution consists on regarding the dynamic update problem from a multi-threaded perspective. After all, in conventional software development, it is the integrated effort of a team of programmers that serves the purpose when a software system needs to undergo any sort of non-trivial modification.

Other interesting characteristics of this approach rely on the fact that it permits dynamic update while maintaining current system state, while other approaches (those based on process-algebraic methods) are not capable of achieving this. The latter is related to the fact that, according to the authors, other techniques tend to conceive the dynamic update problem from a structural perspective; the latter may not be so useful, given that dynamic update mechanisms should strive for transparency.

Zhang, Ji; Cheng, Betty H.C. "Specifying Adaptation Semantics" *Workshop on Algorithms and Data Structures (WADS'05) 2005*. pp. 1-6. [53]

In this paper, the authors present a specification language based on linear temporal logic. They characterize the system states in which updates can be performed as mutually consistent assertions in temporal logic. Also, the authors introduce a graph based complementary technique in order to make their approach more understandable in practice.

Although effective, their approach may be considered as not entirely efficient, due to the fact that the algorithms that undertake the verification of their models have exponential complexity.

Kogekar. S., Neema, S. et al."Constraint-Guided Dynamic Reconfiguration in Sensor Networks" *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN'04) 2004*. pp. 379-387. [26]

This article presents an approach by which sensor network - based software can be dynamically adapted. The authors use a technique by which they can characterize the operation environment (all allowed values for the control parameters) as formal constraints.

System states are represented as specific points within the operation space and reconfiguration or dynamic update is regarded as transitions along those points. Thus, software update is regarded as the exploration of the operation space; given the nature of the parameters for these types of systems (Sensor Networks) and without adequate heuristics, this problem may imply some level of combinatorial explosion.

Poizat, P., Salan, G. and Tivoli, M. "On Dynamic Reconfiguration of Behavioral Adaptations. *Proceedings of the 3rd International Workshop on Coordination and Adaptation Techniques for Software Entities* (WCAT'06) 2006. pp. 61-69. [39]

This article considers the problem of dynamic update from an additive perspective. Modifications on the software system will be done by integrating software components called "adaptors" whose behavior will change based on changes in the set of requirements. The specific role of the adaptors is to modify the interfaces exposed by system modules.

The authors also discuss an important concept in the realm of dynamic update, they denominate it the silent behavior portion. By the latter they mean that, in the face of a change in the system specification, the system part(s) which suffer(s) the change should be isolated and stopped. The latter partially guarantees that, after the update, the system invariants will still be consistent.

Stoyle, G., Hicks, M. "Mutatis Mutandis: Safe and Predictable Dynamic Software Updating". *Symposium on Principles of Programming Languages (POPL'05)* 2005. pp. 183-189. [49]

In this article, the authors consider the problem of dynamic update from a type theoretical perspective. The authors develop a calculus-based formalism (Proteus) which characterizes the changes in software behavior as changes in the types in a function signature. The change is char-

acterize by a set of type transformer functions, by which, explicit use of some changed data type are casted into the modified data type.

Taenzer, G., Goedicke, M., et al. "Dynamic Accommodation of Change: Automated Architecture Configuration of Distributed Systems." *14th International IEEE Conference on Automated Software Engineering. (ASE'99)* 1999. pp. 287-291. [51]

In this article, the authors present an approach by which, module interfaces are modified via graph transformation methods. Dynamic update, thus, is modeled as sequence of graph transformations where a particular graph represents the current or potential state of a give distributed system.

One of the major contributions regarding this paper is the definition of a quiescent state. In order to apply an update to some set of modules within a software system, all of its components (nodes) must be in a consistent state and all communication should have been suspended. It is only under these conditions that the dynamic update is guaranteed not to violate the system's invariants.

Shen, j., Xi, S,. Huang, G. Jiao, W., et al. "Towards a Unified Formal Model for Supporting Mechanisms of Dynamic Component Update". 4th Joint Meeting of the European Software Engineering Conference and the Symposium of the Foundations of Software Engineering. (ESEC-FSE'05) 2005. pp. 80-89. [46]

In this article, the authors try to envisage the dynamic update problem from a software-architectural point of view. Their solution to the problem is realized as a special kind of connector; where the idea of connector is directly borrowed from the work by Shaw and Garland. Concurrently, the authors try to abstract their approach from any platform/hardware-dependent issues; hence, they define the behavioral dynamics of their approach using CSP (Communicating Sequential Processes).

One particular aspect about this approach is the authors' intention to make a case about the feasibility of it based on real application to well-known distributed systems frameworks, namely, CORBA and J2EE. They also consider how to apply their approach to a web services-based architecture. A common underlying issue in these three separate cases was the problem of maintaining and transferring state when updating the system.

Batista, T., Rodrguez, N. "Dynamic Reconfiguration of Component-Based Applications." *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'00)* 2000. pp 32-39. [3]

In the context of this article the authors put the runtime emphasis on dynamic update. This approach relies on the dynamic nature of interpreted languages in order to achieve automatic system reconfiguration. According to the authors, the main reason for using an interpreted language to achieve this is because doing so will eliminate the need for compilation phases in between software updates.

This approach also relies on the CORBA standard to guarantee distributed model encapsulation via the separation between interface and specification. It also achieves dynamic update due to a characteristic inherent to the CORBA standard; this characteristic consists on partially-expressive interfaces. Given that interfaces are not strongly expressed, behavioral variations are determined by characterization of the type of messages that the modules use to communicate; functionality is thus determined at runtime which smoothly implies dynamic update capabilities.

Chen, W., Hiltunen, M., Schlitchting, R. "Constructing Adaptive Software in Distributed Systems." *Proceedings of the 21st International Conference on Distributed Computer Systems (ICDCS'01)* 2001. pp. 413-420. [8]

This article situates the problem of dynamic update within the realm of sensor networks. Similar to other approaches, it conceptualizes the dynamic update problem as a finite sequence of state transitions. The state space is denominated as the operation space. Given that system requirements are characterized as all the possible values of the operational parameters, the operation space is the feasible zone determined by the union of all constraints induced by the legal values of all operational parameters.

Thus, the set of all possible configurations is modeled as the set of system components (modules) and their alternative configurations along with the relationships among modules. The latter intuitively indicates that the dynamic update process is plagued with inefficiency due a combinatorial explosion derived from all the possible configurations available in the operation space.

Gupta, D,. Jalote, P. And Barua G. *A Formal Framework for On-Line Software Version Change*. IEEE Transactions on Software Engineering. Volume 22, Issue2. 1996. pp. 120-131. [20]

This article exposes the dynamic update problem from a software version change perspective. Hence, dynamic update is achieved, by source code substitution at run-time. The authors assume the existence of correct source code version before and after the change. This approach, although dated, elaborates a very illustrative model that deals with two important aspects in dynamic update: overall correctness and consistent state transitions. Also, the authors suggest other important issues such as quiescent state of the system when the update process is being executed.

One particular issue of this approach is that it relies on an imperative programming scheme to model the dynamic update process. Also, the scope for this approach does not go beyond a stand-alone sequential setting, where a centralized processor and the idea of global time can be relied on. They succinctly suggest that their approach could be extended to a distributed setting. However, as it is shown in standard distributed system literature, just the idea of a consistent distributed state is somewhat complicated to achieve.

The authors emphasize on the fact of the instrumentality of the module invariant. Which they define as the set of conditions with denote the state in which is safe to a modification on the system. This approach relies on the programmer to specify this invariant.

Bierman, G., Hicks, M. et al. "Formalizing Dynamic Software Updating". *In Proceedings of the 2nd International Workshop on Unanticipated Software Evolution (IWUSE'03)* 2003. pp. 1-17. [7]

This article presents and approach that strives to present a solution to the dynamic update problem via a λ Calculus-based method. Their main motif is to provide an update mechanism which, aside from being simple provides precise mathematical semantics.

Their approach implies a two-fold methodology. In one aspect they provide the syntax of their update calculus; which, according to the authors is a first order, simply-typed, call-by-value lambda calculus. The other side of their approach is the semantics, which is composed of a set of reduction rules whose purpose is to realize an expression transformation mechanism.

The authors justify their approach by asserting that it follows two principles which ultimately guarantee correctness: Type Safety by use the module wide signature (the set of types of the module's public members) they can implement a subtype operator to distinguish between different module versions at runtime. The other principle is Correctness and this one is weakly justified given that the authors assert that when an update happens, the updating actions are not initiated while ongoing computations are active; however, this is asserted by way of an example. Also, they justify that the latter case is a rough approximation to a Hoare-based scheme, in which a mechanism of pre and post conditions is used to guarantee consistencies across updates.

Zhang, S., Huang L. "Formalizing Class Dynamic Software Updating". In *Proceedings of the 6th International Conference on Quality Software (QSIC'06)* 2006. pp. 403-409. [55]

This article shows the type theoretical side of dynamic update problem and also emphasizes on the object oriented aspect of it. The authors assert that although some approaches deal with the dynamic software update (DSU) problem from an object oriented point of view. These approaches still lack an adequate degree of rigor and hence their type safety cannot be objectively characterized.

Given that the objective of this approach is rigor, the authors show that their solution is adequate via a calculus for the DSU problem. Their result section is a set proofs (theorems and their correspondent lemmas) on the consistency induced by each and one of their type judgments (assertions in the form of theorems proper of type theory). Their approach emphasizes on type safety and thus, the proofs shown in the article are instrumental for the validity of their approach.

An interesting twist shown in this approach is the use of FeatherWeight Java [24]. According to the references FWJ is a reduced version of the JAVA programming language; and its purpose rests on making type theoretical proof about JAVA programs in an easy manner.

Duggan D. "Type-Based Hot Swapping of Running Modules". In P*roceedings of the International Conference on Functional Programming (ICFP'01) 2001*. pp. 62-73. [12]

In this article, the authors expose an approach which conceptually takes advantage of the majority of existing object oriented languages. The main contribution of this article is the novel use

of reflection capability on a given language to realize what they call Hot Swapping; which is defined as changing the implementation of a given module and have it be transparent to its respective clients.

This article improves on an approach which establishes isomorphism between old types and new types. This scheme is based on the co-existence of instances of the new and old types as a result of a modification on the types of formal parameters, a return type for a function or, in general, the interface of a given module. Their contribution is based on considering an equivalence relation between the old type and the new type which differs from the referred approach in that a subtype relationship is not symmetric while the equivalence is.

This approach requires the existence of a pair of isomorphisms (structure-preserving transformations) between the old and new type and hence, a way to implement the coexistence of corresponding types in an update. More precisely, if a type isomorphism can be constructed the types are structurally equivalent and hence an instance of one may be an instance of the other.

Chapter 4

Code-Change Impact Analysis Using Counterfactuals

4.1 Introduction

In this chapter we present a framework for *what-if* analysis of programs based on Lewis' theory of counterfactuals [29]. The framework can be used to statically perform change-impact analysis for source code. It enables us to verify assertions about a changed version of the program without actually incorporating the changes. We present a logical calculus that precisely characterizes structural modifications to source code and their impact on the behavior of the program.[1]

In the software development life-cycle, the majority of costs are usually incurred during the testing and maintenance phase. Addition of new features, optimizations, refactoring and fixing of defects necessitates modifications of the software system's source code. While a combination of formal methods and testing as it has been shown in [2], [4] can lead towards a defect-free software, aggressive optimizations and other modifications can undo the quality resulting from thousands of hours of verification and validation efforts. In many cases, such optimizations and modifications are done without a complete understanding of the system (specially in the cases of parallel programs). Due to the complexity and size of today's software systems, completely understanding a system by code review is out of question. Regardless of how a programmer modifies the program, extensive regression tests are needed in order to verify that (1) the new program version still complies with its correctness constraints and/or (2) the new version complies with the properties implied by the new requirements. While regression tests make sure that the modified software system passes the test cases, defects that were detected through static analysis techniques and

[1] Portions of this chapter were published in:

Manuel Peralta and Supratik Mukhopadhyay. Code-Change impact analysis using counterfactuals. *Computer Software and Applications Conference, Annual International*, 0:694–699, 2011.

subsequently removed may creep-in as a result of modifications and will remain undetected by regression tests. We need automated tool-support that enables us to understand software systems and the effects of the changes on them; automated tools should be able to statically determine whether a set of modifications applied to a software system resulted in modifications of its semantics.

In this chapter we present a framework for *what-if* analysis of programs based on Lewis' theory of counterfactuals [29]. The framework can be used to statically perform change-impact analysis for source code. It enables us to verify assertions about a changed version of the program without actually incorporating the changes. We present a logical calculus that precisely characterizes potential structural modifications to source code and their impact on the program's behavior. Our framework blends model theoretic verification techniques with proof theoretic ones. The space of program versions under modifications is treated as a *Kripke* structure with neighborhood semantics. The completeness and soundness theorems for counterfactual logic described below are used to transfer model theoretic facts to proof theoretic ones and vice versa. We use the expression *counterfactual logic* to precisely mean *propositional counterfactual logic*.

One can argue that it is possible to actually apply the modifications to the program and then statically analyze the modified program to check if it conforms to the expected behavior. However, if we expect a certain behavior to emerge after the changes are applied and it turns out, based on the result of static analysis, that the applied changes do not enforce the desired behavior, the entire effort spent in modifying the code is wasted. Our framework allows a programmer to think of alternate ways of implementing a program and prevents waste of efforts in writing code that does not meet the objectives.

4.2 Counterfactual Theory

The logic of counterfactuals helps us reason about assertions that are not matter of fact. In [29] Lewis provided a sound and complete proof system and proved its decidability. Based on the logic of counterfactuals we derive a logical calculus that allows us to assert properties that would hold

for a future version of a given program and verify that these would indeed hold if the changes needed to obtain that version were actually implemented.

4.2.1 The Language of Counterfactual Theory

In coherence with [29] we will briefly introduce the language regarding the logic of counterfactuals below [2]:

$$\phi ::= p_i | \neg\phi | \phi \wedge \phi | \phi \vee \phi | \phi \rightarrow \phi | \phi \boxdot\!\!\rightarrow \phi$$

The counterfactual sentence $\phi \boxdot\!\!\rightarrow \psi$ should be read as: **if it had been the case that ϕ, it would have been the case that ψ**. Thus, if we had an assertion whose antecedent ranged over the properties of some given program and also the changes needed to produce a new version and whose consequent ranged over the properties that a new version would have, then we could use counterfactual logic to code such an assertion.

4.3 Program Transform Model

In this section we provide a formal exposition of the two-fold model we use to formalize counterfactual change. On one side (subsections: 4.3.1, 6.4.3) we introduce our the part of our logical calculus that let us encode change at the source code level based on higher order logic. On the other side (subsections: 7.4.4, 7.4.5) we provide the semantics of our approach using Lewis' approach [29] based on *neighborhood semantics* of Kripke.

4.3.1 Logical Description of a Program

The programs we will be dealing with are initially well formed strings over the following syntax [3] shown in Figure 4.1.

[2] p_i denotes a propositional variable.

[3] For the sake of brevity the non-expanded non-terminals shall be interpreted as usual.

⟨Stat⟩ ::= **skip** | ⟨AsgStmt⟩ | ⟨IfStmt⟩ | ⟨LoopStmt⟩ | **begin** | **end** | **else**
⟨AsgStmt⟩ ::= ⟨VarName⟩ **:=** ⟨IntExp⟩
⟨IfStmt⟩ ::= **if** ⟨BoolExp⟩ **then begin**
⟨LoopStmt⟩ ::= **for** ⟨AsgStmt⟩ **to** ⟨IntConst⟩ **do begin**

Figure 4.1: Restricted grammar for our ALGOL-like language

Furthermore, we define $U \subseteq \mathbb{N}$ to be a prefix set of the natural numbers. Moreover, let $Stat$ also denote the set of strings obtained from the grammar show in Figure 4.1. Hence, we can further define our set of well-defined programs as the following:

$$\mathcal{P} = \{f : U \to Stat\}$$

Any program $f_0 \in \mathcal{P}$ is just a mapping from a subset of the natural numbers (which denote line numbers) to the set of well-formed strings from the grammar in Figure 4.1. Since $\mathcal{P}$ is a function space, it may contain more functions that the ones that interest us, hence it is necessary to characterize the class of functions we are interested in.

First of all, our programs need a relation among line numbers to capture the notion of *nested statements* or more intuitively, the notion of matching `begins` and `ends`. For each program $f \in \mathcal{P}$ we have a relation $R \subseteq U \times U$ such that:

1. $(\forall u \in U) f(u) = \langle IfStmt \rangle$ or $f(u) = \langle LoopStmt \rangle$
2. $(\exists v \in U)(u < v) \wedge (f(v) = end) \wedge (u, v) \in R$
3. $(\forall u, v \in U)(u, v) \in R \to (w, v) \notin R$ where $(\forall w \in U) w \neq u$
4. $(\forall u, v, w, x \in U)(u, v) \in R \wedge (u \leq w \leq v) \wedge (w, x) \in R \to (x \leq v)$

Furthermore, we choose to model possible program transformations using the following rule:

$$
\begin{aligned}
(\forall u \in U)(u \neq u_i) \wedge (u \neq u_j) \quad &\rightarrow \quad (f(u) = f'(u)) \\
&\wedge \quad (f(u_i) = f'(u_j)) \\
&\wedge \quad (f(u_j) = f'(u_i))
\end{aligned}
$$

The latter expression denotes the existence of a *new* program $f' \in \mathcal{P}$ that happens to differ from our original program $f \in \mathcal{P}$ only by swapping two statements (i.e. u_i and u_j).

4.3.2 Program Transformers

In the context of Hoare Logic the notion of *predicate transformer* [17] is widely known. A predicate transformer may be regarded as a first order logic formula which via *existential quantifier elimination* produces the weakest precondition for a given command and its respective post condition. Following the same notion of predicate transformers we have thought of specifying our programs and their transformations as *Program Transformers*.

Let Ψ_P denote the rules given in section 4.3.1 and let Ψ_S denote the rule shown below. Thence, we can formally express our *program transformer expression* as:

$$\Psi_T \triangleq (\exists f)(\exists R)\Psi_P \wedge \Psi_S$$

Notice that Ψ_T is a *second order logic* formula since we are quantifying over one function symbol and one relation symbol. Also, let us assume that we have a fixed program $f_0 \in \mathcal{P}$. By definition, f_0 is a finite list of statements in $Stat$. Thence, we can logically express $f_0 \colon U \rightarrow Stat$ by a finite conjunction of equalities as it is expressed below:

$$\Phi_f \triangleq \bigwedge_{i \in U} f(i) = Stat_i$$

Where Φ_f is the logical formula that represents f_0 and $Stat_i$ is a statement in $Stat$. Therefore, we can "apply" Ψ_T to Φ_f by joining them by conjunction which produces the following formula:

$$\Psi_T \wedge \Phi_f \triangleq (\exists f)(\exists R)\Psi_P \wedge \Psi_S \wedge \Phi_f$$

Also, do notice that, we may only eliminate the quantified f in the latter formula. The resulting formula will be the logical expression that denotes our new version of f_0, namely, f', however, notice that, since we cannot eliminate the quantified variable R (as elimination of n-ary quantified relation symbols is an open problem), the resulting expression is not quite the corresponding formula for f' in the same manner Φ_f denoted f_0.

Furthermore, notice that our notion of a relation $R \subseteq U \times U$ may be critiqued as being too vague. To the latter we assert that for each $f \in \mathcal{P}$ R will be the least relation that satisfies Ψ_P. Moreover, R's unique characterization may be provided by the following second order logic formula:

$$\exists R(\Psi_P^R \wedge (\forall R' \Psi_P^{R'} \wedge (\forall x, y R'(x, y))$$
$$\rightarrow R(x, y)) \rightarrow (\forall u, v R(u, v) \rightarrow R'(u, v)))$$

Where $\Psi_P^{R_0}$ denotes the same formula Ψ_P with R substituted for R_0. Intuitively, the latter formula means that R is the least relation that satisfies Ψ_P since if there is any ohter relation R' that also satisfies Ψ_P then both relations are equivalent. More simply, the latter assertion may be formalized using set-theoretic notation, this yields a more succinct expression.

$$\exists R R \models \Psi_P \wedge (\forall R' R' \models \Psi_P) \wedge (R \subseteq R') \rightarrow (R' \subseteq R)$$

4.3.3 Kripke Versioning Model

In [29] the author provides the semantics of his counterfactual propositional logic using a multiple-world interpretation. In that same manner we have chosen to interpret our program transformation. In our case, each program version will represent a world. When we applied a *program transformer* (as it is defined in the last section) we obtain a new version. Below, we provide a formal interpretation based on a Kripke model.

Definition 4.3.1 (Kripke Version Model) *A Kripke Version Model $\mathcal{R}$ is a triple $\langle \mathcal{P}, \Rightarrow, P_0 \rangle$ where:*

1. *$\mathcal{P} = \{P_k\}_{k \in \mathbb{N}}$ is the set of all n-line programs which are the different program versions.*
2. *P_0 is the initial program.*
3. *$\Rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ is a binary relation defined the set of all possible program versions. Where $\Rightarrow$ is the smallest relation such that the following properties hold:*
   (a) *$s_i \leftarrow s_i \triangleq$ statement s_i is left unchanged. This stands for the do nothing transformation.*
   (b) *$s_i \leftarrow s_j \triangleq$ statement s_j replaces statement s_i, where $s_j \in P_k$. We usually call this primitive transformation, a swap.*
   (c) *$s_i \leftarrow s_j \triangleq$ statement s_j replaces statement s_i, where $s_j \notin P_k$. Thence, s_k is a new statement.*
   (d) *$(\forall i) s_i \in P_k$ can be changed only once.*

Furthermore, we assume that the relation $\Rightarrow$ complies with the properties of *reflexivity*, *symmetry*, and *transitivity*. Below, we justify each property based on the latter definition of $\Rightarrow$:

1. **Reflexivity:** For any program $P_i \in \mathcal{P}$, it is obvious that the *do-nothing transformation* will yield that any program can be transformed into itself. Therefore, $P_i \Rightarrow P_i$ given that for all $s_j \in P_i$, $P_i = P_i[s_j/s_j]$

2. **Symmetry:** For any programs $P_i, P_j \in \mathcal{P}$ any of the above transformations can be reversed and thence, $P_i \Rightarrow P_j$ implies $P_j \Rightarrow P_i$.

3. **Transitivity:** For any programs $P_i, P_j, P_k \in \mathcal{P}$, applying two or more transformations to a program will yield intermediate versions; this is equivalent to transforming the initial version by composing the transformations into one. Thus, $P_i \Rightarrow P_j$ and $P_j \Rightarrow P_k$ imply that $P_i \Rightarrow^+ P_k$. Where $\Rightarrow^+ \triangleq \Rightarrow \circ \Rightarrow^{n-1}$ and $n > 1$.

4.3.4 Interpreting the Counterfactual Implication

As it was stated earlier, the purpose of our model is to help interpret assertions in the language of counterfactual logic. Let P_0 denote our given source program. Also, let us assume we had a counterfactual assertion, namely $\phi \boxdot\!\!\rightarrow \psi$ in which:

- ϕ stands for assertions regarding P_0 and some transformation $s_i \leftarrow s_j$ that implies that $P_1 = P_0[s_i/s_j]$
- ψ stands for assertions regarding P_1

Thence, following the model-theoretic interpretation proposed by Lewis in [29], our version of the counterfactual implication is interpreted as:

$$\mathcal{R} \models \phi \boxdot\!\!\rightarrow \psi \ . \tag{4.1}$$

Where $\mathcal{R}$ denotes our previously defined *Kripke Versioning Model*. Moreover, letting α_i, β_i denote propositional statements about the structure of P_i and P_j respectively, then, we can state that:

$$\phi \triangleq (\bigwedge_{i=1}^{n} \alpha_i) \wedge (P_i \Rightarrow^+ P_j)$$
$$\psi \triangleq \bigwedge_{j=1}^{m} \beta_j$$

Where $\Rightarrow^+$ denotes the positive/transtive closure for the relation $\Rightarrow$. Furthermore, given an initial program version, namely P_0, we produce several versions by applying one or more transformations to it. In the context of a counterfactual assertion, the properties regarding the current version and the changes applied to it (in order to produce a new version) imply properties possessed the new version and hence:

$$\mathcal{R} \models \phi \boxdot\!\!\rightarrow \psi \quad \triangleq \quad (\exists_{\min} k \in \mathbb{N})(\bigwedge_{i=1}^{n} \alpha_i) \\ \wedge (P_0 \Rightarrow^k P') \rightarrow (\bigwedge_{j=1}^{m} \beta_j) \ .$$

The latter should be interpreted as there exists a minimal number of transformation steps such that given the properties of our initial program P_0 (namely, $\bigwedge_{i=1}^{n} \alpha_i$) and the transformation between the two program versions implies the desired properties of the future program version (namely, $\bigwedge_{j=1}^{m} \beta_j$).

