Policy-Driven Framework for Static Identification and Verification of Component Dependencies

by

Anastasios Livogiannis

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2011

Abstract

Software maintenance is considered to be among the most difficult, lengthy and costly parts of a software application's life-cycle. Regardless of the nature of the software application and the software engineering efforts to reduce component coupling to minimum, dependencies between software components in applications will always exist and initiate software maintenance operations as they tend to threaten the "health" of the software system during the evolution of particular components. The situation is more serious with modern technologies and development paradigms, such as Service Oriented Architecture Systems and Cloud Computing that introduce larger software systems that consist of a substantial number of components which demonstrate numerous types of dependencies with each other. This work proposes a reference architecture and a corresponding software framework that can be used to model the dependencies between components in software systems and can support the verification of a set of policies that are derived from system dependencies and are relative to the software maintenance operations being applied. Dependency modelling is performed using configuration information from the system, as well as information harvested from component interface descriptions. The proposed approach has been applied to a medium scale SOA system, namely the SCA Travel Sample from Apache Software Foundation, and has been evaluated for performance in a configuration specification related to a simulated SOA system consisting to up to a thousand web services offered in a few hundred components.

Acknowledgements

"To my parents I owe my being,
and to my teachers my well-being"

Alexander the Great for his teacher, Aristotle

First and foremost, I would like to thank my supervisor, Prof. Kostas Kontogiannis for his invaluable help to form my personality as a researcher all these wonderful years I am under his supervision. His advices and exemplary behaviour were and still are beneficial not only during my academic curriculum but through my entire life, as he taught me the meaning of important concepts such as professionalism and critical, research-oriented way of thinking.

Prof. Ladan Tahvildari, my co-supervisor, holds an important role in my graduate curriculum as a masters student at the University of Waterloo. Apart from the significant technical knowledge I received from working with her and her course, her advices on balancing one's life and managing one's time will benefit the entire life of every person who has listened to her.

Acknowledgements go to professors Rudolph Seviora and Paulo Alencar for being the academic readers of this work and taking the time to provide me with helpful and constructive comments.

Last but not least, I would like to thank my parents. Vocabularies of all the world's languages combined do not contain the proper wealth to describe how important is the support I received from my parents. Patiently, listening to my needs, hopes and dreams they were prepared to make a sacrifice in order for me to achieve my goals and make my dreams reality. Even in the darkest times my family has faced, they were always supportive even with the harsh decision to leave them and pursue my dreams and a better life abroad. There is no synonym of *"Thank you"* that can describe the gratitude that I offer to them.

Dedication

This thesis is dedicated to my parents and my sister.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Modern software systems' complexity ever increases due to the growing demands of software customers and the advances in software engineering tools, computer hardware and networks. Additionally, most contemporary software systems are distributed in nature, something that is intensified more with the paradigm of Cloud Computing and modern distributed and network-based software platforms. In this context, it is hard to identify dependencies between the different components of a large software system; hence making software maintenance an even harder issue than it already is.

From all the stages of the life-cycle of the whole software system development, the stage of *software maintenance* is the lengthiest and the most costly of them all, as seen in the pie chart of figure 1.1 from [57]. That is the main reason why it has received significant attention from the scientific and industrial community. Efforts toward improvements of *software maintenance* can reduce software costs and increase the reliability of software systems. The stage of *software maintenance* is the target of the present work.

Software maintenance can be divided into the four following categories, according to [40].

Corrective: Finding errors in software code and performing the necessary actions to fix them.

Perfective: Incorporating new features in the software system, based on increased/changed user needs and enhancing the abilities of the software product.