4.4 Fragment of Counterfactual Logic

Since our final objective is to produce an algorithm and a tool to enable us to reason about the properties that will hold for future versions of a given program, we need a logical calculus as a principal enabling component which will permit us infer these properties in a mechanical manner. Hence, we present below a proof-theoretical fragment that pertains to the logic of counterfactuals as it is presented in [29]:

1. $\dfrac{}{\phi \boxdot\!\!\rightarrow \phi}$ Reflexivity rule
2. $\dfrac{\phi \boxdot\!\!\rightarrow \psi}{\phi \rightarrow \psi}$ Counterfactual-Elimination rule
3. $\dfrac{\vdash \lambda \rightarrow \psi}{\vdash (\phi \boxdot\!\!\rightarrow \lambda) \rightarrow (\phi \boxdot\!\!\rightarrow \psi)}$ Conditional Deduction rule
4. $\dfrac{\vdash (\phi \rightarrow \chi) \wedge (\chi \boxdot\!\!\rightarrow \psi)}{\vdash \phi \boxdot\!\!\rightarrow \psi}$ Partial Transitivity theorem

Notice that the last statement above is a theorem and not a rule. Hence, we are required to present a brief proof which will assure its validity. Such proof is given below[4]:

1. $(\phi \rightarrow \chi) \wedge (\chi \boxdot\!\!\rightarrow \psi)$ Hypothesis
2. $\chi \boxdot\!\!\rightarrow \psi$ $\wedge$-Elimination in 1
3. $\chi \rightarrow \psi$ Counterfactual Elimination rule in 2
4. $\phi \rightarrow \chi$ $\wedge$-Elimination in 1
5. $\phi \rightarrow \psi$ Hypothetical Syllogism in 3, 4
6. $(\phi \rightarrow \psi) \rightarrow [(\phi \boxdot\!\!\rightarrow \phi) \rightarrow (\phi \boxdot\!\!\rightarrow \psi)]$ Conditional Deduction rule
7. $(\phi \boxdot\!\!\rightarrow \phi) \rightarrow (\phi \boxdot\!\!\rightarrow \psi)$ Modus Ponens in 5, 6
8. $\phi \boxdot\!\!\rightarrow \phi$ Reflexivity rule
9. $\phi \boxdot\!\!\rightarrow \psi$ Modus Ponens in 7, 8
10. $(\phi \rightarrow \chi) \wedge (\chi \boxdot\!\!\rightarrow \psi) \therefore \phi \boxdot\!\!\rightarrow \psi$ Conditional Proof from 1 to 9

4.5 First Example

Assume that we are provided with the simple program shown in Figure 4.2. We introduce a set of propositions to characterize different statements of the program.

4.5.1 Program Structure Predicates

The objective of using *counterfactual logic* is to decide upon assertions about the properties of potential program versions and their structure. In the context of this example, we will need propositional assertions like the ones below:

[4]We saw fit to include this proof in the chapter as this theorem is not part of the work presented in [29].

```
function foo(var x : Integer) : Integer
    var
1       y : Integer := 1;
2       z : Integer := 0;
    begin
3       if x > 0 then
4           z := 2*y;
5       else
6           z := 2*y + 1;
7     foo := z
    end
```

Figure 4.2: Example source code

- $\alpha \triangleq f(1) = \texttt{y} : \texttt{Integer} := \texttt{1}$
- $\beta \triangleq f(2) = \texttt{z} : \texttt{Integer} := \texttt{0}$
- $\gamma \triangleq f(3) = \texttt{if x} > \texttt{0 then}$
- $\delta \triangleq f(4) = \texttt{z} := \texttt{2} * \texttt{y}$
- $\epsilon \triangleq f(5) = \texttt{else}$
- $\pi \triangleq f(6) = \texttt{z} := \texttt{2} * \texttt{y} + \texttt{1}$
- $\rho \triangleq f(7) = \texttt{foo} := \texttt{z}$

The latter propositions imply that we can denote a program by a conjunction of a finite number of propositions like the ones shown above. Hence,

$$P_0 \triangleq \alpha \wedge \beta \ldots \wedge \rho \quad . \tag{4.2}$$

Potential program versions will also be denoted by conjunction of propositions about the program's structure. Hence, a new version of P_0 could be a program where γ and δ hold true. Furthermore, we can use our *logical program description* expression to specify the structure a new version may have, based on a given program which, in this case, is P_0. Therefore,

$$\begin{aligned}(u \neq 4) \wedge (u \neq 6) \quad &\rightarrow \quad (f(u) = f'(u)) \\ &\wedge \quad (f(4) = f'(6)) \\ &\wedge \quad (f(6) = f'(4))\end{aligned} \tag{4.3}$$

Where f' denotes the member of $\mathcal{P}$ which denotes P_1 our new version of P_0. Hence, P_1 is the version constructed from P_0 by swapping the statements labeled by D and E. Therefore, according to our definition for the *versioning model* $\langle \mathcal{P}, \Rightarrow, P_0 \rangle$, then $P_0 \Rightarrow^+ P_1$. Furthermore, we can see that the change applied to P_0 may affect its behavior. Initially, from Figure 4.2, we can infer that whenever the input is greater than 0, the output of this program is an even number and it will be an odd number otherwise. Now, if we take P_1 into account, we see that the latter is no more the case and now it holds that if the input is greater than 0, the output is odd and it will be an even number otherwise.[5]

However, the latter assertions cannot be fully accepted since they fall in one of the following categories: 1) they are assertions regarding P_0 and hence, we would have to verify its code in order to prove them; 2) they are assertions regarding P_1 and we do not have the source code for it (although we know precisely how to transform P_0 in order to get P_1). Therefore, we can assert that the latter still bares the question of how can we infer properties of non-existent programs? The following sections will answer this query.

4.5.2 Counterfactual Proof

In this section we will show how to use the *logic of counterfactuals* as a calculus to prove the following assertion.

Given a program $\mathbf{P_0}$, if we *had* swapped the contents of statements $\mathbf{D}$ and $\mathbf{E}$ then it *would have been the case that* whenever the input is a positive integer, then the output is an odd integer.

[5]For the sake of simplicity, we will employ the notation $P \triangleq P'[s_i/s_j]$ to denote a swap in terms of *program transformers*.

We can see that the latter sentence is actually a *counterfactual implication* by definition and thus, we can encode it in order to express it in our already-defined language. Hence, we can write the following:

$$P_0 \wedge \mathtt{swap}_{4,6}^{P_0} \boxdot\!\!\rightarrow [(x > 0) \rightarrow (2 \not| z)] \ . \tag{4.4}$$

where $\mathtt{swap}_{4,6}^{P_0}$ stands for the proposition that denotes interchanging the lines 4 and 6 in P_0 and hence it also shortly denotes the *program transformer* that yields P_1 from P_0. Our main objective is to show that the latter assertion is actually a theorem in the logic of counterfactuals. In order to do so we need the proof theoretical fragment we proposed earlier and some lemmas and theorems we will present in the following sections.

4.5.3 Facts Based on Model Theory

Some of the assertions we will use are firmly grounded in the *program versioning model* we defined earlier. Thence, let $\mathcal{R} = \langle \mathcal{P}, \Rightarrow, P_0 \rangle$ be the model we defined before. Thus, we claim that the following assertions are valid in this model:

1. $P_0 \rightarrow \alpha \wedge \beta$
2. $(\alpha \wedge \beta \wedge \mathtt{swap}_{4,6}^{P_0}) \boxdot\!\!\rightarrow \gamma \wedge \delta$

The first assertion follows trivially from the definition of P_0 as a proposition based on the structure of its source code. The second assertion is based on the fact that the future version P_1 is based on valid transformations applied to P_0, more precisely $P_1 \triangleq P_0[s_4/s_6][s_6/s_4]$ which implies that $P_0 \Rightarrow P_1$. Hence, we have a minimal number of transformation steps that, based on the structure of P_0 yielded the structure of P_1 which in turn implies that $\gamma \wedge \delta$ is a true assertion. The latter is the model theoretical justification for the second assertion above. Thence, by completeness, the two assertions above are theorems of our *counterfactual logic*.

In what follows we will divide the proof regarding the assertion (4.4) in three claims each one representing an intermediate steps that will ultimately lead to the proof of (4.4).

4.5.4 First Claim : $(\delta \wedge \gamma) \rightarrow [(\mathbf{x} > \mathbf{0}) \rightarrow (\mathbf{2} \not| \mathbf{z})]$

Our general proof strategy is based on the *Conditional Deduction rule* whose deduction rule we give below for reference purposes:

$$\frac{\vdash \lambda \rightarrow \psi}{\vdash (\phi \boxdot\!\!\rightarrow \lambda) \rightarrow (\phi \boxdot\!\!\rightarrow \psi)} \ . \tag{4.5}$$

We provide the necessary instantiation that this case requires:

$$\begin{aligned} \lambda &\triangleq \gamma \wedge \delta \\ \phi &\triangleq \alpha \wedge \beta \wedge \mathtt{swap}_{4,6}^{P_0} \\ \psi &\triangleq (x > 0) \rightarrow (2 \not| z) \end{aligned}$$

Hence, the next step is to prove the first claim, which in this case is $(\delta \wedge \gamma) \rightarrow [(x > 0) \rightarrow (2 \not| z)]$. In order to do so, let us assume that the propositions δ and γ hold true. Moreover, let us assume that the program's input is a positive number, i.e., $x > 0$. Thence we will use *predicate transformers* as it is shown in [17] in order to deduce the required implication. The predicate transformers' rule are given below:

$post_A \triangleq \exists_{\langle x,y,z \rangle}(x' = x') \wedge (y' = y) \wedge (z' = z)$

$post_B \triangleq \exists_{\langle x,y,z \rangle}(x' = x') \wedge (y' = y) \wedge (z' = 0)$

$post_C \triangleq \exists_{\langle x,y,z \rangle}[(x > 0) \wedge (x' = x') \wedge (y' = y)$

$\wedge (z' = z)] \vee [(x \leq 0) \wedge (x' = x') \wedge (y' = 1) \wedge (z' = z)]$

$post_D \triangleq \exists_{\langle x,y,z \rangle}(x' = x') \wedge (y' = y) \wedge (z' = 2y + 1)$

$post_E \triangleq \exists_{\langle x,y,z \rangle}(x' = x') \wedge (y' = y) \wedge (z' = 2y)$

$post_F \triangleq \exists_{\langle x,y,z \rangle}(x' = x') \wedge (y' = y) \wedge (z' = z)$

Applying the predicate transformer functions to the proposition $x > 0$ we get

$post_A(x > 0) \triangleq (x' = x) \wedge (y' = 1) \wedge (z' = z) \wedge (x > 0)$

$\rho_1 \triangleq (x' > 0) \wedge (y' = 1)$

$post_B(\rho_1) \triangleq (x' = x) \wedge (y' = y) \wedge (z' = 0)$

$\wedge (x > 0) \wedge (y' = 1)$

$\rho_2 \triangleq (x' > 0) \wedge (y' = 1) \wedge (z' = 0)$

$post_C(\rho_2) \triangleq (x < 0 \vee x \geq 0) \wedge (x' = x) \wedge (y' = y)$

$\wedge (z' = z) \wedge (x > 0) \wedge (y = 1) \wedge (z = 0)$

$\rho_3 \triangleq (x' > 0) \wedge (y' = 1) \wedge (z' = 0)$

$post_D(\rho_3) \triangleq (x' = x) \wedge (y' = y) \wedge (z' = 2y + 1)$

$\wedge (x > 0) \wedge (y = 1) \wedge (z = 0)$

$\rho_4 \triangleq (x' > 0) \wedge (y' = 1) \wedge (z' = 3)$

$post_F(\rho_4) \triangleq (x' = x) \wedge (y' = y) \wedge (z' = z)$

$\wedge (x > 0) \wedge (y = 1) \wedge (z = 3)$

$\rho_5 \triangleq (x' > 0) \wedge (y' = 1) \wedge (z' = 3)$

Thus, we have concluded that the final value for the variable z is 3 and thence, we have concluded that $2 \not| z$. Since we had assumed that $x > 0$ then the implication $(x > 0) \rightarrow (2 \not| z)$ holds. Furthermore, given that we first assumed that δ and γ held then $(\delta \wedge \gamma) \rightarrow [(x > 0) \rightarrow (2 \not| z)]$. ■

4.5.5 Second Claim : $\alpha \wedge \beta \wedge \mathtt{swap}_{4,6}^{\mathbf{P_0}} \boxdot\!\!\rightarrow [(\mathbf{x > 0}) \rightarrow (\mathbf{2} \not| \mathbf{z})]$

Using the *first claim* and the *Conditional Deduction Rule*, then we can assert that:

$$\frac{(\alpha \wedge \beta \wedge \mathtt{swap}_{4,6}^{P_0}) \boxdot\!\!\rightarrow \gamma \wedge \delta}{\alpha \wedge \beta \wedge \mathtt{swap}_{4,6}^{P_0} \boxdot\!\!\rightarrow [(x > 0) \rightarrow (2 \not| z)]} \quad . \tag{4.6}$$

Notice that the premise of this instantiated rule already holds since it is a valid assertion in the context of our model. Once again, by the completeness of the *counterfactual logic* we can use *Modus Ponens* and infer the required assertion, i.e. $\alpha \wedge \beta \wedge \mathtt{swap}_{4,6}^{P_0} \boxdot\!\!\rightarrow [(x > 0) \rightarrow (2 \not| z)]$ and this proves the second claim. ■

In order to prove this claim, we need to employ the last rule of the fragment of counterfactual logic we proposed. We will use the *partial transitivity theorem* which we re-state below:

$$\frac{\vdash (\phi \rightarrow \chi) \wedge (\chi \boxdot\!\!\rightarrow \psi)}{\vdash \phi \boxdot\!\!\rightarrow \psi} \quad . \tag{4.7}$$

Notice that the *first claim* is what we first asserted about our intuition regarding the transformation from P_0 to P_1. Based on our model $\mathcal{R}$ and the *second claim*, we can assert the following:

1. $(P_0 \wedge \mathtt{swap}_{4,6}^{P_0}) \rightarrow (\alpha \wedge \beta \wedge \mathtt{swap}_{4,6}^{P_0})$
2. $(\alpha \wedge \beta \wedge \mathtt{swap}_{4,6}^{P_0}) \boxdot\!\!\rightarrow [(x > 0) \rightarrow (2 \not| z)]$

Thus, if we instantiate the variables in the latter rule like we show below:

$$\begin{aligned} \phi &\triangleq P_0 \wedge \mathtt{swap}_{4,6}^{P_0} \\ \chi &\triangleq \alpha \wedge \beta \wedge \mathtt{swap}_{4,6}^{P_0} \\ \psi &\triangleq (x > 0) \rightarrow (2 \not| z) \end{aligned}$$

Therefore, we can see that, by the first two assertions and the *partial transitivity theorem* stated above, then the assertion $[P_0 \wedge \mathtt{swap}_{4,6}^{P_0}] \boxdot\!\!\rightarrow [(x > 0) \rightarrow (2 \not| z)]$ holds and this proves the theorem. ■

```
1    a : Integer := 30;
2    b : Integer := 9 - a/5;
3    c,d : Integer;
4  begin
5    c:= b * 4;
6    if c > 10 then
7        c := c - 10;
8    d := c * (60/a)
9  end
```

Figure 4.3: Source code for the second example

4.6 Second Example

In the second example, we want to prove that if a *constant propagation transformation* were to be applied to a program then its semantics would have remained unchanged. Below, we show the source code for the initial version of our program.

In order to apply a *constant propagation* transformation to the source code in Figure 4.3, we define the following transformation steps:

1. Do-nothing transformation on statement 1. $\therefore P_1 = P_0[s_1/s_1]$
2. In statement 2, replace the RHS of the assignment with the constant value 3. $\therefore P_2 = P_1[s_1'/s_1]$
3. Do-nothing transformation on line C. $\therefore P_3 = P_2[s_3/s_3]$
4. Replace RHS of the assignment for constant value 12 on line D. $\therefore P_4 = P_3[s_4'/s_4]$
5. Replace current boolean expression with the constant $True$ in line E. $\therefore P_5 = P_4[s_5'/s_5]$
6. Replace RHS of the assignment for constant value 2 on line F. $\therefore P_6 = P_5[s_6'/s_6]$
7. Replace RHS of the assignment for constant value 4 on line G. $\therefore P_7 = P_6[s_7'/s_7]$

Via the latter set of statement replacements we can surely assert that the pair (P_0, P_7) is contained in the transitive closure of $\Rightarrow$ and thence $P_0 \Rightarrow^+ P_7$. Furthermore, based on our first example, in this instance, we will also provide the propositions which denote the structure of each program version (i.e. P_0 and P_7). In the following set of statements each statement label denotes a singleton statement set. The corresponding statement replacement will be given by the corresponding primed Greek variable as it is shown below.

$$\alpha \triangleq f(2) = \texttt{b} := \texttt{9} - \texttt{a}/\texttt{5} \Rrightarrow \alpha' \triangleq f'(2) = \texttt{b} := \texttt{6}$$
$$\beta \triangleq f(5) = \texttt{c} := \texttt{b} * \texttt{4} \Rrightarrow \beta' \triangleq f'(5) = \texttt{c} := \texttt{12}$$
$$\delta \triangleq f(6) = \text{ if } c > 10 \text{ then } \Rrightarrow \delta' \triangleq f'(6) = \text{ if } True \text{ then}$$
$$\gamma \triangleq f(7) = \texttt{c} := \texttt{c} - \texttt{10} \Rrightarrow \gamma' \triangleq f'(7) = \texttt{c} := \texttt{2}$$
$$\epsilon \triangleq f(8) = \texttt{d} := \texttt{c} * (\texttt{60}/\texttt{a}) \Rrightarrow \epsilon' \triangleq f'(8) = \texttt{d} := \texttt{4}$$

Furthermore, it follows that $P_0 \triangleq \alpha \wedge \beta \wedge \gamma \wedge \delta \wedge \epsilon$ and similarly, $P_7 \triangleq \alpha' \wedge \beta' \wedge \gamma' \wedge \delta' \wedge \epsilon'$. Moreover, let us define the following predicate which will denote the transformation applied to P_0 by $\texttt{cfold}(P_0) \triangleq (P_0 \Rightarrow^+ P_7)$ and therefore, $\texttt{cfold}(P_0) \triangleq \alpha' \wedge \beta' \wedge \gamma' \wedge \delta' \wedge \epsilon'$. Thence, in this example. we are required to prove that:

$$P_0 \wedge \texttt{cfold}(P_0) \boxdot\!\!\rightarrow (d = 4) \ . \tag{4.8}$$

4.6.1 Proof

Since the transformations were done via our already defined *versioning model*, then we can assert that

$$\mathcal{R} \models (\alpha \wedge \ldots \wedge \epsilon \wedge \texttt{cfold}(P_0)) \boxdot\!\!\rightarrow (\alpha' \wedge \ldots \wedge \epsilon') \ . \tag{4.9}$$

```
        var:
1           x : Integer := 1;
2           y : Integer := 2;
3           z : Integer := 3;
4           i : Integer := 0;
5           w : Integer := 0;
6           skip;
7       begin
8            for i := 1 to 100 do begin
9                w := i;
10               z := x + y
11           end
12      end
```

Figure 4.4: Source code for the *loop hoisting* example

By a simple post-condition analysis (which we will not include here given its trivial nature) we can also assert that $(\alpha' \wedge \beta' \wedge \gamma' \wedge \delta' \wedge \epsilon') \rightarrow (d = 4)$ and thence, by the *conditional deduction axiom* we can assert that:

$$[\alpha \wedge \ldots \wedge \epsilon \wedge \texttt{cfold}(P_0)] \; \square\!\!\rightarrow (d = 4) \;\; . \tag{4.10}$$

Since $P_0 \triangleq \alpha \wedge \beta \wedge \gamma \wedge \delta \wedge \epsilon$, we can directly conclude that $[P_0 \wedge \texttt{cfold}(()P_0)] \; \square\!\!\rightarrow (d = 4)$ which is exactly what we had claimed. ■

4.7 Third Example

In this example we will show that after a *code hoisting* modification on the provided source code still behaves the same. Below we provide the example source code which consists of a simple loop whose original loop body consists of two assignments. The first statement is surely dependent on the loop variable (i.e. i) and thus it will not be affected by code hoisting. The second statement is composed of a LHS and a RHS which does not depend on the loop variable and hence, the whole statement may be extracted from the loop body.

Let us denote the code in Figure 4.4 as P_0. Notice that, in Figure 4.4 we have only labeled two lines given that the *loop hoisting* program transformation will consist of a swap of these two lines. Furthermore let us declare the following predicates which will let us integrate the programs' structure into our proof:

- $\alpha \triangleq f(6) = \texttt{skip}$
- $\beta \triangleq f(10) = \texttt{z} := \texttt{x} + \texttt{y}$
- $\delta \triangleq f'(10) = \texttt{skip}$
- $\gamma \triangleq f'(6) = \texttt{z} := \texttt{x} + \texttt{y}$

Thus we can declare an alternative version in which statements s_A and s_B will be swapped. Thence, $P_1 \triangleq P_0[s_6/s_{10}][s_{10}/s_6]$. The latter implies that in P_1 the statement $\texttt{z} := \texttt{x} + \texttt{y}$ is removed from the loop body. Moreover, in the same manner as we did in section 4.6, we claim that a simple static analysis similar to the one done in section 4.5 will let us assert that:

$$P_0 \rightarrow (w = 5050) \ . \tag{4.11}$$

At this point we can use the proof theoretical fragment that pertains to our approach. More specifically, we can invoke the *conditional deduction rule* which will assert below for reference purposes,

$$\frac{\vdash \lambda \rightarrow \psi}{\vdash (\phi \ \square\!\!\rightarrow \lambda) \rightarrow (\phi \ \square\!\!\rightarrow \psi)}$$

For this purpose we will use the following instantiation:

$\lambda \triangleq P_0$

$\phi \triangleq P_0 \wedge \texttt{swap}_{6,10}^{P_0}$

$\psi \triangleq w = 5050$

We can see that by 4.11 the needed hypothesis regarding the *conditional deduction rule* already holds. Hence, by *Modus Ponens* we can assert that:

$$[(P_0 \wedge \mathtt{swap}_{6,10}^{P_0}) \boxdot\!\!\rightarrow P_0] \rightarrow [(P_0 \wedge \mathtt{swap}_{6,10}^{P_0}) \boxdot\!\!\rightarrow (w = 5050)] \ . \tag{4.12}$$

If we look at the antecedent for the implication in 4.12, we can readily verify that according to the model theory (semantics) exposed in [29] we have that for any well-formed expressions σ, τ, $(\sigma \wedge \tau) \boxdot\!\!\rightarrow \tau$ and $(\sigma \wedge \tau) \boxdot\!\!\rightarrow \sigma$ are valid assertions. Thence, by the completeness of counterfactual logic, it follows that these are axioms in the proof-theoretical sense. Thus, by *Modus Ponens*, in our case it readily follows that $(P_0 \wedge \mathtt{swap}_{6,10}^{P_0}) \boxdot\!\!\rightarrow (w = 5050)$ which is want we wanted to prove. ■

4.8 Conclusion

We have introduced a logical calculus based on Lewis' theory of counterfactuals. Additionally we have shown that if we know how to unambiguously characterize the transformation from the initial version to the future desired version then, the conjunction between the structural properties of the initial version and the predicates that characterize the transformation, imply the desired future version's properties. The proof we presented could be easily expressed in terms of a *natural deduction system* and be automated.

If we think about the implications regarding the capabilities of our counterfactual calculus, it would not be hard task to extend it up to the point where we can also verify refactoring-based development without actually having to produce the new versions and subjecting them to the usual regression test-based debugging process.

Chapter 5

Counterfactually Reasoning About Security

5.1 Introduction

In this chapter, we provide the background to counterfactual logic and give very general suggestions on how we could employ this logic to help us reason about security policies. It seems very appropriate to use this kind of logic to anticipate a change that will compromise the security concerns of a given system before actually applying the changes[1]

.

In the realm of software security, changes regarding security policies are pervasive. It is in this constant changing environment where a system's security becomes compromised. In practice, security policies are changed and, in the worst case, any defect or undesired effect is usually found after the fact and often too late. Requiring that the security policies remain unchanged is out of the question and blatantly unrealistic. Therefore, we are in need of mechanisms that enable us to formalize the security policies, the changes regarding security policies and the future effect of said changes.

In this chapter we present a framework for *what-if* analysis of security policies based on Lewis' theory of counterfactuals [29]. The framework can be used to statically perform change-impact analysis for access control matrices. It enables us to verify assertions about a changed version of an access control matrix without actually incorporating the changes. We present a logical calculus that precisely characterizes potential structural modifications to source code and their impact on the program's behavior.

[1]Portions of this chapter were published in:

Manuel Peralta, Supratik Mukhopadhyay, and Ramesh Bharadwaj. Counterfactually reasoning about security. In *Proceedings of the 4th international conference on Security of information and networks*, SIN '11, pages 223–226, New York, NY, USA, 2011. ACM.

5.2 Counterfactual Theory

The logic of counterfactuals helps us reason about assertions that are not a matter of fact. In [29] Lewis provided a sound and complete proof system and proved its decidability. Based on the logic of counterfactuals we derive a logical calculus that allows us to assert properties that would hold for a future version of a given program and verify that these would indeed hold if the changes needed to obtain that version were actually implemented.

5.2.1 The Language of Counterfactual Theory

In coherence with [29] we will briefly introduce the language regarding the logic of counterfactuals below (p_i denotes a propositional variable):

$$\phi ::= p_i | \neg\phi | \phi \wedge \phi | \phi \vee \phi | \phi \rightarrow \phi | \phi \boxdot\!\!\rightarrow \phi$$

The counterfactual sentence $\phi \boxdot\!\!\rightarrow \psi$ should be read as: **if it had been the case that ϕ, it would have been the case that** ψ. Thus, if we had an assertion whose antecedent ranged over the properties of some given access control matrix and also the changes needed to produce a new version and whose consequent ranged over the properties that a new version would have, then we could use counterfactual logic to code such an assertion.

5.2.2 Formal Representation of A Security Model

In this section we will define a simple variation of the Access Control Matrix (ACM) model. This model was first introduced in [28] and [16]. We have chosen this model due to its simplicity and readily intuitive nature and widespread use as it is stated in [28]. First of all, we will define three sets: S, O and A which are respectively the set of : subjects, objects and actions. The set of subjects contains the active entities on the system (i.e. users, computer systems, etc.); the set of objects denotes the set of entities over which subject are allowed or denied a certain action. The

set of actions denotes those tasks which a subject can perform on a given object. Hence:

$$\begin{array}{lcll} S & = & \{s_i\}_{i \in I} & \text{The set of subjects} \\ O & = & \{o_j\}_{j \in J} & \text{The set of objects} \\ A & = & \{Read, Write\} & \text{The set of actions} \end{array}$$

Thus, we can now formally define an *access control matrix* as a function $M : S \times O \rightarrow 2^A$ which takes an ordered pair composed of a subject and an object and assigns to them a subset of the possible set of actions. Moreover, in this instance we will have to work with M's *intentional* or set representation and thence:

$$M \triangleq \{(s_i, o_j, \alpha_k)\} \text{ where } \alpha_k \in 2^A$$

Encoding Change in the ACM model

Let $M = \{(s_i, o_j, \alpha_k)_{i,j,k \in \mathbb{N}}\}$ be the current version of the access control matrix. We can encode the state of any ACM by using the following formula:

$$\Psi_M \triangleq \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} \bigwedge_{k=1}^{p} (s_i, o_j, \alpha_k) \in M$$

We will simplify the latter expression using the following notation:

$$\Psi_M \triangleq \bigwedge_{i,j,k} (s_i, o_j, \alpha_k)$$

For the sake of simplicity we will define a change to the ACM as a change to the members of the action-set of a given triple. Hence, a change may be represented as:

$$(s_i, o_j, \alpha_k) \Rightarrow^c (s_i, o_j, \alpha'_k)$$

It can be readily inferred that $\Rightarrow^c$ is a three-place-relation over $S \times O \times A$. Furthermore, let $\mathcal{M}$ denote the class of all possible ACM versions. Therefore, $\Rightarrow^c$ can be thought of as a binary relation over $\mathcal{M}$ and

$$M \Rightarrow^c M' \text{ iff } M' \triangleq M[\alpha_k/\alpha'_k]$$

Moreover we define $\Rightarrow^c$ to be the smallest relation such that the following holds:

$$(s_i, o_j, \alpha_k) \Rightarrow^c (s_i, o_j, \alpha'_k) \text{ iff:}$$

1. $\alpha'_k \neq \emptyset$ when $\alpha_k \neq \emptyset$
2. $\alpha'_k \neq A$ when $\alpha_k = A$

Undesirable Configurations

In any system, there is a set of undesirable states. These states may be very possibly members of the entire set of possible states. One of the fundamental purposes of any security mechanisms is to guarantee that for any possible transition (that originates in a safe/legal state) the target state will not be an illegal/undesirable state. In this instance an *undesired state* will be denoted by a given configuration/triple of subject, object and action. Therefore, let $U \subseteq S \times O \times A$ be the set of illegal configurations and let τ_u range over this set.

We want to avoid allowing a configuration change in which we enable an undesirable triple be part of the new version of the ACM. Hence, we want to avoid the following:

$$M \Rightarrow^c M' \text{ where } \tau_u \in M'$$

Secure Counterfactual Change

Our objective is to enable *counterfactual logic* to let us decide whether or not a change to the current version of the ACM implies that at least one illegal triple is part of the future resulting version. Thence our secure counterfactual implication can be expressed as:

$$[\bigwedge_{i,j,k\in\mathbb{N}}(s_i, o_j, \alpha_k)] \wedge (s_0, o_0, \alpha_0)[\alpha_0/\alpha'_0] \boxdot\!\!\rightarrow (\tau_u \notin M')$$

5.2.3 Kripke Versioning Model

In [29] the author provides the semantics of his counterfactual propositional logic using a multiple-world interpretation. In that same manner we have chosen to interpret our access control matrix transformation. In our case, each ACM version will represent a world. In the following definition, we take the liberty of writing $t_i \leftarrow t_j$ to denote that the tuple t_i was swapped by tuple t_j. Below, we provide a formal interpretation based on a Kripke model.

Definition 5.2.1 (Kripke Version Model) *A Kripke Version Model $\mathcal{R}$ is a triple $\langle \mathcal{M}, \Rightarrow, M_0 \rangle$ where:*

1. *$\mathcal{M} = \{M_k\}_{k\in\mathbb{N}}$ is the set of all access control matrix versions (ACM states).*
2. *M_0 is the initial access control matrix.*
3. *$\Rightarrow \subseteq \mathcal{M} \times \mathcal{M}$ is a binary relation defined the set of all possible ACM versions. Where $\Rightarrow$ is the smallest relation such that the following properties hold:*
   (a) *$t_i \leftarrow t_i$: tuple s_i is left unchanged. This stands for the do nothing transformation.*
   (b) *$t_i \leftarrow t_j$: tuple t_j replaces statement t_i, where $t_j \in M_k$. We usually call this primitive transformation, a swap.*
   (c) *$t_i \leftarrow t_j$: statement t_j replaces statement t_i, where $t_j \notin M_k$. Thus, s_k is a new statement.*

(d) $(\forall i)t_i \in M_k$ *can be changed only once.*

Furthermore, we assume that the relation $\Rightarrow$ complies with the properties of *reflexivity*, *symmetry*, and *transitivity*. Below, we justify each property based on the latter definition of $\Rightarrow$:

1. **Reflexivity:** For any ACM $M_i \in \mathcal{M}$, it is obvious that the *do-nothing transformation* will yield that any ACM can be transformed into itself. Therefore, $M_i \Rightarrow M_i$ given that for all $s_j \in M_i$, $M_i = M_i[s_j/s_j]$

2. **Symmetry:** For any ACMs $M_i, M_j \in \mathcal{M}$ any of the above transformations can be reversed and therefore, $M_i \Rightarrow M_j$ implies $M_j \Rightarrow M_i$.

3. **Transitivity:** For any ACMs $M_i, M_j, M_k \in \mathcal{M}$, applying two or more transformations to a program will yield intermediate versions; this is equivalent to transforming the initial version by composing the transformations into one. Thus, $M_i \Rightarrow M_j$ and $M_j \Rightarrow M_k$ imply that $M_i \Rightarrow^+ M_k$. Where $\Rightarrow^+$ denotes $\Rightarrow \circ \Rightarrow^{n-1}$ and $n > 1$.

5.2.4 Interpreting the Counterfactual Implication

As it was stated earlier, the purpose of our model is to help interpret assertions in the language of counterfactual logic. Let M_0 denote our given initial ACM version. Also, let us assume we had a counterfactual assertion, namely $\phi \boxdot\!\!\rightarrow \psi$ in which:

- ϕ stands for assertions regarding M_0 and some transformation $t_i \leftarrow t_j$ that implies that $M_1 = M_0[t_i/t_j]$

- ψ stands for assertions regarding M_1

Thus, following the model-theoretic interpretation proposed by Lewis in [29], our version of the counterfactual implication is interpreted as:

$$\mathcal{R} \models \phi \boxdot\!\!\rightarrow \psi \ . \tag{5.1}$$

Where $\mathcal{R}$ denotes our previously defined *Kripke Versioning Model*. Moreover, letting α_i, β_i denote propositional statements about the structure of M_i and M_j respectively, then, we can state that:

$$\begin{aligned} \phi &\triangleq (\bigwedge_{i=1}^{n} \alpha_i) \wedge (M_i \Rightarrow^{+} M_j) \\ \psi &\triangleq \bigwedge_{j=1}^{m} \beta_j \end{aligned}$$

Where $\Rightarrow^{+}$ denotes the positive/transtive closure for the relation $\Rightarrow$. Furthermore, given an initial ACM version, namely M_0, we produce several versions by applying one or more transformations to it. In the context of a counterfactual assertion, the current version's structure and the changes applied to it (in order to produce a new version) imply properties possessed the new version and hence:

$$\begin{aligned} \mathcal{R} \models \phi \boxdot\!\!\rightarrow \psi \ \triangleq \ & (\exists_{\min} k \in \mathbb{N})(\textstyle\bigwedge_{i=1}^{n} \alpha_i) \\ & \wedge (M_0 \Rightarrow^{k} M') \rightarrow (\textstyle\bigwedge_{j=1}^{m} \beta_j) \ . \end{aligned}$$

The latter should be interpreted as there exists a minimal number of transformation steps such that given the properties of our initial ACM M_0 (namely, $\bigwedge_{i=1}^{n} \alpha_i$) and the transformation between the two program versions implies the desired properties of the future ACM version (namely, $\bigwedge_{j=1}^{m} \beta_j$).

5.3 Applications of Counterfactual Theory to Security

Each change to the access control matrix modifies the state of the security system. Hence, each change reflects a change in the set of valid policies. It seems very promising to use counterfactual logic to 1) encode the changes to the ACM, 2) express the undesirable state-tuples, and 3) assert whether or not the changes *counterfactually* imply the undesirable tuples are part of the future state

of the ACM. Given all the risks involved in changing security policies, it would be nice to foresee their effect before incorporating them into production systems.

5.4 Conclusion

We have introduced a logical calculus based on Lewis' theory of counterfactuals. Additionally we have shown that if we know how to unambiguously characterize the transformation from the initial ACM state to the future desired ACM state, the conjunction between the structural properties of the initial ACM version and the predicates that characterize the transformation, imply the desired future ACM state's properties.

This chapter has presented a powerful and promising suggestion which consists of jointly using a perhaps modified version of the ACM model and our counterfactual logical calculus. The latter mix would enable practitioners verify *a-priory* the effects of a change to the ACM without actually applying the change to production systems. Although it is widely known that the question of whether or not a given security model enforces a given policy is a non-decidable problem, we are confident that our simplified ACM model and our counterfactual logic will be helpful to enough non-trivial applications.

Chapter 6

Reasoning About Sensor Networks

6.1 Introduction

In this chapter, we study an approach for dynamically reconfiguring sensor networks that operate under dynamically changing environments. We show how dynamic reconfiguration can be achieved using our Secure Operations Language for JAVA - based approach. We show how to specify a reconfiguration using a counterfactual logic and provide techniques that enable us to understand the impacts of change[1].

Sensor networks are embedded networked systems that receive percept streams from the environment and constantly react to them. From a software-based point of view, modifications done to any system (SNS) should be performed under utmost caution as SNSs are often deployed in *mission-critical applications*. The latter means that any disruption that inhibits the system in satisfying its operational semantics will definitively yield catastrophic results. Also, sensor networks are required to react dynamically to ever-changing environmental conditions without human intervention. Therefore, one should strive to provide methods that ensure the correctness of SNSs under reconfiguration is preserved i.e. the SNS respects it operational requirements while the system structure changes in response to an evolving environment.

For SNS applications, requirements dynamically change in a rapid, unpredictable, and continuous fashion. In applications such as those driving search and rescue missions, any operational expansion/contraction requires dynamic reorganization of the system. For these application scenarios, any downtime resulting from upgrade of the control system leads to unacceptable disruption

[1] Portions of this chapter were published in:

Sitharama Iyengar and Richard Brooks, editors. *Distributed Sensor Networks, Second Edition: Image and Sensor Signal Processing*, volume 2 of *Computer & Information Science Series*, chapter 31, pages 693–710. Chapman and Hall/CRC, 2 edition, September 2012.

of service. As a result, continued availability of such systems in a mission-critical setting, even under dynamically changing requirements, is of utmost importance. We need to develop techniques, tools, and methods that can build, manage, and maintain SNS systems whose requirements keep changing perpetually during their lifecycle. Such systems should be able to autonomously re-engineer themselves rapidly online under changing requirements, with minimal or no disruption in service, and yet meeting all constraints of timeliness, cost, and performance in a reasonable way.

Traditional software development methodologies assume that requirements are well understood and available, in the form of a formal or rigorous specification, of the required system behavior. However, this assumption fails to hold for software that is meant to control SNS applications deployed in rapidly evolving scenarios. While it is possible to develop logically precise requirements for software computing mathematical functions, the behavior of a software system depends on extraneous factors that are not usually foreseen during its development. These include factors such as platform of deployment (e.g., the word length of the machine on which the software is run), the communication protocols used, the amount of memory available, etc. In software solving real world problems, such extraneous factors are compounded by those from the system's physical environment that expect the software to cope with dynamically changing business constraints. In some cases, given time and money, it is possible to get the original developers to update the software to meet the changed business requirements. However, in certain cases, such an update may not be possible even with adequate time and money (e.g., the original developers may have moved away from the technology and it may not be possible to acquire a suitable team to build on their work). In these situations, the system must be phased out with millions of dollars in software development cost wasted.

Conventional approaches used in industry are inadequate in rapidly evolving mission-critical scenarios, due to (1) the unpredictable nature of the evolution of the system requirements, (2) dynamically changing situational environments driven by the dynamics of the market/mission-partners, including rapid mergers, disinvestment, formation of coalitions, noisy and unpredictable

communication channels, and cyber-attacks aimed at disruption of the network. We use the phrase "Perpetual Requirements Engineering" to denote an approach where we address dynamically changing requirements throughout the software lifecycle, including autonomous rapid reconfiguration and persistent redeployment of distributed software systems without disruption to service commitments, in an expeditious manner. Our approach improves on the agile development paradigm. Traditionally, agile development has been used successfully for software projects with rapidly changing requirements. One example of the agile development approach is extreme programming. In agile development, activities generally alternate between modeling and coding, with major portions of the design being generated as implementation proceeds. Traditional agile development approaches, however, suffer from a lack of automated support. Manual effort is needed to incorporate any changes in the requirements into a software artifact. Extensive manual refactoring of code is often needed to ameliorate the effects of dynamically changing code requirements. System updates are developed manually, at a huge cost and in an untimely and unpredictable manner. This is one inhibiting factor on the scalability of agile development methods, thereby making them suitable only for development projects of a small or moderate size. This is also the reason why traditional agile development methods tend to succeed only when an experienced development team is available. Lack of dynamic adaptation means frequent shipping of new code in response to ever-changing requirements. This not only adds to the cost but also results in increased downtime of deployed software due to frequent updates to the running code. While the involvement of the customer is an essential component of the agile development method, in many situations, it becomes difficult for the development team to stay in touch with the customer, especially in the post-deployment stage (e.g., consider projects whose development is outsourced).

In large software projects with dynamically changing requirements as usually encountered in the development of SNS applications, instead of manually performing development iterations every time a requirement changes, we need to implement techniques and strategies for automatic incremental update of system or subsystem components comprising the deployed software, in response

to changing requirements. We need a requirements engineering paradigm that can automatically reflect incremental changes in the requirements by a dynamic reconfiguration and persistent update of the running software.

6.2 Perpetual Requirements Engineering for Sensor Networks

In the context of perpetual requirements in SNSs we have developed an approach that relies on two technological elements. The first is Secure Operations Language for Java (SOLj) [6] which is an event-based domain-oriented synchronous programming extension of Java used to write the specification for agents which, in this context, are the software counterparts of one or more sensors. The second element is the Secure Infrastructure for Networked Systems (SINS). SINS a is a distributed run-time system in which the SOLj agents are deployed. The SINS run-time and framework provide a set of security policies which help avoid compromising the SOLj agents behavior and interactions.

6.2.1 SOLj - Secure Operations Language-JAVA

SOLj: Secure Operations Language-Java. Given that SOLj is an extension of Java, it is presented as modular extensions to its core language, i.e., Java. A *module* comprises the specification unit in SOLj; it is composed of type definitions, variable declarations, service declarations, assumptions and guarantees and definitions. In the future we will use the word *agent* to denote a module instance. A SOLj module may include *attributes* as they are described below:

deterministic Declares a module that does not exhibit nondeterministic behavior. The compliance with this attribute is checked by the SOLj compiler.

reactive Declares a module that will cause a state change only when its (visible) environment fires an event via a state change or by invoking a method. Also, this attribute denotes that the module's response to an event will happen in the next immediate step.

The type definition section contains user-defined types as well as enumerated types. The Java comment `//@type definitions` precedes the "type definitions" section. It provides SOLj directive to the compiler indicating the start of the type definition section. The variable declaration section defines three types of variable which are explained below:

monitored variables variables in the environment that influence the agent's behavior.

controlled variables variables in the environment that are changed by the agent's behavior.

internal variables this variables reflect the agent's internal state.

The Java comment `//@Services` precedes the "service declarations" section which declares the agent's external interface. It contains the methods that realize the services the module provides. For each method declaration within a service, the SOLj language provides the capability of declaring the corresponding preconditions and postconditions which denote the conditions under which each service should start and terminate. The preconditions and postconditions are encoded as arithmetic expressions and type declarations. A type declaration is denoted by a type judgement expression `T:x` where `x` is a variable and `T` is a type. These constraints are enforced at runtime in a dynamic manner. Also, each service invocation must use a "continuation variable". This variable includes a boolean field called "done" which is assigned a value of `true` once the service invocation is ready to provide a return value.

The `//@Assumptions` comment denotes the start of the *assumptions* section which includes assumptions that determine the agent's correct way to operate. If any of these assumptions is violated by the environment, the agent's execution is aborted. The `guarantees` section contains the agent's required safety properties. The `definitions` section provides *update functions* which denote variable definitions. These specify the corresponding values for internal and controlled variables. For the sake of future references, we will distinguish between *monitored variables* whose values are given by the environment, and *dependent variables* which are those whose values that

are the result of SOLj agents computations. These values are obtained using the values of monitored variables and (possibly) past dependent variable's values.

6.2.2 SOLj Events

Based on the Software Cost Reduction (SCR) method's SCR Abstract Language (SAL), SOLj has been provided with the capability of defining events [23]. Intuitively, SCR events can be interpreted as state changes. Moreover, the occurrence of an event is triggered when a variable has its value changed. This is done by update functions which change the dependent variable's value.

$$\begin{aligned}
@\mathtt{T}(\mathtt{c}) &=^{def} \neg @\mathtt{PREV}(\mathtt{c}) \wedge \mathtt{c} \\
@\mathtt{F}(\mathtt{c}) &=^{def} @\mathtt{PREV}(\mathtt{c}) \wedge \neg \mathtt{c} \\
@\mathtt{C}(\mathtt{c}) &=^{def} @\mathtt{PREV}(\mathtt{c}) \neq \mathtt{c}
\end{aligned}$$

An initialization method (`init`) assigns starting values to all dependent variables (controlled or internal). Each module contains an `init` method. Each dependent variable is updated by just one of the update functions. Moreover, a *dependency relation* is induced by the interplay between update functions and dependent variables. We denote this relation with D_m. Let `a` and `b` be two dependent variables, then we say that $(\mathtt{a}, \mathtt{b}) \in D_m$ if and only if `a` is updated by the function corresponding to `b`. The fact that `a` may depend on the previous values of other variables and itself does not influence the dependency relation. Also, a *dependency graph* may be derived from D_m by interpreting the set of dependent variables as the nodes and each $(\mathtt{a}, \mathtt{b}) \in D_m$ as the edges [2]. For each module, we need to consider its corresponding dependency graph as acyclic.

From a simplified point of view, a SOLJ program executes in a sequence of *steps* and each step is preceded by the triggering of an event. The module's dependency relation induces the order in which the variable updates and service invocations are carried out. Each computation step may be decomposed as follows:

[2]The notion of a dependency relation is easily extended to the entire system.

1. The environment or the agent itself triggers an event in a nondeterministic manner

2. Each agent *responds* to this event by modifying the values of its dependent variables

From an external point of view, the updates and service invocations can be thought of happening in a synchronous manner (as it is dictated by the *Synchrony Hypothesis* and exemplified by languages such as Esterel and LUSTRE [21]. The latter implies that all dependent variable updates and service invocations that occur as a response to an event triggering, happen before the next event is triggered.

6.2.3 SOLj Definitions

The main part of a SOLj module is the `definitions` section. This section declares and defines the update function that corresponds to each dependent variable. Update functions provide the value of the updated dependent variable. An update function's body is comprised of *return* statements which are constrained by *conditional expressions*. These are activated by events triggers initiated by the environment and/or the agent. Syntactically, these conditional expressions are denoted by Java conditional expressions with the difference that the guards are *SOLj events*.

SOLj expressions can be service invocations such as `A:B(varList)^cont`. The identifier `A` denotes the name/URL of the service; `B` denotes the name of the invoked method; `varList` denotes the set of formal parameters that are provided to the invoked method and `cont` denotes the unique continuation variable associated with the invocation. When the "done" field in the continuation is assigned "true" the service invocation has terminated. Finally, a compiler derives Java code from the SOLj definitions. This derived code executes on the Secure Infrastructure for Networked Systems (SINS).

6.2.4 SINS

SOLj module instances interact in a runtime environment called Secure Infrastructure for Networked Systems (SINS). Generally, a SINS implementation consists of a set (one or more) *SINS Virtual Machines* (SVM). Each of these virtual machines acts as a container for one or more agents in a given host node. Distributed SVMs communicate amongst themselves using the *Agent Control Protocol* (ACP) [52] with the purpose of exchanging agent and control information. A supplementary protocol, known as the Module Transfer Protocol (MTP) takes care of the code distribution, digital signatures, authentication, and code integrity. Complying with locally enforced security policies, SOLj agents are allowed to access local resources in a host. Compliance with these security policies is verified using an inductive theorem prover. Observer agents (termed "security agents") are in charge of enforcing other safety property and security requirements. These agents monitor the execution of application-specific agents and also engage in corrective actions once a violation is detected.

6.3 A SOLJ Example - Auto-regulated Power Generation Network

The following example will be employed as a didactic tool in order to illustrate how our SOLj-SINS-based approach looks like in a practical setting.

6.3.1 General Aspects

In the realm of power generation and considering a very simple perspective, we have the interplay of three interrelated variables: voltage, current and resistance. In large scale power distribution networks such interplay of these variables can mean the difference between efficient or wasteful power distribution. It would be desirable, thus, to have a sensor-network-based system, which would adequately react to changes in this quantities in a dynamic manner.

6.3.2 System Description

Our simplified sensor network for power generation system is comprised mainly by two types of agents:

Distribution Line Agent (*DistLineAgent*) This agent type is in charge of communicating the state of the distribution lines to the `GenAgents` i.e. it reports the conditions regarding voltage and resistance. Figure 7.2 shows the SOLj source code for this type of agent.

Generation-Engine Agent (*GenAgent*) This agent type is responsible of regulating the electrical current that is input into the electrical distribution system based on the voltage and resistance on distribution lines. Figure 7.3 shows the SOLj source code for this type of agent.

6.3.3 System Operation

Let us assume that we are dealing with an electrical distribution network comprised of several `GenAgents` and multiple `DistLineAgents`. Furthermore, let us assume that the environment temperature variations (the material's own reaction to conducting current and environmental heat) dilate and contract the inner core of the conducting cables. Therefore, the conducting cables' resistance is also variable. From general Physics we know that these three quantities are interrelated by the following formula:

$$I = \frac{V}{R}$$

Where I stands for the electrical current in the system (measured in Amperes), V stands for the system's voltage (measured in Volts) and R stands for the conducting cables' resistance (measured in Ohms).

The `DistLineAgents` via sensors placed along the distribution lines, have the knowledge of the actual current and resistance along the distribution line system and they communicate this

information back to the set of `GenAgents` which in turn regulate the electrical generators that provide the input electrical current to the distribution network. When, for instance, the environment exhibits a high temperature (due to heat wave perhaps), the resistance in the distribution lines will increase. Provided that the voltage remains the same, the correct reaction of the system would be to increase power production. The `GenAgents` monitor the resistance and can thus react to this change in the distribution system by actuating the generation engines to produce more power.

6.4 Counterfactuals in Sensor Networks

Counterfactual logic aids us in reasoning about statements that are not matter of fact. Lewis [29] provided a sound and complete inference system and also stated that this logic was a decidable one. We have created a logical calculus based on counterfactual logic. This calculus allows us to express properties that would take place in a future version of a given program and verify that indeed these properties would hold if the changes between the new and old version were applied.

6.4.1 The Language of Counterfactual Theory

As it is shown in [29] we will shortly define the language that comprises the logic of counterfactuals (p_i *denotes a propositional variable*)

$$\phi ::= p_i | \neg\phi | \phi \wedge \phi | \phi \vee \phi | \phi \rightarrow \phi | \phi \boxdot\!\!\rightarrow \phi$$

The counterfactual expression $\phi \boxdot\!\!\rightarrow \psi$ should be read as: **if it had been the case that ϕ, it would have been the case that ψ.** Therefore, if we had a statement that contained an anteceded that expressed the properties of a given program and the required changes to create the new version and the consequent expressed the properties that the new version would exhibit, then it seems plausible that we can encode such a statement using counterfactual logic.

```
//@Module Declaration
deterministic reactive module DistLineAgent

//@Type definitions
Voltage = floatStream;
Current = floatStream;
Resistance = floatStream;

//@Controlled variables
Voltage currVoltage;
Current currCurrent;
Resistance currRes;

//@Monitored variables
Voltage observedVoltage;
Current observedCurrent;
Resistance observedRes;

//@definitions
//@initialization
void init(){
    currVoltage = null;
    currCurrent = null;
    currRes     = null;
}

//@update functions
Resistance outputResistance(){
    if(@C(observedRes) && @T(observedRes >= thresholdRes))
        return currRes;
}

Voltage outputVolatge(){
    if(@C(observedVoltage) &&
    @T(observedVoltage >= thresholdVoltage))
        return currVoltage;
}

Current outputCurrent(){
    if(@C(observedCurrent) &&
    @T(observedCurrent >= thresholdCurrent))
        return currCurrent;
}
```

Figure 6.1: SOLJ code for the `DistLineAgent` module

```
//@Module Declaration
deterministic reactive module DistLineAgent

//@Type definitions
Voltage = floatStream;
Current = floatStream;
Resistance = floatStream;
Revolutions = floatStream;

//@Controlled variables
Revolutions rpm;

//@Monitored variables
Voltage currVoltage;
Current currCurrent;
Resistance currRes;
Revolutions currRevs;

//@Services
//@ Revolutions EGenerator:RevUpEngine(revDelta)
//@  pre  = (currCurrent >= thresholdCurrent) &&
//@         (currVolate  <= thresholdVoltage)
//@
//@  post = currRevs > rpm

//@ Revolutions EGenerator:RevDownEngine(revDelta)
//@  pre  = (currCurrent < thresholdCurrent) ||
//@         (currVoltage > thresholdVoltage)
//@
//@  post = currRevs < rpm


//@Update Functions
Revolutions UpdateRPMs(){
    if(@C(currVoltage) && @C(currCurrent))
        return EGenerator:RevUpEngine(revDelta)^revUpCont;
    else
        return EGenerator:RevDownEngine(revDelta)^revDownCont;
```

Figure 6.2: SOLj source code for `GenAgent`

6.4.2 Formal Description of SOLj Agent Network

Given what was presented in earlier sections, we can abstract a sensor network, by considering the set of SOLj agents that realize its behavior. Let M_i denote the syntactically correct code for a given agent, then, we can denote a SOLj agent network by the following parallel composition:

$$SN_0 \triangleq M_1||M_2||\ldots||M_n$$

A new version of a given sensor network will be created when at least the code of one agent is changed (as indicated by our *Program Transformer Approach*). Hence, a new version of a given by the following expression:

$$SN_1 \triangleq M_1||M_2||\ldots||M_i[s_j/s'_j]\ldots||M_n$$

Where M_i is the changed SOLj agent module and s_j and s'_j are the changed (swapped) statements within M_i's specification code. In general, any number of modules within the parallel composition may be changed by an arbitrary number of statement swaps. In terms of logical descriptions we can also denote the SOLj agent network by the following extended conjunction:

$$\Psi_{SN} \triangleq \bigwedge_{i=1}^{n} M_i$$

Where each M_i is denoted by:

$$\Psi_{M_i} \triangleq \bigwedge_{j=1}^{m} f(j) = Stat_j$$

Where $f(j)$ is the (uninterpreted) function symbol that denotes line in the SOLj agent's code and $Stat_j$ is a well- formed statement in the SOLj specification language. The two latter expressions yield that the logical expression corresponding to a SOLJ network will be:

$$\Psi_{SN} \triangleq \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} f_i(j) = Stat_j$$

The latter expression does not imply that all SOLJ agents have the same number(i.e. m) of statements. We can freely assume that m is just the number that denotes the longest SOLj module in the system.

6.4.3 Program Transformers

In [17] the authors define the notion of *predicate transformer*; it can be defined as a first order logic formula which yields (via *existential quantifier elimination*) the weakest precondition for a given command (a statement in a imperative language) and its corresponding post-condition. Inspired by the latter notion we have found a way to logically express program transformation using what we call *Program Transformers*.

Let Ψ_{SN} denote the conjunction of formulas shown in section 6.4.2. Furthermore, we choose to model possible program transformations using the following formula:

$$\begin{aligned} (\forall u \in U)(u \neq u_i) \wedge (u \neq u_j) \quad &\rightarrow \quad (f(u) = f'(u)) \\ &\wedge \quad (f(u_i) = f'(u_j)) \\ &\wedge \quad (f(u_j) = f'(u_i)) \end{aligned}$$

The latter expression denotes the existence of a *new* SOLj module f' that happens to differ from our original SOLj module f only by swapping two statements (i.e. u_i and u_j). In the rest of the section we shall use Ψ_S to refer to the implication above. With Ψ_{SN} defined as above, we can formally express our *program transformer expression* as:

$$\Psi_T \triangleq (\exists f)(\exists R)\Psi_{SN} \wedge \Psi_S$$

Notice that Ψ_T is a *second order logic* formula since we are quantifying over one function symbol and one relation symbol. Also, let us assume that we have a fixed SOLj module f_0. By definition, f_0 is a finite list of statements in $Stat$. Thence, we can logically express $f_0 \colon U \to Stat$ by a finite conjunction of equalities as it is expressed below:

$$\Phi_f \triangleq \bigwedge_{i \in U} f_0(i) = Stat_i$$

Where Φ_f is the logical formula that represents f_0 and $Stat_i$ is a well-formed statement in SOLj. Therefore, we can "apply" Ψ_T to Φ_f by joining them by conjunction which produces the following formula:

$$\Psi_T \wedge \Phi_f \triangleq (\exists f)(\exists R)\Psi_P \wedge \Psi_S \wedge \Phi_f$$

Notice that we can only eliminate the quantifier f in the formula above. As a result we obtain a logical expression that denotes f_0's new version (f'). Also, since we cannot eliminate the quantified variable R (an n-ary relational symbol) given that elimination of relation symbols is still an open problem in second order logic. Therefore, the resulting formula for f' is relatively more complex compared to the one for f_0.

Also, do notice that, we may only eliminate the quantified f in the latter formula. The resulting formula will be the logical expression that denotes our new version of f_0, namely, f', however, notice that, since we cannot eliminate the quantified variable R (as elimination of n-ary quantified relation symbols is an open problem), the resulting expression is not quite the corresponding formula for f' in the same manner Φ_f denoted f_0.

6.4.4 Kripke Versioning Model

The author of [29] defines the semantics of propositional counterfactual logic using a multiple-world-based model. Therefore, we have opted to use a similar model to interpret our program transformations. In this instance, each version of the program will represent a possible world. Once we apply a *program transformer* (as it was defined in the last section) we produce the formula for a new version i.e. the transformation establishes the relationship between two possible worlds in our model (the two program versions). In what follows, we take the liberty of writing $s_i \leftarrow s_j$ to represent the fact that the statement s_i was swapped by statement s_j. The following definition provide a formal interpretation of our counterfactual logic.

Definition 6.4.1 (Kripke Version Model) *A Kripke Version Model $\mathcal{R}$ is a triple $\langle \mathcal{P}, \Rightarrow, P_0 \rangle$ where:*

1. *$\mathcal{P} = \{P_k\}_{k \in \mathbb{N}}$ is the set of all n-line programs which are the different program versions.*
2. *P_0 is the initial program.*
3. *$\Rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ is a binary relation defined the set of all possible program versions. Where $\Rightarrow$ is the smallest relation such that the following properties hold:*
   - *(a) $s_i \leftarrow s_i \triangleq$ statement s_i is left unchanged. This stands for the do nothing transformation.*
   - *(b) $s_i \leftarrow s_j \triangleq$ statement s_j replaces statement s_i, where $s_j \in P_k$. We usually call this primitive transformation, a swap.*
   - *(c) $s_i \leftarrow s_j \triangleq$ statement s_j replaces statement s_i, where $s_j \notin P_k$. Thence, s_k is a new statement.*
   - *(d) $(\forall i) s_i \in P_k$ can be changed only once.*

Moreover, we have assumed that the relation $\Rightarrow$ is *reflexive*, *symmetric*, and *transitive*. Below we justify the latter statement.

1. **Reflexivity:** For any program $P_i \in \mathcal{P}$, it is obvious that the *do-nothing transformation* will yield that any program can be transformed into itself. Therefore, $P_i \Rightarrow P_i$ given that for all $s_j \in P_i$, $P_i = P_i[s_j/s_j]$

2. **Symmetry:** For any programs $P_i, P_j \in \mathcal{P}$ any of the above transformations can be reversed and thence, $P_i \Rightarrow P_j$ implies $P_j \Rightarrow P_i$.

3. **Transitivity:** For any programs $P_i, P_j, P_k \in \mathcal{P}$, applying two or more transformations to a program will yield intermediate versions; this is equivalent to transforming the initial version by composing the transformations into one. Thus, $P_i \Rightarrow P_j$ and $P_j \Rightarrow P_k$ imply that $P_i \Rightarrow^+ P_k$. Where $\Rightarrow^+ \triangleq \Rightarrow \circ \Rightarrow^{n-1}$ and $n > 1$.

6.4.5 Interpreting the Counterfactual Implication

Based on what was stated earlier, the main purpose of the model we have defined is to provide the interpretation for our counterfactual-based *program transformers*. To that end, we let P_0 denote the source code of a given program. Furthermore, we assume we have a counterfactual statement that expressed a change being done to the program as $\phi \boxdot\!\!\rightarrow \psi$, where

- ϕ stands for assertions regarding P_0 and some transformation $s_i \leftarrow s_j$ that implies that $P_1 = P_0[s_i/s_j]$

- ψ stands for assertions regarding P_1

Thus, following the model-theoretic interpretation proposed by Lewis in [29], our version of the counterfactual implication is interpreted as:

$$\mathcal{R} \models \phi \boxdot\!\!\rightarrow \psi \ . \tag{6.1}$$

Where $\mathcal{R}$ denotes our previously defined *Kripke Versioning Model*. Moreover, letting α_i, β_i denote propositional statements about the structure of P_i and P_j respectively, then, we can state that:

$$\phi \triangleq (\bigwedge_{i=1}^{n} \alpha_i) \wedge (P_i \Rightarrow^{+} P_j)$$

$$\psi \triangleq \bigwedge_{j=1}^{m} \beta_j$$

The $\Rightarrow^{+}$ symbol denotes the positive/transitive closure regarding the relation $\Rightarrow$. Moreover, from an initial version source-code version P_0 we generate future alternative versions via the application of one ore more transformations. The current version's properties and the changes made to this version imply the desired properties the new version would have. Therefore:

$$\mathcal{R} \models \phi \boxdot\!\!\rightarrow \psi \triangleq (\exists_{\min} k \in \mathbb{N})(\textstyle\bigwedge_{i=1}^{n} \alpha_i) \wedge (P_0 \Rightarrow^{k} P') \rightarrow (\textstyle\bigwedge_{j=1}^{m} \beta_j) \ .$$

The expression above means that there is a minimal number of transformation steps such that provided that the properties of our initial program P_0 ($\bigwedge_{i=1}^{n} \alpha_i$) and the transformation that leads from the initial version to the desired future version are true, then it follows that the desired properties (encoded in our logic) of the future version also hold.

6.4.6 Fragment of Counterfactual Logic

Given that we are striving to provide programmers with an mechanical procedure and a tool whose purpose is to aid in reasoning about the properties that would hold for future versions of a given program, we provide a logical calculus which will enable us to infer such properties in a algorithmic manner. Thus, we provide the set of inference rules taken from [29] and known as **VC** logic; these rules will help us formalize realize our approach's proof theoretical fragment.

1. $\dfrac{\vdash \phi \rightarrow \psi, \phi}{\vdash \psi}$ Modus Ponens

2. $\dfrac{\vdash \neg\psi \ \Box\!\!\rightarrow \psi}{\vdash \phi \ \Box\!\!\rightarrow \psi}$ Vacuity

3. $\dfrac{}{\phi \ \Box\!\!\rightarrow \phi}$ Reflexivity rule

4. $\dfrac{\phi \ \Box\!\!\rightarrow \psi}{\phi \rightarrow \psi}$ Counterfactual-Elimination rule

5. $\dfrac{\vdash (\bigwedge_{i=1}^{n} \lambda_i) \rightarrow \psi}{\vdash [\bigwedge_{i=1}^{n} (\phi \ \Box\!\!\rightarrow \lambda_i)] \rightarrow (\phi \ \Box\!\!\rightarrow \psi)}$ Conditional Deduction rule

6. $\dfrac{\vdash \Box(\phi \rightarrow \psi)}{\vdash \phi \ \Box\!\!\rightarrow \psi}$ Counterfactual Necessity Theorem

The reader may wonder about the *Counterfactual Necessity Theorem* (CNT). First of all, a simple intuitive argument can be made in order to show why this is a theorem of **VC**. If we recall Lewis's system of spheres [29] and taking into account that **VC** subsumes many of the standard proof-theoretical system of modal logic, then it is easy to see that the strict implication $\Box(\phi \rightarrow \psi)$ will hold for all worlds in the system of spheres. Hence, the counterfactual implication $\phi \ \Box\!\!\rightarrow \psi$ follows immediately by definition.

If the reader is still unconvinced about the theorem-hood regarding the CNT we suggest she skips to our appendix and/or review [22] in which the authors give sufficient arguments about why CNT is a theorem of **VC**.

6.4.7 Temporal Logic Fragment

Since a SOLj network exhibits a concurrent form of computation, we have to employ a more expressive inference system than the one used for sequential computations (i.e. first order logic). Thence, we need a fragment of temporal logic in order to encode properties of a SOLj network. Such fragment is given below [3].

1. $\dfrac{\vdash \phi}{\vdash \Box\phi}$

[3] ϕ ranges over all the well-formed first-order logic expressions

2. $\Box\phi \leftrightarrow \phi \wedge \bigcirc\Box\phi$

3. $\Box(\phi \rightarrow \psi) \rightarrow (\Box\phi \rightarrow \Box\psi)$

4. $\bigcirc(\phi_1 \vee \phi_2) \leftrightarrow \bigcirc\phi_1 \vee \bigcirc\phi_2$

5. $\dfrac{\vdash \phi_1 \rightarrow \bigcirc(\phi_1 \vee \phi_2)}{\vdash \phi_1 \rightarrow \phi_1 \,\mathcal{U}\, \phi_2}$

6.5 A Counterfactual Example - Auto-Regulated Power Generation Network

Using the agent types defined in section 6.2, we will briefly show how to apply our counterfactual-based approach to a practical situation.

6.5.1 Formal Characterization of a SOLJ Network's Behavior

As it has been stated before, a SOLJ network is a *reactive system* and thus it continually responds to different stimuli. In [35], we used counterfactual logic to assert properties of sequential programs. However, in this instance, we are dealing with a perpetually functioning reactive system and thus we need to integrate *temporal logic* constructs into our logic.

In the most simple case, a run in our system is just an infinite sequence of the basic actions (function/service calls) performed by the network agents. Therefore, the following expression denotes a run of the system we specified in 6.2

$$\rho_1 \triangleq [(oC \wedge oR \wedge oV) \wedge (rD \wedge \neg rU \vee rU \wedge \neg rD)]$$

Where oC, oR, and oV denote the calls to the `outputCurrent`, `outputResistance`, and `outputVoltage` update functions respectively. While ru and rD denote calls to the `revUpEngine` and `revDownEngine` services. Thus, in terms of temporal logic, the runtime behavior exhibited by the agents defined in Figures 7.2 and 7.3 is:

$$\Box\rho_1$$

Which is interpreted as an infinite sequence of calls to the services shown in ρ_1.

6.5.2 Source Code Change

Let us assume that there has been a change in the requirements of the system which imply that the `genAgent` modules need only to augment the generators revolution when the current voltage exceeds certain threshold and decrease the revolutions when the current goes below certain threshold. In terms of our logic, the actual state of the `genAgent` module is given by the following list:

- $f(1) =$ "`if (@C(currVoltage) && @C(currCurrent))`"
- $f(2) =$ "`return EGenerator:RevUpEngine(revDelta)^revUpCont`"
- $f(3) =$ "`else`"
- $f(4) =$ "`return EGenerator:RevDownEngine(revDelta)^revDownCont`"

Our intended change would alter the latter formulas into the ones shown in the following list:

- $f'(1) =$ "`if (@C(currVoltage))`"
- $f'(2) =$ "`return EGenerator:RevUpEngine(revDelta)^revUpCont`"
- $f'(3) =$ "`if (@C(currCurrent))`"
- $f'(4) =$ "`return EGenerator:RevDownEngine(revDelta)^revDownCont`"

The desired basic behavior for this new version of the SOLj agent network is given by the following expression:

$$\rho_2 \triangleq [(oC \wedge oR \wedge oV) \wedge (rD \vee rU)]$$

Let Ψ_{SN} represent the current network structure in the same manner as the formulas defined in 6.4.2. Also, let $\texttt{swap}_{1,3}^{SN}$ denote the following formula:

$$\begin{aligned}(u \neq 2) \wedge (u \neq 4) \quad &\rightarrow \quad (f(u) = f'(u)) \\ &\wedge \quad (f(1) = f'(3)) \\ &\wedge \quad (f(3) = f'(1))\end{aligned}$$

6.5.3 Proof

In this section we will show the counterfactual implication statement that encodes the desired program transformation and its effects. Let Ψ_{SN}^1 and Ψ_{SN}^2 respectively denote the logic expressions for the current and desired configurations of the SOLj network. Furthermore, let ρ_1 and ρ_2 denote the operation sequences that respectively correspond to Ψ_{SN}^1 and Ψ_{SN}^2. Moreover, let $\texttt{swap}_{1,3}^{SN}$ be defined as before (i.e. the formula that encodes the change between Ψ_{SN}^1 and Ψ_{SN}^2). In an intuitive way our claim may be formulated as:

Given the initial structure of the system ($\Psi_{\mathbf{1}}^{\mathbf{SN}}$) and the desired change ($\texttt{swap}_{\mathbf{1,3}}^{\mathbf{SN}}$), then the system would eventually transition from the initial behavior (denoted by $\Box\rho_{\mathbf{1}}$) to the desired behavior (denoted by $\Box\rho_{\mathbf{2}}$.)

Hence, using the already known fragments of counterfactual and temporal logics, we can symbolize the latter assertion as:

$$\Psi_{SN}^1 \wedge \texttt{swap}_{1,3}^{SN} \boxdot\!\!\rightarrow (\Box\rho_1 \,\mathcal{U}\, \Box\rho_2)$$

SOLj Network Axioms

We need to assume several facts in order to derive the desired counterfactual proof. First of all, we readily assume that the formulas that denote each version configuration (i.e. Ψ^1_{SN} and Ψ^2_{SN}) materially implies their respective behaviors. Thence:

1. $\Psi^1_{SN} \rightarrow \Box\rho_1$
2. $\Psi^2_{SN} \rightarrow \Box\rho_2$

Also, by definition of $\Box$ in the fragment of temporal logic we are using, we can assert:

1. $\Psi^1_{SN} \rightarrow \rho_1 \wedge \bigcirc\Box\rho_1$
2. $\Psi^2_{SN} \rightarrow \rho_2 \wedge \bigcirc\Box\rho_2$

Using Propositional Logic we can *weaken the consequent* on both implications and hence we have:

1. $\Psi^1_{SN} \rightarrow \bigcirc\Box\rho_1$
2. $\Psi^2_{SN} \rightarrow \bigcirc\Box\rho_2$

Temporal Logic Proof

The following step requires us to recall the definition of both ρ_1 and ρ_2:

$$\rho_1 \triangleq [(oC \wedge oR \wedge oV) \wedge (rD \wedge \neg rU \vee rU \wedge \neg rD)]$$

$$\rho_2 \triangleq [(oC \wedge oR \wedge oV) \wedge (rD \vee rU)]$$

By simple propositional logic we can assert that $\rho_1 \rightarrow \rho_2$. Since temporal logic allows us to use the *Necessitation Rule* then we know $\Box(\rho_1 \rightarrow \rho_2)$ which in turn entails $\Box\rho_1 \rightarrow \Box\rho_2$. Since we

already know that $\Psi^1_{SN} \rightarrow \Box\rho_1$ therefore, by transitivity of material implication, we can state that $\Psi^1_{SN} \rightarrow \Box\rho_2$. Again, taking into account the definition of $\Box$ and by *weakening the consequent* we can assert:

1. $\Psi^1_{SN} \rightarrow \bigcirc\Box\rho_1$
2. $\Psi^1_{SN} \rightarrow \bigcirc\Box\rho_2$

Notice that it is the case that ρ_1 implies ρ_2, however, they are not equivalent. Hence, the system could exhibit either of the two behaviors. Thence, given the latter implications and by weakening and strengthening the consequent and antecedent respectively, we can assert the following:

$$\Psi^1_{SN} \wedge \texttt{swap}^{SN}_{1,3} \rightarrow \bigcirc\Box\rho_1 \vee \bigcirc\Box\rho_2$$

Strengthening the antecedent in the latter step may seem arbitrary, but regarding our *Program Transformers*, it means that we are applying the code transformation to the first version of the SOLj Network. The temporal logic fragment we are employing allows us to assert $\Psi^1_{SN} \rightarrow \bigcirc(\Box\rho_1 \vee \Box\rho_2)$. Based on the last step and using temporal logic again, we can assert what follows:

$$\Psi^1_{SN} \wedge \texttt{swap}^{SN}_{1,3} \rightarrow \Box\rho_1 \,\mathcal{U}\, \Box\rho_2$$

Counterfactual Proof

Temporal logic allows us to use the *necessity rule* and thus assert the following:

$$\Box[\Psi^1_{SN} \wedge \texttt{swap}^{SN}_{1,3} \rightarrow (\Box\rho_1 \,\mathcal{U}\, \Box\rho_2)]$$

Lastly, we employ the CNT from our counterfactual logic fragment and conclude that:

$$\Psi^1_{SN} \wedge \mathtt{swap}^{SN}_{1,3} \boxdot\!\!\rightarrow (\Box\rho_1 \,\mathcal{U}\, \Box\rho_2) \blacksquare$$

6.6 Theoremhood of The Counterfactual Necessity Theorem within The VC-Logic

Earlier in section 6.5.3 we established how our claim was a theorem of our proposed logic. In this section we used the *Counterfactual Necessity Theorem* (CNT) whose proof was not given immediately. The purpose of this section is to serve as a complement to the simple model-theoretic justification we used. In what follows, we will provide a two fold argument of why CNT is a theorem. The first subsection will re-define the notion of *necessity* (in our case, the global temporal operator $\Box$) in terms of the counterfactual conditional. The second subsection will establish the formal proof of CNT's theoremhood[4].

6.6.1 Necessity in Counterfactual Logic

In [22], the authors propose and prove a definition of necessity based on the counterfactual implication. We provide such definition below:

$$\Box\alpha \leftrightarrow (\neg\alpha \boxdot\!\!\rightarrow \bot)$$

Although the justification of such equivalence falls outside of the scope of this section, we can intuitively and readily justify it based on Lewis's model of spheres. A statement is necessary in this model when it cannot counterfactually entail a contradiction. Moreover, if there are no α-worlds in which contradiction holds then α must be a necessary statement. A more involved explanation and a proof-theoretical justification of this can be found in [22].

In order to show the theoremhood of CNT we need to prove an additional auxiliary equivalence:

[4]The proofs given in this section are adaptations of those shown in [22]. We saw the need of expanding the proofs as we found that they were too succinct.

$$(\neg\alpha \boxdot\!\!\rightarrow \bot) \leftrightarrow (\neg\alpha \boxdot\!\!\rightarrow \alpha)$$

By proving that the latter equivalence holds in **VC** we would be in position to assert $\Box\alpha \leftrightarrow (\neg\alpha \boxdot\!\!\rightarrow \alpha)$. Let α and β range over well-formed expressions in temporal logic. Let DWC be an abbreviation of the *Deduction Within Conditional* rule of **VC**. Last but not least, we expand our language's vocabulary with $\bot$ to denote proof-theoretical contradiction.The required proof is given below:

1. $(\alpha \wedge \neg\alpha) \rightarrow \bot$ $\bot$-Definition
2. $[(\neg\alpha \boxdot\!\!\rightarrow \alpha) \wedge (\neg\alpha \boxdot\!\!\rightarrow \neg\alpha)] \rightarrow (\neg\alpha \boxdot\!\!\rightarrow \bot)$ DWC in 1
3. $\neg\alpha \boxdot\!\!\rightarrow \neg\alpha$ Reflexivity
4. $\neg\alpha \boxdot\!\!\rightarrow \alpha$ Assumption
5. $(\neg\alpha \boxdot\!\!\rightarrow \neg\alpha) \wedge (\neg\alpha \boxdot\!\!\rightarrow \alpha)$ $\wedge$-Intro in 3,4
6. $\neg\alpha \boxdot\!\!\rightarrow \bot$ Modus Ponens in 2 and 5
7. $(\neg\alpha \boxdot\!\!\rightarrow \alpha) \rightarrow (\neg\alpha \boxdot\!\!\rightarrow \bot)$ $\rightarrow$-Intro in 4-6
8. $\bot \rightarrow \alpha$ $\bot$-Triviality
9. $(\neg\alpha \boxdot\!\!\rightarrow \bot) \rightarrow (\neg\alpha \boxdot\!\!\rightarrow \alpha)$ DWC in 8
10. $[(\neg\alpha \boxdot\!\!\rightarrow \bot) \rightarrow (\neg\alpha \boxdot\!\!\rightarrow \alpha)] \wedge [(\neg\alpha \boxdot\!\!\rightarrow \alpha) \rightarrow (\neg\alpha \boxdot\!\!\rightarrow \bot)]$ $\wedge$-Intro 7 and 9
11. $(\neg\alpha \boxdot\!\!\rightarrow \bot) \leftrightarrow (\neg\alpha \boxdot\!\!\rightarrow \alpha)$ $\leftrightarrow$-Intro in 10 ■

6.6.2 Proof of CNT

Using the latter subsection's results we will give the derivation which shows that CNT is a theorem of **VC**.

1. $\Box(\alpha \rightarrow \beta) \rightarrow [\neg(\alpha \rightarrow \beta) \boxright (\alpha \rightarrow \beta)]$ $\Box$-def, $\wedge$-elim
2. $\bot \rightarrow (\alpha \rightarrow \beta)$ $\bot$-Triviality
3. $[\neg(\alpha \rightarrow \beta) \boxright \bot] \rightarrow [\neg(\alpha \rightarrow \beta) \boxright (\alpha \rightarrow \beta)]$ DWC in 2
4. $[\neg(\alpha \rightarrow \beta) \boxright (\alpha \rightarrow \beta)] \rightarrow [\alpha \boxright (\alpha \rightarrow \beta)]$ Vacuity
5. $\Box(\alpha \rightarrow \beta) \rightarrow [\alpha \boxright (\alpha \rightarrow \beta)]$ Transitivity in 1 and 4
6. $[\alpha \wedge (\alpha \rightarrow \beta)] \rightarrow \beta$ Instantiation of Modus Ponens
7. $[(\alpha \boxright \alpha) \wedge (\alpha \boxright (\alpha \rightarrow \beta))] \rightarrow (\alpha \boxright \beta)$ DWC in 6
8. $\Box(\alpha \rightarrow \beta)$ Assumption
9. $[\alpha \boxright (\alpha \rightarrow \beta)]$ Modus Ponens in 8 and 5
10. $\alpha \boxright \alpha$ Reflexivity
11. $(\alpha \boxright \alpha) \wedge [\alpha \boxright (\alpha \rightarrow \beta)]$ $\wedge$-Intro in 9 and 10
12. $(\alpha \boxright \beta)$ Modus Ponens in 7 and 11
13. $\Box(\alpha \rightarrow \beta) \rightarrow (\alpha \boxright \beta)$ $\rightarrow$-Intro in 8-12 ■

Since CNT is indeed a theorem in **VC** we can freely use it in our main proof. Additionally, although the authors of [22] use $\Box$ as *metaphysical necessity* or simply alethic necessity, they also assert that any logic which complies with the axioms of system **K** will be able to incorporate this definition of $\Box$. Since both **VC** and temporal logic subsume **K** we can assert that the redefinition of this operator based on the counterfactual implication is not conflictive.

6.7 Conclusion

We have introduced a framework for guaranteing safe source-code changes regarding reactive systems (i.e. sensor networks). The principles exhibited by this framework are realized by two pieces of technology. The first is SOLj which is a domain-centric language based in the Software Cost Reduction project (SCR). This language lets us precisely specify the desired behavior the reactive agents will exhibit during execution. In a nutshell, the SOLj language allows us to provide the functional specification regarding a given reactive system. Moreover, we are able to readily compile the agents' SOLj code into standard JAVA source code. The next piece of technology that allows us to realize our approach is known as Secure Infrastructure for Networked Systems (SINS). SINS provides a run-time environment that enables a set of SOLj agents to run in a manner which does not compromise system's integrity (i.e. SINS helps us realize non-functional requirements as general security, access policies, etc.).

Also, we have shown that our counterfactual verification approach (defined for sequential cases in [35]) can also be used in simple reactive systems. We defined the language for counterfactual logic, its syntax and semantics via *program transformers* formulas and *Kripke structures* respectively. In this instance, we were required to augment our logical language with a fragment of temporal logic (as shown in [43]). The reason for the latter is that reactive systems exhibit properties that are expressed in terms of *safety* and *liveness* which can be conveniently expressed using temporal logic. We first conjectured and then showed that the desired change and the current structure of the SOLj code counterfactually implied the desired change by encoding it as a sentence in our logical language and then proving it was a theorem in the logic.

Chapter 7

Reasoning About Security in Sensor Networks

7.1 Introduction

We present a formal framework for reasoning about security concerns in the context of embedded sensor networks. We first provide an agent-based programming model for sensor networks. A logical framework enables reasoning about security, safety, and integrity with respect to usage of resources in this model. Embedded sensor networks often operate in rapidly changing mission-critical environments where both functional and non-functional requirements can alter dynamically in an unforeseen manner. The network may need to be reconfigured and reprogrammed in response to changes in its operating conditions. We provide a framework based on counterfactual logic to formally represent changes to the system and perform what-if reasoning about their impact on security and safety even before they have been applied.

Sensor network systems (SNSs) are distributed embedded monitoring and control systems that receive percept streams from the environment and generate reactions that can be actuated through actuators [25]. They often operate in rapidly changing *mission-critical environments* where both functional and non-functional requirements can alter dynamically in an unforseen manner. The system may need to be reconfigured and reprogrammed in response to changes in its operating conditions. Since they are often deployed in mission-critical applications, modifications to SNSs should be performed under utmost caution. Any disruption that results in system failure and/or malfunction can yield catastrophic results. Therefore, we should strive to develop methods that ensure that the correctness, security, and trust of SNSs are preserved even under structural changes and reconfigurations in response to an evolving environment.

For many applications involving SNSs, requirements change dynamically in a rapid, unpredictable, and continuous fashion. In applications that drive search and rescue missions, any oper-

ational expansion/contraction requires dynamic reorganization of the system. In surveillance and monitoring applications using sensor networks, nodes can die changing the network structure or the environmental parameters can change requiring reconfiguration of the system.

We need to develop techniques, tools, and methods that help build, manage, and maintain SNSs whose requirements keep changing during their life-cycle. Conventional software engineering approaches used in industry are inadequate in rapidly evolving mission-critical scenarios since they do not allow reasoning about changes arising out of dynamically evolving system requirements.

We present a formal framework for reasoning about security and safety concerns in the context of embedded sensor networks. We first provide an agent-based programming model for sensor networks. A logical framework enables reasoning about security, safety, and integrity with respect to usage of resources in this model. We then provide a framework based on counterfactual logic to formally represent changes to the system and perform what-if reasoning about their impact on security and safety even before they have been applied.

7.2 A Programming Framework for Sensor Networks

In the context of SNSs we have created a framework for the development of secure sensor network applications [6]. This framework is embodied by two main components. The first is the Secure Operations Language for JAVA (SOLj) which is an event-based domain-oriented synchronous programming extension of JAVA used to write the specification for agents which, in this context, are the software counterparts of one or more sensors. The second element is the Secure Infrastructure for Networked Systems (SINS). SINS a is a distributed run-time system in which the SOLj agents are deployed. The SINS run-time provides the necessary features that help uphold the system's security policies. Also, SINS can run both on the Android™ and Sun SPOT™ planforms. For the sake of brevity we refer the reader to [6] for the details of other SOLj elements such as *definitions* and the Secure Infrastructure for Networked Systems (SINS).

7.2.1 SOLj - Secure Operations Language-JAVA

Given that SOLj is an extension of JAVA, it is presented as modular extensions to its core language. A *module* comprises the specification unit in SOLj; it is composed of type definitions, variable declarations, service declarations, assumptions, guarantees, and definitions. In the future we will use the word *agent* to denote a module instance. A SOLj module may include *attributes* as they are described below:

`deterministic` Declares a module that does not exhibit nondeterministic behavior. The compliance with this attribute is checked by the SOLj compiler.

`reactive` Declares a module that will cause a state change only when its (visible) environment fires an event via a state change or by invoking a method. Also, this attribute denotes that the module's response to an event will happen in the next immediate step.

The type definition section contains user-defined types as well as enumerated types. The JAVA comment `//@type definitions` precedes the "type definitions" section. It provides SOLj directive to the compiler indicating the start of the type definition section. The variable declaration section defines three types of variable which are explained below:

"monitored variables" variables in the environment that influence the agent's behavior.

"controlled variables" variables in the environment that are changed by the agent's behavior.

"internal variables" this variables reflect the agent's state.

The JAVA comment `//@Services` precedes the "service declarations" section which declares the agent's external interface. It contains the methods that realize the services the module provides. For each method declaration within a service, the SOLj language provides the capability of declaring the corresponding preconditions and postconditions which denote the conditions

under which each service should start and terminate. The preconditions and postconditions are encoded as arithmetic expressions and type declarations respectively. A type declaration is denoted by a type judgement expression `T:x` where `x` is a variable and `T` is a type. These constraints are enforced at runtime in a dynamic manner. Also, each service invocation must use a "continuation variable". This variable includes a boolean field called "done" which is assigned a value of `true` once the service invocation is ready to provide a return value.

The `//@Assumptions` comment denotes the starts of the *assumptions* section which includes assumptions that determine the agent's correct way to operate. If any of these assumptions is violated by the environment, the agent's execution is aborted. The `guarantees` section contains the agent's required safety properties. The `definitions` section provides *update functions* which denote variable definitions. These specify the corresponding values for internal and controlled variables. For the sake of future references, we will distinguish between *monitored variables* whose values are given by the environment, and *dependent variables* which are those whose values that are the result of SOLj agents computations. These values are obtained using the values of monitored variables and (possibly) past dependent variable's values.

Based on the work presented in [6] we are fully capable of transforming well-formed SOLj code (as it is defined above) and produce the equivalent JAVA source via a SOLj-JAVA compiler.

7.2.2 SOLj Events

Based on the Software Cost Reduction (SCR) method's SCR Abstract Language (SAL), SOLj has been provided with the capability of defining events [23]. Intuitively, SCR events can be interpreted as state changes. Moreover, the occurrence of an event is triggered when a variable has its value changed. This is done by update functions which change the dependent variable's value. In the subsequent `@PREV(c)` denotes the value of variable c in the previous state. `@T(c)` denotes when variable c changes from *false* to *true*. `@F(c)` denotes when the variable c changes from *true* to *false*. `@C(c)` denotes when the value of the variable c changes between the previous and current

states.

$$\begin{aligned}
@\mathtt{T}(\mathtt{c}) &=^{def} \neg @\mathtt{PREV}(\mathtt{c}) \wedge \mathtt{c} \\
@\mathtt{F}(\mathtt{c}) &=^{def} @\mathtt{PREV}(\mathtt{c}) \wedge \neg \mathtt{c} \\
@\mathtt{C}(\mathtt{c}) &=^{def} @\mathtt{PREV}(\mathtt{c}) \neq \mathtt{c}
\end{aligned}$$

An initialization method (`init`) assigns starting values to all dependent variables (controlled or internal). Each module contains an `init` method. Each dependent variable is updated by just one of the update functions. Moreover, a *dependency relation* is induced by the interplay between update functions and dependent variables. We denote this relation with D_m. Let `a` and `b` be two dependent variables, then we say that $(\mathtt{a}, \mathtt{b}) \in D_m$ if and only if `a` is updated by the function corresponding to `b`. The fact that `a` may depend on the previous values of other variables and itself does not influence the dependency relation. Also, a *dependency graph* may be derived from D_m by interpreting the set of dependent variables as the nodes and each $(\mathtt{a}, \mathtt{b}) \in D_m$ as the edges [1]. For each module, we need to consider its corresponding dependency graph as acyclic.

From a simplified point of view, a SOLj program executes in a sequence of *steps* and each step is preceded by the triggering of an event. The module's dependency relation induces the order in which the variable updates and service invocations are carried out. Each computation step may be decomposed as follows:

1. The environment or the agent itself triggers an event in a nondeterministic manner
2. Each agent *responds* to this event by modifying the values of its dependent variables

From an external point of view, the updates and service invocations can be thought of happening in a synchronous manner (as it is dictated by the *Synchrony Hypothesis* and exemplified by languages such as Esterel and LUSTRE [21]). The latter implies that all dependent variable

[1] The notion of a dependency relation is easily extended to the entire system.

updates and service invocations that occur as a response to an event triggering, happen before the next event is triggered.

The SOLj-based framework allows us to use a model-driven approach wherein agents are specified and verified at a high level and then compiled to JAVA code that can run on Sunspot or Android platforms.

7.3 Example - SOLj Distributed Resource Access System

7.3.1 General Aspects

In the context of this example, the system will exhibit two types of SOLj agents. The first type is a *Resource Monitor Agent* (RMA). This type of agent has control of a given subset of the system's resources. The latter is due to the fact that the *Access Control List* (ACL) is allocated to different nodes in the system. The reason for the latter is to prevent single-point-of-control failures and/or ill-intended attacks. The other type of system agent is the *Resource Requester Agent* (RRA). These agents, aside from executing other specific tasks, use distributed protected resources in order to carry-out their work. In this example we will only emphasize the security concerns while assuming that the system carries out a non-specified set of functional requirements.

7.3.2 System Description

Our security distributed system is mainly composed of two agent types:

Resource Monitor Agent (RMA) This agent type is in charge of granting access to protected resources. Each agent instance oversees accesses to a given set of resources based on the entries of a given co-located access control list. The security concerns are distributed and hence, a security agent may grant access to a non-local resource by acting as a *proxy-requester*.

Resource Request Agent (RRA) This agent type has as its main task the duty of requesting access to the set of RMAs in the system. Its state is composed of the resources it is currently using.

The resource controlled by an RMA are physically distributed and co-located with each RMA instance. Also, the ACL is distributed as well. For a given RMA its corresponding ACL will contain only entries related to the agents that have permission to use the resources it controls.

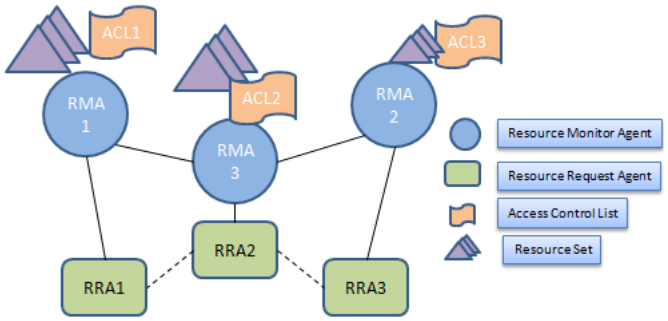

Figure 7.1: Distributed security system schematics

An example of an instance of a distributed security system is given in the diagram shown in Figure 7.1. The SOLj source code examples for both types of agents RMA and RRA are given in Figures 7.2 and 7.3 respectively.

7.3.3 SOLj Security Features

Based on the work exposed in [41], we explain how *enforceable* safety and security policies [44] are expressed in SOLj as *enforcement automata* (also known as *security agents* [5], [41]). The enforcement mechanism of SOLj works by terminating all executions of a program for which the policy being enforced no longer holds. For reasons of readability and maintainability, we prefer to use explicit automata for enforcing safety properties and security policies, although any language that allows references to previous values of variables may suffice. Unlike assertions, where no additional state is maintained, SOLj enforcement automata may include additional variables that are updated during the transitions of the automata.

```
//@Module Declaration
deterministic reactive module ResourceMonitorAgent

//@Type definitions
Permission  = SecurityRecord
ResourceSet = OrderedSet<Resource>
AccessControlLst = OrderedSet<Permission>

//@Controlled variables
ResourceSet resourceList;

//@Internal variables
boolean usedResource2;

//@Monitored variables
%ResourceSet usedResources;
AccessControlList secList;

//@definitions
//@initialization
void init(){
    %usedResources = null;
    usedResource2 = false;
}
//@update functions
Resource grantResourceAccess(int requesterID, int resourceID1){
    if(@F(secList.isInUse(resourceID1) &&
       @T(secList.hasPermission(requesterID, resourceID1) &&
       @T(!usedResource2)){
        resourceList.getResource(resourceID1).setInUse(true);
        return resourceList.getResource(resourceID1);
    }
    if(@F(secList.isInUse(resourceID2) &&
       @T(secList.hasPermission(requesterID, resourceID2)){

        if(!usedResource2){
            usedResource2 = true;
        }
        resourceList.getResource(resourceID2).setInUse(true);
        return resourceList.getResource(resourceID2);
    }
```

Figure 7.2: SOLj code for the `ResourceMonitorAgent` module

```
//@Module Declaration
deterministic reactive module ResourceRequestAgent

//@Type definitions
Permission  = SecurityRecord
ResourceSet = OrderedSet<Resource>

//@Controlled variables
ResourceSet heldResources;
int agentID;

//@Monitored variables
AccessControlList secList;

//@Services
//@ Resource ResourceMonitorAgent:grantResourceAccess(resourceID)
//@     pre  = !heldResources.contains(resourceID)
//@     post = heldResources.contains(resourceID)

//@Update Functions
void getResourceAccess(int resourceID1, int resourceID2){
    heldResources.addResource(grantResourceAccess(resourceID1));
    heldResources.addResource(grantResourceAccess(resourceID2));
}
```

Figure 7.3: SOLj source code for `ResourceRequestAgent`

The classical way of specifying the correct safe use of shared resources would be to write a so-called *class invariant*, often specified as predicates on the "old" and "new" values of program variables. Languages such as Eiffel [31] with explicit support for Design by Contract [32] include constructs for specifying and checking such invariants. However, presently popular object-oriented programming languages lack such mechanisms, and therefore treat class invariants mostly as comments, and provide no tool support to analyze them. A unique feature of SOLj is the ability to perform such checks on existing implementations in a language-neutral manner.

SOLj Safety Automata

We show how SOLj agents are used to enforce safety properties. Within the example in Section 7.3 we show two types of agents (*Resource Monitor Agent* and *Resource Request Agent*) that cooperate

in order to ensure that a set of distributed resources are used accordingly since no resources shall be used by a given agent while it is being held by another agent i.e., an RMA will never grant a resource that is already in use.

SOLj Security Automata

Another aspect shown in our example is security. In this sense, the security policy being upheld is that an agent may access a resource provided that it has already been given access to it. If the latter does hold at the moment of the request, access to the resource is denied and the request has no effect. Furthermore, notice that a given resource can only be requested once, therefore, in the context of two subsequent requests for a resource only the first one will be honored while the second one has no effect whatsoever.

7.3.4 Source Code Description

In Figures 7.2 and 7.3 we have already given examples of an RMA and a RRA respectively. In this section we describe the code presented in the referred figures.

Resource Monitor Agent (RMA)

For the purpose of this example our RMA agent manages a list of resources. Details about this type of agent are given below. We describe each agent type by outlining the elements that comprise each of the agent-code sections.

Type Definitions

Permission This type represents the right of any given requester agent to access a resource, assuming such resource is not in use. For this type we assume that a special kind of *record type* (`SecurityRecord`) already exists.

ResourceSet This type represents the group of resources an RMA controls. For this type, we assume that a the type `Resource`, a native *record type*, already exists.

AccessControlLst This type represents the state (free/held) of the resources controlled by the agent and whether or not a given agent has access to a specific resource.

Controlled Variables

resourceList This variable represents a lists which is used to keep track of the resources currently in use and controlled by the RMA.

Monitored Variables

secList This variable represents a list that is used to keep track which RRA agents have access to the set of controlled resources.

Internal Variables

The only internal variable for this module is `isUsedResource2` which denotes if the second controlled resource is being used.

Update Functions

grantResourceAccess This function takes as parameters three integers (`resourceID1`, `resourceID2`, and `requesterID`) and does the following:

1. Waits (via event-triggers) until `resourceID1` is freed.
2. `resourceID1` is assigned to the requester provided it has never used `resourceID2` indicated by the `usedResource2` variable being false. `resourceID1` is flagged as being used.

3. Waits (via event-triggers) until the `resourceID2` is freed.
4. It flags `resourceID2` as being used. If `resourceID2` was never previously used by the requester (indicated by the flag `usedResource2` being false), the flag `usedResource2` is turned on

The above security policy cannot be expressed in JAVA's built-in security model that either always allows access to a resource or never.

Resource Requester Agents (RRA)

In the context of this example an RRA, aside from other functional concerns, has the role of asking RMAs access to resources. We will describe this type of agent by outlining the elements in each one of its sections. We will not cover the `Type Definitions` section given that RMAs and RRAs share the same used types.

Controlled Variables

heldResources This variable denotes a list which keeps track of the resources being used by the RRA.

Update Functions

getResourceAccess This function calls the `grantResourceAccess` service on the corresponding RMA and updates the `heldResources` list with the requested resource.

7.4 Secure Counterfactuals in Sensor Networks

It is widely known that changes regarding security policies are, more often than not, frequent and pervasive. System administrators always struggle to uphold the security concerns of the systems

they help manage. The system's security is under constant threat in this continuous change environment. Traditionally, changes to security policies are partially tested in a development environment. However, these environments lack the scale/size that characterizes a live production environment. The live environment's sheer size may bring forth defects that were not captured during the testing phase. The latter will surely mean that the ill-modified security policy will cause a non-trivial breach. This scenario was mainly caused by changing the security policy. It would be useful to asses the impact of a changed policy without changing the policy itself. It is in this setting where counterfactual logic plays a crucial role.

Counterfactual logic aids us in what-if reasoning about statements that are not matter of fact. Lewis [29] provided a sound and complete inference system and also showed that this logic was a decidable one. We have created a logical calculus based on counterfactual logic. This calculus allows us to express properties that would take place in a future version of a given security policy and verify that indeed these properties would hold if the changes between the new and old version were applied.

7.4.1 The Language of Counterfactual Logic

As it is shown in [29] we will shortly define the language that comprises the logic of counterfactuals (p_i $denotes$ a $propositional$ $variable$)

$$\phi ::= p_i | \neg\phi | \phi \wedge \phi | \phi \vee \phi | \phi \rightarrow \phi | \phi \boxdot\!\!\rightarrow \phi$$

The counterfactual expression $\phi \boxdot\!\!\rightarrow \psi$ should be read as: **if it had been the case that ϕ, it would have been the case that ψ**. Therefore, if we had a statement that contained an antecedent that expressed the properties of a given security policy and the required changes to create the new version and the consequent expressed the properties that the new version would exhibit, then it seems plausible that we can encode such a statement using counterfactual logic.

7.4.2 Formal Representation of A Security Model

In this section we will define a simple variation of the Access Control Matrix (ACM) model. This model was first introduced in [28] and [16]. We have chosen this model due to its simplicity and intuitive nature and widespread use [28]. First, we will define three sets: S, O and A which are respectively the set of : subjects, objects and actions. The set of subjects contains the active entities on the system (i.e., users, computer systems, etc.); the set of objects denotes the set of entities over which subject are allowed or denied a certain action. The set of actions denotes those tasks which a subject can perform on a given object. Hence:

$$\begin{array}{lcll} S & = & \{s_i\}_{i \in I} & \text{The set of subjects} \\ O & = & \{o_j\}_{j \in J} & \text{The set of objects} \\ A & = & \{Read, Write\} & \text{The set of actions} \end{array}$$

Thus, we formally define an *access control matrix* as a function $M : S \times O \rightarrow 2^A$ which takes an ordered pair composed of a subject and an object and assigns to them a subset of the possible set of actions. Moreover, in this instance we will have to work with M's *intensional* or set representation and thence:

$$M \triangleq \{(s_i, o_j, \alpha_k)\} \text{ where } \alpha_k \in 2^A$$

Encoding Change in the ACM model

Let $M = \{(s_i, o_j, \alpha_k)_{i,j,k \in \mathbb{N}}\}$ be the current version of the access control matrix. We can encode the state of any ACM by using the following formula:

$$\Psi_M \triangleq \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} \bigwedge_{k=1}^{p} (s_i, o_j, \alpha_k) \in M$$

We will simplify the latter expression using the following notation:

$$\Psi_M \triangleq \bigwedge_{i,j,k} (s_i, o_j, \alpha_k)$$

For the sake of simplicity we will define a change to the ACM as a change to the members of the action-set of a given triple. Hence, a change may be represented as:

$$(s_i, o_j, \alpha_k) \Rightarrow^c (s_i, o_j, \alpha'_k)$$

Furthermore, let $\mathcal{M}$ denote the class of all possible ACM versions. Therefore, $\Rightarrow^c$ can be thought of as a binary relation over $\mathcal{M}$ and

$$M \Rightarrow^c M' \text{ iff } M' \triangleq M[\alpha_k/\alpha'_k]$$

provided that $(s_i, o_j, \alpha_k) \in M$ and $(s_i, o_j, \alpha'_k) \in M'$. Moreover we define $\Rightarrow^c$ to be the smallest relation such that the following holds:

$$(s_i, o_j, \alpha_k) \Rightarrow^c (s_i, o_j, \alpha'_k) \text{ iff:}$$

1. $\alpha'_k \neq \emptyset$ when $\alpha_k \neq \emptyset$
2. $\alpha'_k \neq A$ when $\alpha_k = A$

Undesirable Configurations

In any system, there is a set of undesirable states that the system should not assume. One of the fundamental purposes of any security mechanism is to guarantee that for any possible transition (that originates in a safe/legal state) the target state will not be an illegal/undesirable state. An *undesired state* will be denoted by a given configuration/triple of subject, object and action. Therefore, let $U \subseteq S \times O \times A$ be the set of illegal configurations and let τ_u range over this set.

We want to avoid allowing a configuration change in which we enable an undesirable triple be part of the new version of the ACM. Hence, we want to avoid the following:

$$M \Rightarrow^{c} M' \text{ where } \tau_u \in M'$$

Secure Counterfactual Change

Our objective is to use *counterfactual logic* to let us decide whether or not a possible change to the current version of the ACM implies that at least one illegal triple is part of the future resulting version. Therefore our secure counterfactual implication can be expressed as:

$$[\bigwedge_{i,j,k \in \mathbb{N}} (s_i, o_j, \alpha_k)] \wedge (s_0, o_0, \alpha_0)[\alpha_0/\alpha_0'] \boxdot\!\!\rightarrow (\tau_u \notin M')$$

7.4.3 From SOLj to Access Control Matrix And Back

In Section 7.3 we have shown how a network of SOLj agents preserves a security policy expressed via the *assumptions* and *guarantees* that agents satisfy. However, the security policy might need to be modified in response to a changing requirement. Clearly, an arbitrary change may compromise the system's security. We have chosen an access control matrix-based model to express a system's security policy. We can derive the corresponding system's ACM by statically analyzing the SOLj modules. Then we use our counterfactual logic-based approach to determine whether or not a given change would compromise the ACM's integrity. If it is determined that the changes do not compromise security, they are applied to the ACM. The changed ACM can now be translated into a first order logic formula as shown in Section 7.4.2 and incorporated in the *assumptions* and *guarantees* section of the SOLj modules.

7.4.4 Kripke Version Model

In [29] the author provides the semantics of the counterfactual propositional logic using a possible worlds interpretation. In our case, each ACM version will represent a world. In the following definition, we write $t_i \leftarrow t_j$ to denote that the tuple t_i was swapped by tuple t_j. Below, we provide the formal semantics of the counterfactual implication based on a Kripke version model.

Definition 7.4.1 (Kripke Version Model) *A Kripke Version Model $\mathcal{R}$ is a triple $\langle \mathcal{M}, \Rightarrow, M_0 \rangle$ where:*

1. *$\mathcal{M} = \{M_k\}_{k \in \mathbb{N}}$ is the set of all access control matrix versions (ACM states).*
2. *M_0 is the initial access control matrix.*
3. *$\Rightarrow \subseteq \mathcal{M} \times \mathcal{M}$ is a binary relation defined the set of all possible ACM versions. Where $\Rightarrow$ is the smallest relation such that the following properties hold:*
   (a) *$t_i \leftarrow t_i$: tuple t_i is left unchanged. This stands for the do nothing transformation.*
   (b) *$t_i \leftarrow t_j$: tuple t_j replaces tuple t_i, where $t_j \in M_k$. We usually call this primitive transformation, a swap.*
   (c) *$t_i \leftarrow t_j$: tuple t_j replaces tuple t_i, where $t_j \triangleq t_i[\alpha'/\alpha]$.*

We assume that the relation $\Rightarrow$ complies with the properties of *reflexivity*, *symmetry*, and *transitivity*. Below, we justify each property based on the latter definition of $\Rightarrow$:

1. **Reflexivity:** For any ACM $M_i \in \mathcal{M}$, it is obvious that the *do-nothing transformation* will yield that any ACM can be transformed into itself. Therefore, $M_i \Rightarrow M_i$ given that for all $t_j \in M_i$, $M_i = M_i[t_j/t_j]$
2. **Symmetry:** For any ACMs $M_i, M_j \in \mathcal{M}$ any of the above transformations can be reversed and therefore, $M_i \Rightarrow M_j$ implies $M_j \Rightarrow M_i$.

3. **Transitivity:** For any ACMs $M_i, M_j, M_k \in \mathcal{M}$, applying two or more transformations to an ACM will yield intermediate versions; this is equivalent to transforming the initial version by composing the transformations into one. Thus, $M_i \Rightarrow M_j$ and $M_j \Rightarrow M_k$ imply that $M_i \Rightarrow^+ M_k$. Where $\Rightarrow^+$ denotes $\Rightarrow \circ \Rightarrow^{n-1}$ and $n > 1$.

7.4.5 Interpreting the Counterfactual Implication

Intuitive Interpretation

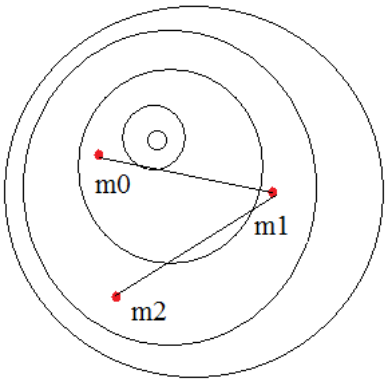

Figure 7.4: Lewis' concentric spheres diagram in the context of our formulas

Based on the *neighborhood Kripke* model [29] Lewis provided an interpretation of the counterfactual implication. A counterfactual implication is true if and only if the antecedent is true at some worlds and among these, the ones in which the consequent is true are closer to the actual world than those in which the consequent does not hold. Additionally, in [29] Lewis states that he does not attach any fixed interpretation to the notion of *closeness* or a *distance/metric* between the *worlds* in his model. We precisely establish the notion of *distance* used in this paper. Let M_1 and M_2 be the formal representations of two different versions as defined in section 7.4.2. We also identify each formula (i.e., Ψ_{M_1} and Ψ_{M_2}) with a specific *world* in our *Kripke Version Model*. Then we define the distance between Ψ_{M_1} and Ψ_{M_2} as the number of changes (as defined in section 7.4.4) between them. More precisely, if we were to identify each formula Ψ_{M_1} and Ψ_{M_2} with its corresponding string, then, the distance between them would be the *Damerau-Levenshtein distance* between their

corresponding strings. In Figure 7.4 we show that each world is identified with the formula Ψ_{M_i} that in turn corresponds to the ACM instance M_i.

Formal Interpretation

As it was stated earlier, the purpose of our model is to help interpret assertions in the language of counterfactual logic. Let M_0 denote our given initial ACM version. Also, let us assume we had a counterfactual assertion, namely $\phi \boxdot\!\!\rightarrow \psi$ in which:

- ϕ stands for assertions regarding M_0 and some transformation $t_i \leftarrow t_j$ that implies that $M_1 = M_0[t_i/t_j]$
- ψ stands for assertions regarding M_1

Thus, following the model-theoretic interpretation proposed by Lewis in [29], our version of the counterfactual implication is interpreted as:

$$\mathcal{R} \models \phi \boxdot\!\!\rightarrow \psi \ . \tag{7.1}$$

Where $\mathcal{R}$ denotes our previously defined *Kripke Version Model*. Moreover, letting α_i, β_i denote statements about the structure of M_i and M_j respectively, then, we can state that:

$$\phi \triangleq (\bigwedge_{i=1}^{n} \alpha_i) \wedge (M_i \Rightarrow^{+} M_j)$$

$$\psi \triangleq \bigwedge_{j=1}^{m} \beta_j$$

Where $\Rightarrow^{+}$ denotes the positive/transtive closure for the relation $\Rightarrow$. Furthermore, given an initial ACM version, namely M_0, we produce several versions by applying one or more transformations to it. In the context of a counterfactual assertion, the current version's structure and the changes applied to it (in order to produce a new version) imply properties possessed the new version and hence:

$$\mathcal{R} \models \phi \boxdot\!\!\rightarrow \psi \quad \triangleq \quad (\exists_{\min} k \in \mathbb{N})(\bigwedge_{i=1}^{n} \alpha_i) \wedge (M_0 \Rightarrow^k M') \rightarrow (\bigwedge_{j=1}^{m} \beta_j) \ .$$

The latter should be interpreted as there exists a minimal number of transformation steps such that given the properties of our initial ACM M_0 (namely, $\bigwedge_{i=1}^{n} \alpha_i$) and the transformation between the two versions implies the desired properties of the future ACM version (namely, $\bigwedge_{j=1}^{m} \beta_j$).

7.4.6 Fragment of Counterfactual Logic

Given that we are striving to provide a mechanical procedure and a tool whose purpose is to aid in reasoning about the properties that would hold for future versions of a given ACM, we define a logical calculus which will enable us to infer such properties in a algorithmic manner. Thus, we provide the set of inference rules taken from [29] and known as **VC** logic; these rules will help us formalize our approach's proof theoretical fragment. The soundness of these rules can be provided using a model-theoretic argument that we omit here.[2]

1. $\dfrac{\vdash \phi \rightarrow \psi, \phi}{\vdash \psi}$ Modus Ponens
2. $\dfrac{}{\phi \boxdot\!\!\rightarrow \phi}$ Reflexivity rule
3. $\dfrac{\phi \boxdot\!\!\rightarrow \psi}{\phi \rightarrow \psi}$ Counterfactual-Elimination rule
4. $\dfrac{\vdash \lambda \rightarrow \psi}{\vdash (\phi \boxdot\!\!\rightarrow \lambda) \rightarrow (\phi \boxdot\!\!\rightarrow \psi)}$ Deduction Within Conditionals
5. $\dfrac{\vdash (\phi \rightarrow \chi) \wedge (\chi \boxdot\!\!\rightarrow \psi)}{\vdash \phi \boxdot\!\!\rightarrow \psi}$ Partial Transitivity Theorem

7.5 Example - Security Proof Under Change Using Counterfactuals

First, we assume that all legal Access Control Matrices (ACM) are contained in the class $\mathcal{M}$. Furthermore, let us assume that $U \subseteq S \times O \times 2^A$ is the set of all illegal tuples. Additionally, we want

[2] In appendix 7.7 we give a model-theoretic proof for the Partial Transitivity Theorem.

to emphasize the difference between the set representation of a given ACM and its corresponding logical formula. The first is a semantic entity while the second is a syntactical entity. Our proofs proceed by manipulating the syntactical representation of an ACM. Below, we define the class $\mathcal{M}$ of all legal ACMs.

$$\mathcal{M} \triangleq \{M_i \mid M_i \subseteq S \times O \times 2^A \ , M_i \cap U = \emptyset\}$$

Let the ACM initial version be M_0, the next future version be M_1, and let M_3 be the next subsequent version. Without loss of generality we base our proof just on two subsequent changes. Moreover, in order to keep our proof short we will replace the following expressions:

$$(s_0, o_0, \alpha_0) \wedge (s_1, o_1, \alpha_0) \wedge, \ldots, \wedge (s_n, o_m, \alpha_l) \in M_i$$

by their proposed shorthand equivalent:

$$\bigwedge_{i,j,k} (s_i, o_j, \alpha_k)$$

In order to precisely define the change we need to give M_0's and M_1's set representation. In the following, $\alpha' \in 2^A$ is the new action to be incorporated to the tuple (s_0, o_0, α_0).

- $M_0 = \{(s_i, o_j, \alpha_k) \mid 0 \leq i \leq n, 0 \leq j \leq m, 0 \leq k \leq 4\}$
- $M_1 = (M_0 - \{(s_0, o_0, \alpha_0)\}) \cup \{(s_0, o_0, \alpha')\}$

Correspondingly, we also state the logic formula equivalents for M_0 and M_1.

- $\Psi_{M_0} \triangleq \bigwedge_{i,j,k} (s_i, o_j, \alpha_k)$
- $\Psi_{M_1} \triangleq \bigwedge_{i,j,k \neq 0} (s_i, o_j, \alpha_k) \wedge (s_0, o_0, \alpha_0)[\alpha'/\alpha_0]$

In order to flesh-out the proof, we need to agree on several facts. First of all, we can (by definition) assert that our initial ACM version M_0 is not compromised ($M_0 \cap U = \emptyset$).

Specifying Change

As it was defined before, a single change consists of renaming the third component on a given tuple. For readability purposes change will be expressed as a predicate.

$$\texttt{change}(\alpha', \alpha_k) \triangleq (s_i, o_j, \alpha_k)[\alpha'/\alpha_k]$$

Moreover, we shall use this predicate to express change being applied to the current ACM version as follows:

$$\Psi_{M_i} \wedge \texttt{change}(\alpha', \alpha) \; \Box\!\!\rightarrow \Psi_{M_j}$$

The latter implication should be interpreted as M_j being the resulting version if the change predicate had been applied to the previous ACM version.

The Proof

In this section we will use the counterfactual inference rules (given in section 7.4.6) to deduce whether or not a changed tuple implies a security breach i.e., an illegal tuple being included in the new version of ACM. Let us assume that M_1, M_2, M_3, and U be defined as before. Firstly, we need to take into account that $\Psi_{M_1} \wedge \texttt{change}(\alpha, \alpha') \; \Box\!\!\rightarrow \Psi_{M_2}$ and $\Psi_{M_2} \wedge \texttt{change}(\beta, \beta') \; \Box\!\!\rightarrow \Psi_{M_3}$ are model-theoretic facts. Each ACM version can be thought of as a world. Moreover, the formula corresponding to a given ACM version holds true at this world. Applying the change would imply that the formula corresponding to the next version is true.

We want to show that a sequence of changes may imply a security breach. Without loss of generality, let us assume that $(s', o', \alpha) \in M_0$ and $(s', o', \beta') \in U$. Furthermore, we will use the following change predicates in our proof:

$$\texttt{change}(\alpha, \alpha') \triangleq (s', o', \alpha)[\alpha'/\alpha]$$

$$\texttt{change}(\alpha', \beta') \triangleq (s', o', \alpha')[\beta'/\alpha']$$

Our main intention is to show that if a sequence of changes had been applied to the initial ACM version, then some security breach would have occurred. We may be tempted to allow only those counterfactual implications that denote safe changes as theorems of our logic.

Claim : $\mathbf{\Psi_{M_1}} \wedge \texttt{change}(\alpha, \alpha') \wedge \texttt{change}(\alpha', \beta) \boxdot\!\!\rightarrow (\mathbf{s}', \mathbf{o}', \beta') \in \mathbf{M_3}$

1. $\Psi_{M_1} \wedge \texttt{change}(\alpha, \alpha') \boxdot\!\!\rightarrow \Psi_{M_2}$ (Model-theoretic fact)
2. $\Psi_{M_1} \wedge \texttt{change}(\alpha, \alpha') \rightarrow \Psi_{M_2}$ ($\boxdot\!\!\rightarrow$-Elimination in 1)
3. $\Psi_{M_1} \wedge \texttt{change}(\alpha, \alpha') \wedge \texttt{change}(\alpha', \beta) \rightarrow \Psi_{M_2} \wedge \texttt{change}(\alpha', \beta)$ (Propositional Logic in 2)
4. $\Psi_{M_2} \wedge \texttt{change}(\alpha', \beta) \boxdot\!\!\rightarrow \Psi_{M_3}$ (Model-theoretic fact)
5. $\Psi_{M_1} \wedge \texttt{change}(\alpha, \alpha') \wedge \texttt{change}(\alpha', \beta) \boxdot\!\!\rightarrow \Psi_{M_3}$ (Partial Transitivity in 3 and 4)
6. $\Psi_{M_3} \rightarrow (s', o', \beta')$ (Propositional Logic)
7. $(\Psi_{M_1} \wedge \texttt{change}(\alpha, \alpha') \wedge \texttt{change}(\alpha', \beta) \boxdot\!\!\rightarrow \Psi_{M_3}) \rightarrow (\Psi_{M_1} \wedge \texttt{change}(\alpha, \alpha') \wedge \texttt{change}(\alpha', \beta) \boxdot\!\!\rightarrow (s', o', \beta'))$ (Deduction Within Conditionals in 6)
8. $\Psi_{M_1} \wedge \texttt{change}(\alpha, \alpha') \wedge \texttt{change}(\alpha', \beta) \boxdot\!\!\rightarrow (s', o', \beta')$ (Modus Ponens in 5 and 7) ■

By showing that $\Psi_{M_1} \wedge \texttt{change}(\alpha, \alpha') \wedge \texttt{change}(\alpha', \beta) \boxdot\!\!\rightarrow (s', o', \beta')$ is a theorem of our logic and considering that we asserted that $(s', o', \beta') \in U$ (i.e., (s', o', β') is an illegal tuple), we have caught a possible security breach that would have occurred.

Justification for Our Approach

The reader may argue that we could have applied the changes to the security policy and then used some form of static analysis to verify whether or not the new version suffers from a security breach. However, in the least risky scenario in which these verification efforts are done in a development

environment, the developers and IT professionals need to materialize a new version. In the most risky scenario, the changes are released to the production environment, possibly introducing a security breach in it. Therefore, it follows that our approach which only manipulates a representation of the security policy will not yield the scenarios that we just described.

7.6 Implementation Results

We have implemented the proof in Section 7.5 in Prolog. In the following we will define the predicates that realize the proof system.

matrix(Subject, Resource, Action) This predicate characterizes the tuples that comprise the system's legal configurations. This predicate has been declared as *dynamic* which implies that it can be changed during run-time.

prohibited(Subject, Resource, Action) This predicate denotes the system's illegal tuples.

violation(Subject, Resource, Action) This is an utility predicate that helps us check the consistency of the security model, namely, it checks whether or not $M_i \cap U = \emptyset$.

change(Tuple, NewAction) This predicate will change the current knowledge base as long as `Tuple` denotes a current legal tuple in the system.

boxArrow(Tuple1, Tuple2) This predicate will change the ACM's formal representation provided that `Tuple2` is not an illegal tuple.

transformACM(Tuple1, Tuple2) This predicate succeeds when the change from `Tuple1` to `Tuple2` implies that `Tuple2` is part of the current legal tuples.

chainTrans(Tuple1, Tuple2, Tuple3) This predicate succeeds when the transformation between `Tuple1` and `Tuple2` implies transformation between `Tuple2` and `Tuple3`.

We have modeled the security policies of the system shown in 7.3 as facts (knowledge base) against which the `chainTrans` goal (shown as the first line in Figure 7.5) is verified.

```
-chain_trans(matrix(a, resource1, r), matrix(a, resource1, rw), matrix(a, resource1, r)).
  Call: (7) chain_trans(matrix(a, resource1, r), matrix(a, resource1, rw), matrix(a, resource1, r)) ? :-
     Call: (8) transform_ACM(matrix(a, resource1, r), matrix(a, resource1, rw)) ? |
        Call: (9) box_arrow(matrix(a, resource1, r), matrix(a, resource1, rw)) ? |
        ^  Call: (10) not(prohibited(a, resource1, rw)) ? |
Call: (11) prohibited(a, resource1, rw) ? |
   Fail: (11) prohibited(a, resource1, rw) ? |
   ^  Exit: (10) not(prohibited(a, resource1, rw)) ? |
      Call: (10) change(matrix(a, resource1, r), rw) ? |skip
         Exit: (10) change(matrix(a, resource1, r), rw) ? |
 Exit: (9) box_arrow(matrix(a, resource1, r), matrix(a, resource1, rw)) ? |
    Call: (9) matrix(a, resource1, rw) ? |
       Exit: (9) matrix(a, resource1, rw) ? |
          Exit: (8) transform_ACM(matrix(a, resource1, r), matrix(a, resource1, rw)) ? |
             Call: (8) transform_ACM(matrix(a, resource1, rw), matrix(a, resource1, r)) ? |
Call: (9) box_arrow(matrix(a, resource1, rw), matrix(a, resource1, r)) ? |
^  Call: (10) not(prohibited(a, resource1, r)) ? |
   Call: (11) prohibited(a, resource1, r) ? |
      Fail: (11) prohibited(a, resource1, r) ? |
      ^  Exit: (10) not(prohibited(a, resource1, r)) ? |
         Call: (10) change(matrix(a, resource1, rw), r) ? |skip
Exit: (10) change(matrix(a, resource1, rw), r) ? |
   Exit: (9) box_arrow(matrix(a, resource1, rw), matrix(a, resource1, r)) ? |
      Call: (9) matrix(a, resource1, r) ? |
         Exit: (9) matrix(a, resource1, r) ? |
            Exit: (8) transform_ACM(matrix(a, resource1, rw), matrix(a, resource1, r)) ? |
               Exit: (7) chain_trans(matrix(a, resource1, r), matrix(a, resource1, rw), matrix(a, resource1, r)) ?
               Yes
```

Figure 7.5: Prolog run for the `chainTrans` predicate

Figure 7.5 shows the proof of the change that consists in initially agent `a` being given access to `read` and `write resource1` and subsequently agent `a` being given access to just `read resource1`. We succeed in proving the correctness of the change; the two target tuples are not illegal tuples. We have trimmed the call-stack by skipping the execution of the `change` predicate which in turn just calls system predicates `retract` and `assert`. In addition, we verified the impact of the following changes to security policies in the system shown in Figure 7.3.

chainTrans(matrix(b, resource3, rw), matrix(b, resource3, w), matrix(b, resource3, null)) Agent `b` is currently able to `read` and `write resource3`, we remove its right to `read resource3`, and finally we remove all rights to access `resource3`.

chainTrans(matrix(c, resource2, w), matrix(c, resource2, null), matrix(c, resource2, rw)) Agent `c` is currently allowed to `write resource2`, we remove all access to this resource, and finally we grant agent `c` access to `read` and `write resource2`.

chainTrans(matrix(a, resource4, null), matrix(a, resource4, w), matrix(a, resource4, rw)) Agent `a` is currently not allowed access to `resource4`, we give it access to `write` this resource, and finally we provide agent `a` rights to `read` and write `resource4`.

chainTrans(matrix(c, resource1, rw), matrix(c, resource1, r), matrix(c, resource1, w)) Agent `c` is currently able to `read` and `write` `resource1`, we remove its access to `read` this resource, and finally we grant agent `c` access to just `write` `resource1`.

All experiments were done on a dual core Dell Precision T3500 desktop. For the execution depicted in Figure 7.5 and the executions of the latter four goals the time needed was negligible. As stated in section 7.4.5, our formulas can be regarded as strings. Hence the proofs (i.e., goal traces) are strings as well and we can quantify their length using character count (not taking blank spaces into account). In Table 8.1 we show the length of the trace for each of the goals listed above.

Prolog Goal	**Proof Length**
Goal trace 1	3,165
Goal trace 2	3,274
Goal trace 3	3,227
Goal trace 4	3,199

Table 7.1: Execution lengths for the example goals

7.7 Model-Theoretic Justification for The Partial Transitivity Theorem

As it is stated in [29] the **VC**-Logic can be interpreted using models of index sets (these models are a formalized version of Lewis' model of concentric spheres). In order to establish a complete context we re-state the *Partial Transitivity Theorem* (PTT) below.

$$\frac{\vdash (\phi \rightarrow \chi) \wedge (\chi \boxdot\!\!\rightarrow \psi)}{\vdash \phi \boxdot\!\!\rightarrow \psi}$$

Since the **VC**-Logic is sound, the latter statement must hold in all models. In order to derive a contradiction, let us assume that there is a model in which the PTT does not hold. Let this model be denoted by:

$$\mathcal{S} = \{S_i \mid i \in I\}$$

Furthermore, as it is defined in [29], we identify $\mathcal{L}_c$ with the language corresponding to **VC**-Logic and $Snt(\mathcal{L}_c)$ with the set of sentences of this language. We let I be the index set of maximally consistent sentences. Also, let $\mathcal{I} : Snt(\mathcal{L}_c) \to 2^I$ be the interpretation function.

As it is stated in [29], we only require $\mathcal{S}$ to comply with the *centered* property, which amounts to $\bigcap_{i\in I} S_i = \{i_0\}$ for some index $i_0 \in I$. Additionally, since $\mathcal{S}$ is closed under arbitrary unions we can state $S^* = \bigcup_{i\in I} S_i$. We provide the interpretation of the sentences that comprise the PTT below:

1. $\mathcal{I}(\phi \to \chi) = (I - \mathcal{I}(\phi)) \cup \mathcal{I}(\chi)$
2. $\mathcal{I}(\chi \boxdot\!\!\!\rightarrow \psi) = \{i \in I \mid \mathcal{I}(\chi) \cap S^* \neq \emptyset \to \exists S \in \mathcal{S}(\mathcal{I}(\chi) \cap S \subset \mathcal{I}(\psi))\}$
3. $\mathcal{I}(\phi \boxdot\!\!\!\rightarrow \psi) = \{i \in I \mid \mathcal{I}(\phi) \cap S^* \neq \emptyset \to \exists S \in \mathcal{S}(\mathcal{I}(\phi) \cap S \subset \mathcal{I}(\psi))\}$

Based on the non-vacuous interpretation of $\phi \to \chi$ we can infer that $\mathcal{I}(\phi) \subset \mathcal{I}(\chi)$. Correspondingly, a non-trivial interpretation of $\chi \boxdot\!\!\!\rightarrow \psi$ requires us to accept $\mathcal{I}(\chi) \cap S^* \neq \emptyset$ as true. For the sake of deriving a contradiction, let us assume that $\phi \boxdot\!\!\!\rightarrow \psi$ is false and hence its interpretation is the empty set. Thus, the following statements hold:

- $\mathcal{I}(\phi) \cap S^* \neq \emptyset$
- $\forall S \in \mathcal{S}(\mathcal{I}(\phi) \cap S \not\subset \mathcal{I}(\psi))$

Thus, we must accept that for all S in $\mathcal{S}$, $\mathcal{I}(\phi) \cap S \not\subset \mathcal{I}(\psi)$. Notice also that by assuming $\mathcal{I}(\chi \boxdot\!\!\!\rightarrow \psi)$ as non-trivially valid and having assumed that $\mathcal{I}(\chi) \cap S^* \neq \emptyset$ then we must also accept

that there exists some $S_0 \in \mathcal{S}$ such that $\mathcal{I}(\chi) \cap S_0 \subset \mathcal{I}(\psi)$. However, we had assumed that $\mathcal{I}(\phi) \subset \mathcal{I}(\chi)$ and thus $\mathcal{I}(\phi) \cap S \not\subset \mathcal{I}(\psi)$ for all S in $\mathcal{S}$ cannot hold since $\mathcal{I}(\phi) \cap S_0 \subset \mathcal{I}(\chi) \cap S_0 \subset \mathcal{I}(\psi)$. Therefore, we have reached a contradiction and such model $\mathcal{S}$ cannot exist. Thus, the PTT must hold in all models of **VC**-Logic. ■

7.8 Conclusion

We have introduced a framework for formally addressing security concerns in sensor networks. The SOLj-based framework allows us to use a model-driven approach wherein agents are specified and verified at a high level and then compiled to JAVA code that can run on Sun SPOT™ or Android™ platforms. Also, we have demonstrated that our counterfactual-based framework can be used to formally represent changes to SNSs and perform what-if reasoning about their impact on security and safety even before they have been applied.

Chapter 8

Implementation Results

8.1 Introduction

In this chapter we show how the theoretical claims we have made thus far can be effectively realized by our implementation results. Since our techniques were mainly proof-theoretical in nature, our implementation relies heavily on theorem proving. We show that what was presented as manual proofs in *Counterfactual Logic* can be represented as programs in Prolog. We show that source code and security policies can be represented as asserted terms in a Prolog program. Furthermore, we show that in both cases: source-code and security policies, simple properties (e.g, variable bindings and legal-tuples) are preserved in two scenarios: the manual proof-theoretical derivations and the implemented Prolog goals.

Many formal verification efforts provide manual proofs that give some idea about the proposed approach's correctness. It is not different in our case, since we have already provided proofs that show how our proposed counterfactual approach can be applied to source code. However, programmers and software engineers may lack the skill-set and resources (e.g., time, personnel, etc.) to manually derive proofs that assert that their code is relatively correct in the face of change. Therefore, we must provide implementation results and tools that deem our approach as effectively feasible, i.e., it can be successfully realized by means of theorem proving.

We have implemented a proof procedure that guarantees the relative correctness of source code using Prolog. We capitalized the strong similarity relationship between context-free grammar production rules and Horn clauses. The latter relationship allows us to parse our source code also using Prolog. In essence, the parsing phase allows us to transform the source code into Prolog terms and assert them as facts. Additionally, we are able to express the source-code changes (e.g., statement swaps) in Prolog by manipulating the knowledge base of program terms. Additionally,

we are able to express the properties that need to be upheld by the source code as Prolog terms. Using all these elements we have reduced our manual proofs to Prolog resolution; i.e., given a source-code fragment, a set of properties that the need to be preserved and a given change to the source code, we can decide whether or not the change respects the properties after the change has been applied. The latter implies that we can decide whether or not a given source code-change will result in undesired secondary effects without changing the source-code.

8.2 Programming as Theorem Proving

According to the literature the *Program as Proofs* approach is not a far fetched idea. As it was stated in [27], a given problem domain can be axiomatized (i.e., be represented as rules and facts) and a given problem may be represented as a theorem (i.e., a query) that will be derived from the proposed axiomatization. In this scenario, the syntax is relatively simple. Programs are composed of *sentences* in *clausal* form as shown below:

$$Q_1, \ldots, Q_n \leftarrow P_1, \ldots P_m$$

Each Q_i and P_j is considered as an atomic formula whose syntax follows that of a *First Order Predicate logic* atomic formula and we shall not cover here for the sake of brevity. The semantics for this approach are given by two well known techniques: *Resolution* and *Unification*. Furthermore, our approach requires us to transform the provided source-code fragment into logical formulas. This is done entirely via Prolog using its *Definite Clause Grammar* feature which implements *parsing by deduction*-approach presented in [37].

8.3 Implementation Roadmap

This section deals to with the task of using the *Program as Theorem Proving* approach to provide an implementation for the code-change impact analysis method shown in section 4.5.

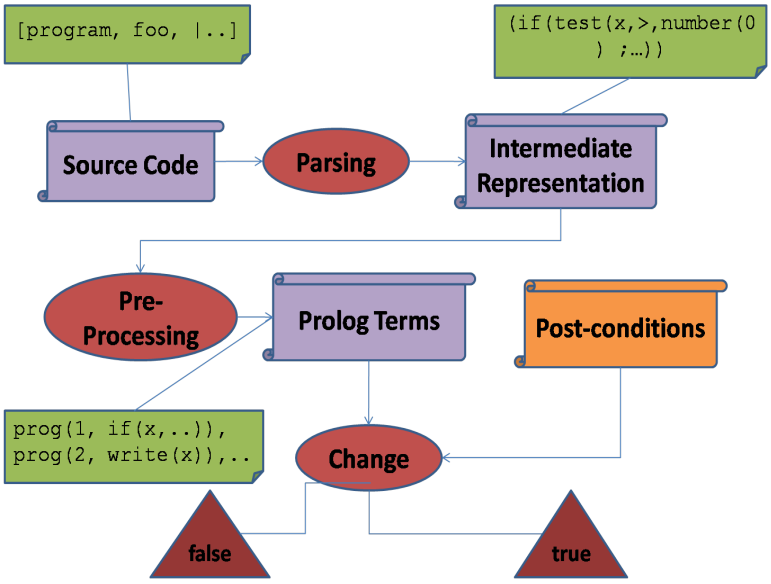

Figure 8.1: System diagram for the code-change impact analysis implementation

The diagram in Figure 8.1 illustrates the road-map we followed to automatically verify that a changed code fragment respects a set of future properties. In other words, we are able to verify that the counterfactual implication between the change in the code (represented as a logical formula) and the desired properties of the change is a theorem in our logic. Below we enumerate and explain the steps depicted in the diagram:

Parsing The input to this step is a token list that denotes the program source-code. We use Prolog's inherent *Definite Clause Grammar* (DCG) capability to parse the list and produce a suitable intermediate representation.

Pre-Processing The intermediate representation is then enriched with information (e.g. line-numbers, operators, and arguments) to produce the *Prolog-terms* which denote the logical representation of the original source-code submitted in the *Parsing* step. The program terms are asserted as facts in the Prolog run-time knowledge base.

Change The input for this step is comprised by the logical representation for the source-code (derived in the latter step) and the properties that should be respected by the change (e.g.,

variables bindings). More specifically, the change is represented by a *swap* or transposition between two program-terms.

8.4 The Parser

Due to Prolog's inherent DCG feature we can provide the grammar for our source language as input to the Prolog compiler. The compiler transforms the DCG rules into standard Horn clauses. Therefore, parsing is reduced to normal Prolog resolution. Parsing as resolution, a form of theorem proving, was a technique first shown in [37]. This form of parsing is possible due to the use of First Order Logic *definite clauses* to axiomatize *context free grammars* [37].

```
%Entry point for the parsing phase
parse(Source, Structure) :- alProgram(Structure, Source, []).

% Statement parsing rules
alProgram(S) -->
[program], identifier(X), [';'], statement(S).
statement((S;Ss)) -->
    [begin], statement(S), restStatement(Ss).
statement(assign(L,X,V)) -->
    [L], identifier(X), [':='], expression(V).
statement(if(L, T,S1,S2)) -->
    [L], [if], test(T), [then], statement(S1),
    [else], statement(S2), {integer(L)}.
statement(while(L, T,S)) -->
    [L], [while], test(T), [do], statement(S).
statement(read(L,X))  -->
    [L], [read], identifier(X), {integer(L)}.
statement(write(L, X)) -->
    [L], [write], expression(X), {integer(L)}.
restStatement((S;Ss)) -->
    [';'], statement(S), restStatement(Ss).
restStatement(void) --> [end].
```

Figure 8.2: First fragment of the DCG grammar

As it was stated earlier, we implemented the parsing effort using Prolog and its inherent DCG feature. In Figure 8.2 we show a fragment of the DCG grammar. We can implement parsing via

```
alProgram(A, B, C) :-
    'C'(B, program, D),
identifier(E, D, F),
'C'(F, (;), G),
statement(A, G, C).

statement(if(A, B, C, D), E, F) :-
    'C'(E, A, G),
    'C'(G, if, H),
    test(B, H, I),
    'C'(I, then, J),
statement(C, J, K),
    'C'(K, else, L),
    statement(D, L, M),
    integer(A),
    F=M.

statement(while(A, B, C), D, E) :-
    'C'(D, A, F),
    'C'(F, while, G),
    test(B, G, H),
'C'(H, do, I),
statement(C, I, J),
integer(A),
E=J.
```

Figure 8.3: Prolog clauses corresponding to the DCG rules

resolution due to the fact (first stated in [36]) that parsing is just a *constructive* way of proving that a given string belongs to certain *context-free* language.

We surely could have used any other language/software for parsing purposes, however, it is undeniably more consistent if the whole effort was executed using Prolog-resolution. For example, as it will be shown in section 8.8, for metric and benchmark purposes we only need to to look into the depth of the search tree obtained by chaining the execution of the parsing source-fragment and the counterfactual-proof source-fragment. In Figure 8.3 we can see how the compiler produced standard Prolog source using the DCG specification shown in Figure 8.2.

8.5 Intermediate Code Representation

The parser takes a list of tokens as inputs and produces the intermediate code representation we use in later phases of the implementation process.

```
program example;
    begin
    1    x := 2;
    2    if x > 0 then begin
    3        y := 0;
    4        write y
         end
         else begin
    5        y := 1;
    6        write y;
         end
    end
```

Figure 8.4: Example program

The DCG parser specified in section 8.4 takes the program shown in Figure 8.4 and generates the following output structure (i.e, S).

```
S = assign(1, x, number(2));
    if(2, compare(>, name(x), number(0)),
    (assign(3, y, number(0));
    write(4, name(y));void),
    (assign(5, y, number(1));
    write(6, name(y));void));void
```

Figure 8.5: Result of the parsing phase

The structure S shown in Figure 8.5 stands for the parse tree generated from the code shown in Figure 8.4. The structure can be readily identified with a parse tree due to its nested structure.

The tree depicted in Figure 8.6 shows the explicit structure of S (shown in Figure 8.5). The rest of our Prolog verification effort uses S as input.

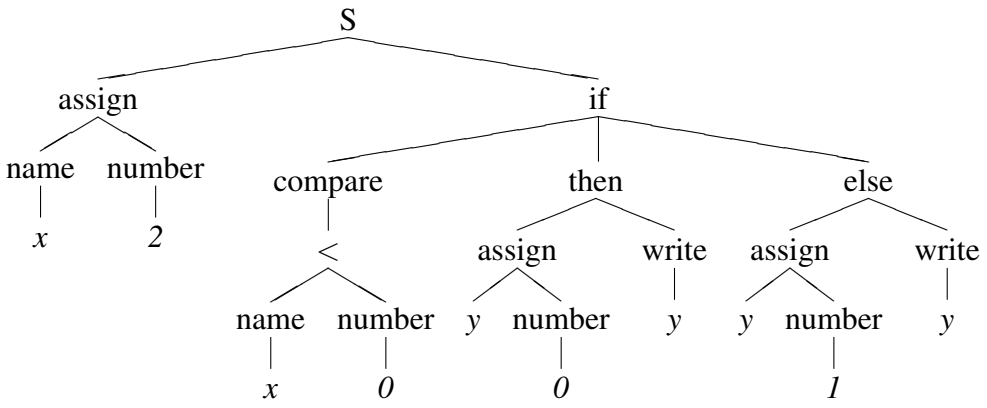

Figure 8.6: Parse tree that corresponds to the structure S

8.6 Program Terms as Facts

If we look at S as it is depicted in Figure 8.5, we can see it is composed of terms such as:

- `assign(n, x, 5)`,
- `compare(m, <, name(x), number(2))`,
- `write(s, name(x))`

In the parsing phase each term (e.g., `assign`, `compare`, `write`) denotes the parsed pseudo-code statement. In the subsequent phases each of these terms will be viewed as an *un-interpreted function symbol*. This allows us to readily interpret each program statement as a term in Prolog's run-time knowledge base.

We transform S into program terms using the code shown in Figure 8.7. The `stmtSplit` predicate takes S, a nested list of atoms, and produces a list of program terms. The resulting program terms have the form `f(n, <statement>)`, e.g., `write(n, name(x))` where n denotes the line-number in which this statement happens in the source code.

The code shown in Figure 8.8 takes a list of *program terms* (which should be already asserted in the knowledge base) as arguments and produces the program properties relevant to the source-code statement that the program term represents. For example, the program term `assign(5,`

```
stmtSplit((Stmt;Stmts),[Stmt1|Stmts1]) :-
    stmtSplit(Stmt, Stmt1),
    stmtSplit(Stmts, Stmts1).
stmtSplit(if(X, Y, (Z;Zs), (W;Ws)),
    [f(X, if(Y))|Stmts]) :-
    stmtSplit((Z;Zs), Stmts1),
    stmtSplit((W;Ws), Stmts2),
    append(Stmts1, Stmts2, Stmts).
stmtSplit(read(X, Y), f(X, read(Y))).
stmtSplit(write(X, Y), f(X, write(Y))).
stmtSplit(assign(X, Y, Z), f(X, assign(Y, Z))).
stmtSplit(void, []).
```

Figure 8.7: Prolog code that transforms S into program terms

`y, number(2))` shown in Figure 8.5 produces the terms `bind(y, 1)` and asserts it as part of the knowledge base.

8.7 Verifying The Change

We implement code changes or transformation by performing *swaps* between two given program terms contained in the knowledge base. The later is done via *pattern matching*; each program term has a corresponding line-number, the code-change is performed by swapping the line numbers in two given program terms.

The code fragment shown in Figure 8.9 uses the *retract* and *assert* pre-defined Prolog predicates to remove the old program terms from the knowledge base and add the new ones. The new state of the knowledge base reflects the changed program source code. Now, we are in position to verify whether or not this new version of the knowledge base is consistent with a desired future property.

We will use the source-code fragment shown in Figure 8.4 to illustrate how our Prolog verification code treats a simple example. The program in Figure 8.4 is mainly composed of an *if-then-else*-block. The value to which the variable `y` is bound (i.e., 0 or 1) depends on the value of

```
assertProgFacts([ProgTerm|ProgTerms]) :-
    progCond(ProgTerm),
    assertProgFacts(ProgTerms).
assertProgFacts([]) :- true.

progCond(f(_, assign(Var, number(N)))) :-
    Val = N, assert(bind(Var, Val)).
progCond(f(_,
    if(compare(>, name(Var), number(N))))) :-
    bind(Var, Val),
    Val > N,
    assert(isGreater(Var, N)).

progCond(f(_, write(name(Var)))) :-
    bind(Var,_), assert(isOutput(Var)).
```

Figure 8.8: Prolog code fragment that extracts the properties of program-terms

```
programSwap(Line_i, Line_j) :-
    retract(f(Line_i, Stmt_i)),
retract(f(Line_j, Stmt_j)),
assert(f(Line_i, Stmt_j)),
assert(f(Line_j, Stmt_i)).
```

Figure 8.9: Prolog source-code fragment that implements the swap transformation

the input variable, `x`. If we had swapped the third and fifth statements, the variable `y` would have been bound to an alternative value, namely 1.

In Figure 8.10 we show the Prolog code that connects all the elements we have explained so far. The `progChange` predicate takes three variables as parameters; the first parameter, `L`, is a list of program terms; the second and third parameters are integers that denote the lines that will be swapped; the fourth and fifth parameters denote the variable and the value to which it would be bound if the swap occurred. The first three predicates remove all program-facts from the run-time knowledge base in case the `progChange` predicate had been run before. Then the `progAssert` and `assertProgFacts` respectively integrate the program terms (e.g., `f(2, assign(name(x), number(2)))`) and program properties (e.g., `bind(x,2)`) to the run-time knowledge base. As explained earlier, the predicate `programSwap(Line1, Line2)`

```
progChange(L, Line_1, Line_2, Var, Val) :-
    retractall(f/2),
    retractall(bind/2),
    retractall(isGreater/2),
    progAssert(L),
    assertProgFacts(L),
    programSwap(Line_1, Line_2),
    isGreater(x,0) -> bind(Var, Val).
```

Figure 8.10: Prolog code that verifies a binding-variable change

changes the program terms in the knowledge base by swapping the line numbers corresponding to the terms. In this instance, we want to find out the binding for variable `y` if the we had swapped the two statements. Therefore, we need add the predicate `isGreater(x,0) bind(Var, Val)` to the set of goals we want to resolve.

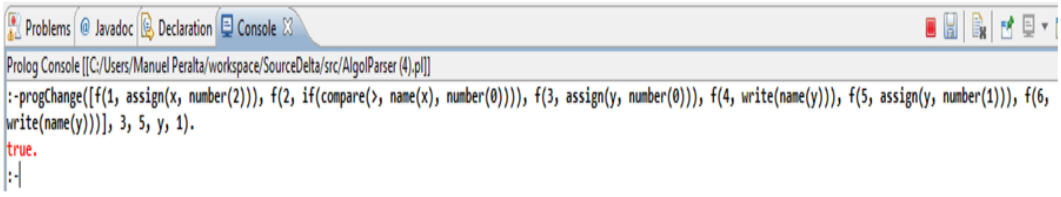

Figure 8.11: Result for the resolution of the verification goals

The positively resolved goal shown in Figure 8.11 indicates that if we had swapped the third and fifth lines, the variable `y` would have been bound to the value `1`. We have to take into account that this verification effort has not modified the source-code per se. What we have changed is the `program terms` that logically denote the source code in Figure 8.4. The program terms are indeed a simpler representation of a program if we compared them to their corresponding source code. Changing the `program-term` representation is far less complex than changing the source-code.

The result depicted in Figure 8.12 shows how the change has affected the knowledge base. The new version of the set of program terms complies with the proposed variable binding (i.e., a desired future property) as it was shown in Figure 8.11. The results shown in this section prove

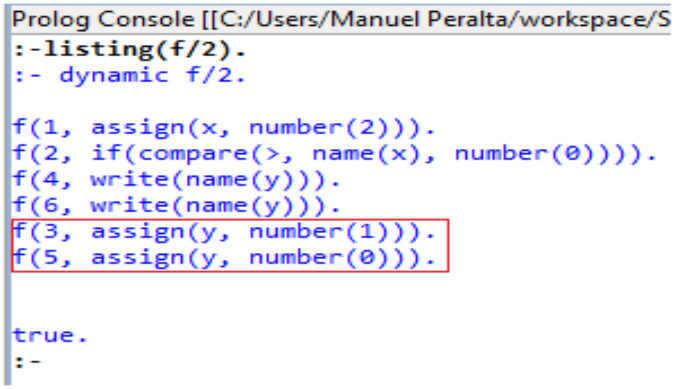

Figure 8.12: Changed program terms in the knowledge base

that we can fully implement the manual proof shown in section 4.5 via Prolog resultion. In both theoretical and implementation scenarios we have steered away from modifying the code. In both cases we have found a suitable and simpler source-code representation that aids us in reasoning about code structure, its meaning and possible changes to it.

8.8 Benchmarks

In this section we show how our implementation behaves in terms of several examples. This section includes a set of benchmarks that will help us judge the relative efficiency of our implementation approach. We have decided to use two metrics for our benchmarks:

- Proof length (in characters)
- Proof-tree depth (in call-stack depth)

The use of these metrics is heavily justified by the arguments made in [48] and, more in depth, in [45]. The use of the first metric, *proof length* is related to the *meaning* of a Prolog program. In simple terms the meaning of a program is the set of instantiated or ground-terms related to the program. The proof of the program ranges over the meanings of it and thus, the size of the set of ground-terms is proportional to the length of the proof. Proof-length is thus an adequate

measure of the effort made to resolve the initial goal. Alternatively, we use the depth of the call stack to determine the depth of the proof tree. Using the *trace* predicate we can follow how the run-time tries to resolve a goal. The traced execution of a goal resolution is equivalent to a proof tree; each step of the trace shows the selected goal and the variable instantiations/binding used to resolve the goal. Hence, the maximum depth number reached by the call-stack at run-time accurately describes the depth of a successful goal search.[1] Below we provide the source code for the different program-change predicates use to obtain the benchmark data.

```
 progChange(L, Line_1, Line_2, TestVar, TestValue, OUVar) :-
clearKnowledgeBase,
 progAssert(L),
 assertProgFacts(L),
 programSwap(Line_1, Line_2),
 isGreater(TestVar,TestValue) -> isOutput(OUVar).
```

Figure 8.13: First program-change predicate

The code in Figure 8.13 takes six variables as input. After asserting the set of program-terms contained in `L`, still respects the implication between the `isGreater` predicate and the `isOutput` predicate. The `programSwap` predicate works as explained in Section 8.6.

The code fragment shown in Figure 8.14 acts on a set of program terms comprised mostly on assignment statements (or the set of assignment statements of a more general program). The set of `bind` predicates denotes the variable bindings the source-transformation affects. The pre-condition to this change is that the bounded values respect some ordering. The `threeWayScramble` predicate perform a permutation between the program terms labeled by `Line1`, `Line2`, and `Line3`. After the program-terms are modified the values are compared to verify whether or not the ordering is still respected.

The code fragment shown in 8.15 takes a list of program terms, `L` and, `OperList`, a list of comparison operators. After the program terms and program properties are asserted, we use the

[1] All benchmarks were done on a dual-core Dell Precision T3500 dual-core desktop.

```
 progChange2(L, Line_1, Line_2, Line_3) :-
clearKnowledgeBase,
progAssert(L),
assertProgFacts(L),
(bind(Line_1, x, Val1),
 bind(Line_2, y, Val2),
 bind(Line_3, z, Val3)),
Val1 < Val2, Val2 < Val3,
threeWayScramble(Line_1, Line_2, Line_3),
findall(f(N,S), f(N,S), L2),
assertProgFacts(L2),
(bind(Line_1, x, Val1),
 bind(Line_2, y, Val2),
 bind(Line_3, z, Val3)),
(Val1 =\= Val2), (Val2 =\= Val3).
```

Figure 8.14: Second program-transform predicate

```
 progChange3(L, OperList) :-
             clearKnowledgeBase,
progAssert(L),
assertProgFacts(L),
getAllIfTerms(IfList),
changeIfStmts(IfList, OperList),
findall(f(N,S), f(N,S), L2),
assertProgFacts(L2),
verifyIfConds(OperList).
```

Figure 8.15: Third program-transform predicate

`getAllIfTerms` to collect all the *if-program-terms* from the knowledge base. The predicate `changeIfStmts` applies the new operators (contained in `OperList`) to the list of if-program-terms. We modify the knowledge-base (program terms and property terms) via the `findall` and `assertProgFacts` predicates. Finally, the `verifyConds` predicate which correlates the new comparison operators used in the `if-program-statements` with the new property terms that were incorporated in the program's knowledge base.

Table 8.1 shows the results of running several successful queries against a set of test programs. We can see that the all three predicates are progressively more complex (they involve more inter-

Prolog Goal	Proof Length	Proof Depth	Applied Predicate
Benchmark 1	18,875	15	`progChange`
Benchmark 2	13,783	15	`progChange`
Benchmark 3	12,369	14	`progChange`
Benchmark 5	215,648	36	`progChange2`
Benchmark 6	80,703	18	`progChange2`
Benchmark 7	99,178	18	`progChange`
Benchmark 8	173,073	21	`progChange3`
Benchmark 9	60,407	16	`progChange3`
Benchmark 10	27,805	15	`progChange3`

Table 8.1: Benchmark table for five different program-transformation queries

mediate predicates and modify the knowledge-base more than one time), therefore, for example `progChange2` will yield larger *proof length* counts than `progChange` would for the same test program. We derived the *proof-length* and *proof-depth* metrics using Prolog's `trace` predicate which executes a query in a one-step-at-a-time manner. At each step the run-time system shows which goal has been selected by the resolution process and the set of variable instantiations that are relevant to that step. We consider each step as a level of the proof-tree. The step count for the query's trace is identified with the *proof-depth* metric. Additionally, we consider the character count as the length of the trace. The trace's length is identified with the *proof-length* metric. There is a definite positive correlation between the length and depth of different benchmarks. We can see that certain cases (i.e., benchmarks 5 and 8) the proof lengths seem to be outlying all other benchmarks. Correspondingly, the depths associated with these benchmarks are also the largest amongst all other benchmarks.

It is widely known that Prolog resolution (i.e., *SLDNF resolution*) has a worst time complexity of $O(n^3)$. We assume that the unification that is done within the resolution task incorporates the *occurs-check*) where n is the bit-length of the goal to be resolved. Such time complexity is usually expected for theorem-proving algorithms. Although such time-complexity may yield unfeasible

resolution times for programs that are exceedingly large, the *proof-length* and *proof-depth* metrics can be seen as more descriptive of the size and computational effort that Prolog resolution requires.

Chapter 9

Concluding Remarks and Future Work

The work presented in this dissertation dealt mainly with the problem of assessing the impact of a given source code fragment in different contexts. The logic of counterfactuals has been a commonly used tool throughout all these contexts (sequential programming, general security, sensor network systems, and security within sensor network systems). Although counterfactual logic may be regarded as a non-traditional tool in the realm of software verification we have found it ideal for the endeavor of performing source-code change impact analysis in an *a priori* manner. Given that our approach is a relatively novel way of formally expressing change-impact concerns, we needed a new kind of logic which enough expressive power to do the latter. The work presented in this dissertation suggests that there is a new way by which change-impact analysis can be performed. This new way of evaluating change is characterized by bypassing any modification to a source code fragment, expressing the change in a formal manner, formalizing the desired properties, expressing the source structure as a logical formula, and determining whether or not the formalized changes follow from the desired future properties and the source-code structure.

In Chapter 4 we have applied our counterfactual logic approach to a sequential programming scenario. We express the source code structure as a formula in our logic. Subsequently, we formalize the source-code modification as a *program transformer*, i.e., a second-order logical formula which joins the source-code formula and the change formula and produces the formula corresponding to the new version of the program. Using two examples we were able to successfully prove that the desired changed formal versions of two programs were theorems of our logic.

In Chapter 5 we adapted our counterfactual logic approach to a general security setting. We took advantage of the up-front simplicity of the Access Control Matrix (ACM) security model i.e., the formulas that encode the ACM are simply statements about set membership. We formalized

a given ACM instance and expressed it as a formula. In this instance, change was defined as a modification in at least one of the tuples that pertain to the ACM. Furthermore, we model the set of illegal configurations as a specific set of tuples. A coherent change is defined as one that does not produce an illegal ACM i.e., one that contains at least one illegal tuple.

In Chapter 6 we apply our counterfactual logic approach to sensor networks. Due to the reactive nature of sensor network systems we had to augment our logic with temporal logic operators and temporal logic inference rules. In this chapter we were able to successfully formalize the structure of the source code that denotes the parallel composition of a set of [SOLj] agents, express the behavior of this set of agents as temporal logic formula, and successfully show that for a given example system changing one of the code modules (agent code) would eventually lead to an alternative desired behavior (also expressed in temporal logic).

In Chapter 7 we have furthered the work presented in Chapter 5 by using our counterfactual logic to address security concerns within sensor network systems. We first show the inherent security characteristics that pertain to sensor network systems comprised of SOLj agents. We do this by showing that SOLj agents behave as *security automata* and in this case security policies are expressed as *transition invariants*. Furthermore, we provided a concrete example in which a set of resources must be used in a secure (i.e., an agent can use a resource provided that it has been given permission to use it via the *access control list*) and safe (i.e., a given resource can only be acquired once by an authorized agent and can only be successfully requested once it has been released from its previous user) manner. We were able to prove that our counterfactual logic approach has enough expressive power to express the security policy of the example system. Furthermore, we show that a given change in our system will yield an illegal operating configuration. Therefore, we were able to catch a security breach even before the change was in fact incorporated into the system. The latter is very valuable in scenarios as this one were introduction of secondary ill-defects in live-production environment will mostly lead to relatively large financial losses and even loss of life.

In Chapter 8 we showed how our theoretical claims made in previous chapters could be implemented via logic programming. We chose Prolog as a tool due to its widespread use on the realms of automatic theorem proving. Regarding change-impact-analysis for source code, we developed a technique by which source code is simplified via a parsing technique which used Prolog's *Definite Clause Grammar* feature. The input source-code was transformed in Prolog terms. These terms comprised the knowledge-base which encoded the source code's structure. Additionally, we use the program terms to deduce simple program properties (e.g., variable bindings, if-conditions, etc.). These program properties were also encoded as Prolog terms and asserted and complemented the knowledge base. Moreover, we modeled a source code transformation based on swapping line numbers among two or more program terms. Based on the latter we produced four program-transform predicates. These predicates encoded the counterfactual changes we theoretically proved in earlier chapters. These change predicates were used as goals which always succeeded when the proposed change did not conflict with the properties that needed to be preserved and/or incorporated. In summary, we were able to implement our theoretical proof strategies using Prolog resolution for parsing the input source code and deducing whether or not the source-code-transformation was correct.

In terms of future work we have recently discovered a very strong (and very promising) relationship between fragments of *Monadic Second-Order Logic* (MSOL) and automata theory. It is already known that MSOL has enough expressive power to encode finite automata .We are already capable of expressing a program's source code as a second-order logic formulas. Furthermore, we can identify the program source code with a string. This string along with some the usual interpreted function and relation symbols comprises the *word model* for the program's MSOL formula. Moreover, using our second-order logic *program transformer notion* we can produce a formula for a second version of the program. The literature establishes that one can produce an automaton using the string's *word model*. We conjecture that the transitions on this automaton may be *decorated* with properties that the program and/or it's second version should satisfy. Furthermore, it would

be interesting to determine whether or not the MSOL formulas that denote the first and second version's of the program can be satisfied by the same *word model* and hence the same automaton. If the latter conjecture were to be achievable we could reduce the verifying whether or not a program complies with a set of properties to the *language-emptiness checking problem* which is known to be decidable.

[1] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: towards an environmental computing paradigm for distributed sensor networks. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 582–589, 2004.

[2] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods*, pages 1–20. 2004.

[3] T. Batista and N. Rodriguez. Dynamic reconfiguration of component-based applications. In *Software Engineering for Parallel and Distributed Systems, 2000. Proceedings. International Symposium on*, pages 32–39. IEEE, 2000.

[4] Boris Beizer. *Software Testing Techniques, 2nd Edition*. International Thomson Computer Press, 2 sub edition, June 1990.

[5] Ramesh Bharadwaj, Judith Froscher, Amit Khashnobish, and James Tracy. An infrastructure for secure interoperability of agents. Technical report, Naval Research Lab, July 2002.

[6] Ramesh Bharadwaj and Supratik Mukhopadhyay. SOLj: A Domain-Specific language (DSL) for secure Service-Based systems. pages 173–180, March 2007.

[7] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating, 2003.

[8] Wen-Ke Chen, M. A. Hiltunen, and R. D. Schlichting. Constructing adaptive software in distributed systems. In *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 635–643. IEEE, April 2001.

[9] Hana Chockler, Joseph Y. Halpern, and Orna Kupferman. What causes a system to satisfy a specification? *ACM Trans. Comput. Logic*, 9(3):1–26, June 2008.

[10] Nurit Dor, Tal L. Ami, Shay Litvak, Mooly Sagiv, and Dror Weiss. Customization change impact analysis for erp professionals via program slicing. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 97–108, New York, NY, USA, 2008. ACM.

[11] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *of Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[12] Dominic Duggan. Type-based hot swapping of running modules (extended abstract). In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 62–73, New York, NY, USA, 2001. ACM.

[13] David Evans. Programming the swarm. Technical report, Naval Research Lab, 2000.

[14] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael C. Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, New York, NY, USA, 2005. ACM.

[15] Matthew L. Ginsberg. Counterfactuals. *Artificial Intelligence*, 30(1):35–79, October 1986.

[16] G. Scott Graham and Peter J. Denning. Protection: principles and practice. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, AFIPS '72 (Spring), pages 417–429, New York, NY, USA, 1972. ACM.

[17] David Gries. *The Science of Programming (Monographs in Computer Science)*, chapter 7, pages 108–113. Springer, February 1987.

[18] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(3):229–247, June 2006.

[19] Bo Guo and Mahadevan Subramaniam. Formal change impact analyses of extended finite state machines using a theorem prover. *Software Engineering and Formal Methods, International Conference on*, pages 335–344, 2008.

[20] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *Software Engineering, IEEE Transactions on*, 22(2):120–131, February 1996.

[21] Nicolas Halbwachs. Delay analysis in synchronous programs. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 333–346. Springer Berlin / Heidelberg, 1993.

[22] Bob Hale and Aviv Hoffmann. *Modality: Metaphysics, Logic, and Epistemology*, chapter 4, pages 81–96. Oxford University Press, USA, May 2010.

[23] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. pages 231–261, April 1996.

[24] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and GJ. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[25] S. S. Iyengar and R. R. Brooks. *Distributed Sensor Networks*. Chapman & Hall/CRC, 2005.

[26] S. Kogekar, S. Neema, and X. Koutsoukos. Dynamic software reconfiguration in sensor networks. In *Systems Communications, 2005. Proceedings*, pages 413–420. IEEE, August 2005.

[27] Robert A. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.

[28] Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, January 1974.

[29] David K. Lewis. *Counterfactuals*, chapter 6, pages 118 – 143. Wiley-Blackwell, 2nd edition, January 2001.

[30] Ronaldo Menezes and Robert Tolksdorf. A new approach to scalable linda-systems based on swarms. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 375–379, New York, NY, USA, 2003. ACM.

[31] Bertrand Meyer. *Eiffel : The Language (Prentice Hall Object-Oriented Series)*. Prentice Hall, 1 edition, October 1991.

[32] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[33] Graeme Mitchison and Richard Jozsa. Counterfactual computation. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 457(2009):1175–1193, May 2001.

[34] Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, March 2000.

[35] Manuel Peralta and Supratik Mukhopadhyay. Code-Change impact analysis using counterfactuals. *Computer Software and Applications Conference, Annual International*, 0:694–699, 2011.

[36] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis: A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278, May 1980.

[37] Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proceedings of the 21st annual meeting on Association for Computational Linguistics*, ACL '83, pages 137–144, Stroudsburg, PA, USA, 1983. Association for Computational Linguistics.

[38] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[39] Pascal Poizat, Gwen Salaün, and Massimo Tivoli. On Dynamic Reconfiguration of Behavioural Adaptations. In *Proceedings of the 3rd International Workshop on Coordination and Adaptation Tech- niques for Software Entities (WCAT'06)*, pages 61–69.

[40] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 39, pages 432–448, New York, NY, USA, October 2004. ACM.

[41] Ramesh Rharadwaj. Development of dependable component-based distributed applications. In Tiziana Margaria, Bernhard Steffen, Anna Philippou, and Manfred Reitenspie, editors, *International Symposium on Leveraging Applications of Formal Methods, ISoLA 2004, October 30 - November 2, 2004, Paphos, Cyprus. Preliminary proceedings*, volume TR-2004-6 of *Technical Report*, pages 194–200. Department of Computer Science, University of Cyprus, 2004.

[42] Kay Römer, Oliver Kasten, and Friedemann Mattern. Middleware challenges for wireless sensor networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):59–61, October 2002.

[43] Fred B. Schneider. *On Concurrent Programming (Texts in Computer Science)*, chapter 3, pages 55–74. Springer, 1 edition, May 1997.

[44] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.

[45] Ehud Y. Shapiro. Alternation and the computational complexity of logic programs. *The Journal of Logic Programming*, 1(1):19–33, June 1984.

[46] Junrong Shen, Xi Sun, Gang Huang, Wenpin Jiao, Yanchun Sun, and Hong Mei. Towards a unified formal model for supporting mechanisms of dynamic component update. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, volume 30 of *ESEC/FSE-13*, pages 80–89, New York, NY, USA, September 2005. ACM.

[47] Brian Skyrms. Counterfactual definiteness and local causation. *Philosophy of Science*, 49(1):43–50, 1982.

[48] Leon Sterling and Ehud Y. Shapiro. *The art of Prolog : advanced programming techniques*. MIT Press, 2 edition, March 1994.

[49] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: Safe and predictable dynamic software updating. In *In Proceedings of the 32 nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL*, pages 183–194, 2005.

[50] Mahadevan Subramaniam, Bo Guo, and Zoltan Pap. Using change impact analysis to select tests for extended finite state machines. *Software Engineering and Formal Methods, International Conference on*, pages 93–102, 2009.

[51] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic accommodation of change: automated architectureconfiguration of distributed systems. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 287–290. IEEE, October 1999.

[52] E. Tressler. Inter-agent protocol for distributed sol processing. Technical report, Naval Research Lab, 2002.

[53] Ji Zhang and Betty H. C. Cheng. Specifying adaptation semantics. In *Proceedings of the 2005 workshop on Architecting dependable systems*, WADS '05, pages 1–7, New York, NY, USA, 2005. ACM.

[54] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, volume 0 of *ICSE '06*, pages 371–380, New York, NY, USA, 2006. ACM.

[55] Shi Zhang and Linpeng Huang. Formalizing Class Dynamic Software Updating. In *Quality Software, 2006. QSIC 2006. Sixth International Conference on*, pages 403–409, Washington, DC, USA, October 2006. IEEE.

Appendix: Copyright Forms for Published Chapters

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE.Please read the form carefully and keep a copy for your files.

TITLE OF PAPER/ARTICLE/REPORT, INCLUDING ALL CONTENT IN ANY FORM, FORMAT, OR MEDIA (hereinafter, "The Work"):**Code-Change Impact Analysis Using Counterfactuals**

COMPLETE LIST OF AUTHORS:**Manuel Peralta**

IEEE PUBLICATION TITLE (Journal, Magazine, Conference, Book):**2011 35th IEEE Annual Computer Software and Applications Conference**

COPYRIGHT TRANSFER

1. The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers,Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the above Work,including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

CONSENT AND RELEASE

2. ln the event the undersigned makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the undersigned, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction ofthe Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast ofthe Presentation during or after the conference.

3. In connection with the permission granted in Section 2, the undersigned hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale ofproducts incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

4. The undersigned hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the undersigned has obtained any necessary permissions. Where necessary, the undersigned has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE.

[ ] Please check this box ifyou do not wish to have video/audio recordings made ofyour conference presentation.

See below for Retained Rights/Terms and Conditions, and Author Responsibilities.

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE

PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEEs publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/pub_tools_policies.html. Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

General

1. Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.

2. Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.

3. In the case of a Work performed under a U.S. Government contract or grant, the IEEE recognizes that the U.S. Government has royalty-free permission to reproduce all or portions of the Work, and to authorize others to do so, for official U.S. Government purposes only, if the contract/grant so requires.

4. Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.

5. Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

Author Online Use

6. Personal Servers. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore.Authors shall not post the final, published versions of their papers.

7. Classroom or Internal Training Use. An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the authors personal web site or the servers of the authors institution or company in connection with the authors teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials,course packs, e-reserves, conference presentations, or in-house training courses.

8. Electronic Preprints. Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employers site, or to another server that invites constructive comment from colleagues.Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the IEEE work with a Digital Object Identifier (DOI) or link to the article abstract in IEEE Xplore, or (2) the accepted version only (not the IEEE-published version), including the IEEE copyright notice and full citation, with a link to the final, published article in IEEE Xplore.

INFORMATION FOR AUTHORS

IEEE Copyright Ownership

It is the formal policy of the IEEE to own the copyrights to all copyrightable material in its technical publications and to the individual contributions contained therein, in order to protect the interests of the IEEE, its authors and their employers, and, at the same time, to facilitate the appropriate re-use of this material by others.The IEEE distributes its technical publications throughout the world and does so by various means such as hard copy, microfiche, microfilm, and electronic media.It also abstracts and may translate its publications, and articles contained therein, for inclusion in various compendiums, collective works, databases and similar publications.

Author/Employer Rights

If you are employed and prepared the Work on a subject within the scope of your employment, the copyright in the Work belongs to your employer as a work-for-hire. In that case, the IEEE assumes that when you sign this Form, you are authorized to do so by your employer and that your employer has consented to the transfer of copyright, to the representation and warranty of publication rights, and to all other terms and conditions of this Form.If such authorization and consent has not been given to you, an authorized representative of your employer should sign this Form as the Author.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing grant of rights shall become null and void and all materials embodying the Work submitted to the IEEE will be destroyed.
4. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.

Manuel Peralta	**28-04-2011**
Author/Authorized Agent For Joint Authors	**Date(dd-mm-yy)**

THIS FORM MUST ACCOMPANY THE SUBMISSION OF THE AUTHOR'S MANUSCRIPT.
Questions about the submission of the form or manuscript must be sent to the publication's editor.Please direct all questions about IEEE copyright policy to:
IEEE Intellectual Property Rights Office, copyrights@ieee.org, +1-732-562-3966 (telephone)

ACM CONFERENCE COPYRIGHT FORM AND AUDIO/VIDEO RELEASE

Title of the Work: Counterfactually Reasoning About Security

Publication and/or Conference Name: Proceedings of the 4th International Conference on Security of Information and Networks Proceedings

Author/Presenter(s): Manuel Peralta; Supratik Mukhopadhyay; Ramesh Bharadwaj

Auxiliary Materials (provide filenames and a description of auxiliary content, if any, for display in the ACM Digital Library. The description may be provided as a ReadMe file):

I. COPYRIGHT TRANSFER

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable -**see Part I. B. below**) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Employer / Author(s) Retained Rights. Each of the Employer/Author(s) retains the following rights:

1. All other proprietary rights to the work such as patent;
2. The right to reuse any portion of the Work, without fee, in future works of the Authors (or Authors Employers) own, including books, lectures and presentations in all media, provided that the ACM citation, notice of the Copyright and the ACM DOI are included (See Section 4 below). Requests made on behalf of others, however (i.e. contributions to the work of other authors or other editors), usually require payment of a fee;
3. The right to revise the work. (See Policy 2.4 Definitive Versions and Revisions);
4. The right to post author-prepared versions of the Work covered by the ACM copyright in a personal collection on their own home page, on a publicly accessible server of their employer and in a repository legally mandated by the agency funding the research on which the Work is based. Such posting is limited to noncommercial access and personal use by others, and must include the following notice both embedded within the full text file and in the accompanying citation display as well:
   " ACM, (YEAR). This is the authors version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in PUBLICATION, {VOL#, ISS#, (DATE)} http://doi.acm.org/10.1145/{nnnnnn.nnnnnn}".
   (You may find the nnnnnn.nnnnnn number for your article DOIs on its citation page in the ACM Digital Library.)
5. The right of an employer who originally owned the copyright to distribute definitive copies of its author-employees Work within its organization. Posting these works for access outside of the employers organization requires explicit permission from ACM.

Authors should understand that consistent with ACMs policy of encouraging dissemination of information, each work published by ACM appears with a copyright and the following notice:

(c) 2011 Association for Computing Machinery. ACM acknowledges that this

contribution was authored or co-authored by an employee, contractor or affiliate of the national government of . As such, the government of retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

☑ A. Assent to Assignment. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)), with the exception of third party materials detailed in section III below. I have obtained permission for any third-party material included in the Work.

☐ B. Declaration for Government Work. I am an employee of the National Government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are you a contractor of your National Government? ○ Yes ◉ No

Are any of the co-authors, employees or contractors of a National Government? ◉ Yes ○ No

Government Agency: Naval Research Laboratory

Country: United States

II. PERMISSION FOR CONFERENCE TAPING AND DISTRIBUTION (Check A and, if applicable, B)

A. Audio /Video Release

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately by itself as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Do you agree to the above Audio/Video Release? ◉ Yes ○ No

B. Auxiliary Materials, not integral to the Work

I hereby grant ACM permission to serve files named below containing my Auxiliary Material from the ACM Digital Library. I hereby represent and warrant that my Auxiliary Material contains no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software, and I hereby agree to indemnify and hold harmless ACM from all liability, losses, damages, penalties, claims, actions, costs and expenses (including reasonable legal expense) arising from the use of such files.

☐ I agree to the above Auxiliary Materials permission statement

III. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.

◉ We/I have not used third-party material.
○ We/I have used third-party materials and have necessary permissions.

IV. Artistic Images

If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part IV and be sure to include a notice of copyright with each such image in the paper.

◉ We/I do not have any artistic images.
○ We/I have have any artistic images.

V. Liability Waivers & Indeminifcations

* Your Liability Waiver is conditional upon you agreeing to the terms set out below.

Because I retain certain rights in my work under the ACM Copyright Transfer Agreement and under the ACM Permissions Release Form, such as patent and moral rights, I therefore hereby agree not to assert any of my rights against ACM in connection with ACM's use of my work as agreed to herein, and I further acknowledge that ACM is under no obligation to exercise all of the rights I have granted.

I hereby agree to indemnify ACM and its agents and assigns against any and all losses incurred in connection with any claim or proceedings asserting that I have violated a prior agreement in presenting my work at an ACM event and/or in granting ACM rights to publish my work.

I hereby agree to indemnify ACM and its agents and assigns against any and all losses incurred in connection with any claim or proceeding asserting plagiarism and/or copyright infringement if the investigation carried out by ACM according to its Plagiarism Policy (see: http://www.acm.org/publications/policies/plagiarism_policy) determines that my work is the plagiarizing or infringing work. [Note, in accordance with its policies, ACM generally defends its authors against charges of plagiarism and will reasonably investigate others on behalf of the author who plagiarize a work copyrighted and published by ACM.]

All permissions and releases granted by me herein shall be effective in perpetuity and shall extend and apply to ACM and its agents and assigns.

All permissions and releases granted by me herein shall be effective in perpetuity and extend and apply to ACM and its assigns, contractors, sublicensed distributors, successors, and agents.

☑ I agree to the above liability waiver

DATE: **09/22/2011** sent to mperal4@lsu.edu at **16:09:57**

Editor: Iyengar **Cat. No. K12917**

Taylor & Francis Group, LLC Contributor Agreement

(REQUIRED) *
It is agreed on this* 24th **day** of * August **2012**, by and between the following:

***Chapter Author(s):** Manuel Peralta, Supratik Mukhopadhyay, Ramesh Bharadwaj
(Please print name)

and Taylor and Francis Group, LLC, that the above-mentioned Author(s) shall prepare textual material, including all references, figures and tables, typed double-spaced on 8 1/2 X 11 inch paper, prepared in accordance with the *Author's Guide to Publishing*, a copy of which will be provided by the Publisher. In addition to the Manuscript, the artwork must be delivered on disk as well as paper. The Work is due to the Editor for the chapter tentatively entitled:

***Chapter Number:** 33 ***Title:** Reasoning about Sensor Networks

commissioned by Taylor and Francis, LLC, for use as a contribution to a collective work (tentatively) entitled: ***Distributed Sensor Networks, Second Edition: Image and Sensor Signal Processing* (Volume 1),** which shall be deemed to be a work made for hire. As such, copyrights in this publication will insure to the benefit of Taylor and Francis Group, LLC, and the company will own the publication, its title and component parts, and all publication rights. This permits the company, in its own name, to claim copyrights in the work, make applications to register its copyright claims, and to renew its copyright certificate.

The Contributor represents that the Manuscript is original except for the material in the public domain and such excerpts from other Works as may be included with prior written permission of the copyright owners in both electronic and book form in perpetuity.

If the Contribution is accepted for publication in the Work by the Publisher, the Publisher agrees that the Contributor shall receive: a credit as the author of the Contribution in the first and subsequent editions of the Work in which the Contribution appears.

Warranty & Permissions: You represent and warrant to us that the article and all figures, illustrations, tabular material and other supplementary material shall be original on your part. You further warrant that the article shall contain no libelous or unlawful statements, contain no instructions that may cause harm or injury and shall not infringe upon or violate any copyright, trademark, or other right or the privacy of others.

Please note that each contributor must complete an individually signed agreement

Addresses provided on this form are used for mailing correspondence as well as contributor copies. Please notify us of any address changes, so that you do not miss out on your complimentary copy.

***Contributor Signature:** [signature] ***Date:** Aug 24, 2012

***Contributor Full Name (Print in all capital letters):** Manuel Peralta

***AFFILIATION (Company, Univ.):** Louisiana State University

***MAILING ADDRESS:** No P.O. Boxes please. **If a University address, please include Building Name, Number & Street Address.**

This is a home address ☐

Department of Computer Science and Engineering
3127 Patrick F. Taylor Hall
Baton Rouge, LA, 70803

* **PHONE:** 435-216-2555 **FAX:** 225-578-1465

***E-MAIL ADDRESS:** mperal4@lsu.edu

This Agreement must be on file prior to the publication of the Work. Please return this form to **Samantha White** as soon as possible

Fax **+001 (561) 997-7249** or email:**Samantha.white@taylorandfrancis.com**

FOR OFFICE USE ONLY: Publisher: **Date:**

Vita

Manuel Alfonso Peralta was born on 8th of January 1981 in Santo Domingo, Dominican Republic. He obtained a Bachelors Degree in Systems Engineering and Applied Computer Science from Pontificia Universidad Catlica Madre y Maestra (PUCMM) in 2003. After obtaining his bachelors degree (Summa Cum Laude) he worked in the banking industry as a programmer analyst from 2003 to 2005. Concurrently, he also worked as a lead co-designer for a new project which implemented the use of a new iterative-based software development methodology based mostly on the Rational Unified Process(tm). In 2005, the Dominican Republic government (via the Ministry of Higher Education) awarded him with a scholarship to pursue a masters degree in the United States of America. In Fall 2005, he joined Utah State University (USU) as a graduate student. While pursuing his masters degree he worked on the topic of *Design Rationale in The Context of Agile Software Development Methodologies*. In Spring 2007 he was awarded the degree of Master of Science in Computer Science. In Fall 2007 he chose to continue his graduate education by enrolling in USU's Computer Science Ph.D. program. While pursuing his PhD degree he worked and published in the degree of formal verification, distributed networks, and applications of various kinds of logics (e.g. propositional, first-order, modal) to the field of software verification. In Spring 2010 he transferred to Louisiana State University (LSU) in order to continue pursuing his PhD degree. In Summer of 2012 he successfully defended his dissertation titled: "Perpetual Requirements Engineering" which shows how we can use *Counterfactual Logic* in program analysis

and verification, sensor networks and security. He will be formally awarded his PhD degree on December 14 2012.