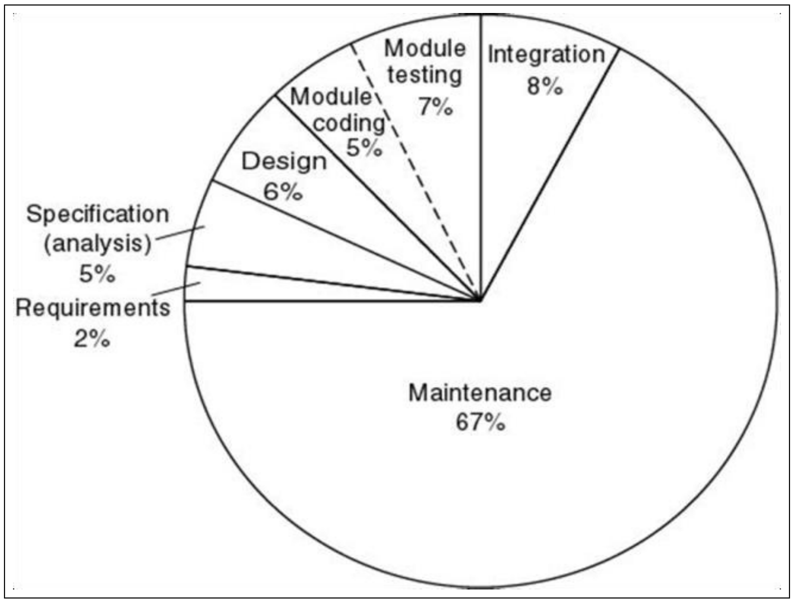


Figure 1.1: Cost percentage of different stages of the software development life-cycle [57]

Adaptive: Changing the software system as to be able to perform in environments different than what was originally intended (i.e. porting it to a different platform).

Preventive: Enhancing the quality of the software trying to reduce the maintenance effort needed in the future.

Because of the importance of software maintenance to the cost and efforts needed for software development, this thesis focuses on that phase of the software development life-cycle.

1.2 Problem Description

It is evident that software system evolution is the main consequence of software maintenance. Usually, during such maintenance operations there is the need to change only one part of the software system. Hence, a number of problems might arise due to the dependencies that exist between the system that does not change and its portion that will be different after a maintenance operation. In this respect, the present work focuses on dependency modelling and analysis during software maintenance of large software systems, by applying the proposed methodologies to service oriented software systems, such as SCA specified systems.

Large service oriented software systems are complex applications that require the use of diverse and distributed components. Examples of such systems are banking applications, procurement systems, payroll systems and customer resource management systems. Typical maintenance scenarios in these systems include the replacement of a component with a new component, the upgrade of a component to a newer version, or the deprecation of a component altogether. Such maintenance operations in large applications require significant planning, as the potential for undesired side effects affecting other components could be really high. To date, the software engineering community is attempting to address this problem in two major directions.

The first direction relates to static and dynamic analysis of a software system for identifying a number of different types of dependencies between components and predicting to a certain extend, ripple effects that may occur when a component in a system will be altered. The second direction relates to just-in-time analysis that is performed as changes are applied by the use of installers which apply scripts that perform a series of tests with respect preconditions and resources that need be in place for the installation or the upgrade to be completed successfully. However, these approaches require either component dependency models to be compiled or they focus only on the component being installed or upgraded and fall short on taking a global view of how other components of the system are affected by this installation or upgrade.

The approach of the current thesis is to first model dependencies between software entities in a general manner and then present a framework where the proposed models will be used for checking the validity of software maintenance scenarios. Based on the discussion so far, the questions this thesis tries to address are presented in the following list:

1. Can the dependencies of any software system be modelled in a general manner, facilitating software checks during the maintenance phase?
2. Is it possible for a framework to be able to focus on specific types of dependencies, based on the context of the needed checks for a specific maintenance scenario?
3. Can all the above be incorporated into a single framework that will be able to model dependencies between software artifacts and check their violation during the software maintenance phase?

4. Can this framework and those dependency models be extensible enough to accommodate domain-specific characteristics of the dependencies and use them in the verification policy?

1.3 Thesis Contributions

The contribution points of this thesis are as follows:

First, proposes a Dependency Multi-Graph Meta-Model for representing parallel dependencies between software artifacts, which is agnostic to the specific domain of the software system being modelled. Furthermore, it attempts to provide the necessary means to facilitate extensions in cases where this is needed to incorporate domain-specific information.

Second, proposes a Dependency Tree Meta-Model which can be derived from the aforementioned Multi-Graph and it provides focus to dependencies that are important to be checked in a specific context (e.g. when checking for interface compatibility).

Third, it proposes a process and a corresponding algorithm that facilitates the generation of the Dependency Tree from the Dependency Multi-Graph, given the correct context.

Fourth, it provides a framework that uses the aforementioned Meta-Models in order to facilitate the dependency checking during the software maintenance phase.

Fifth, it evaluates the applicability of the proposed framework to a case study, where the proposed framework is used for dependency analysis in Service Component Architecture (SCA) systems.

Sixth, it comments on the performance of the proposed framework in a use-case study comprising of a significant number components and services.

1.4 Organization of the Thesis

The present chapter, 1, introduces the thesis by explaining the motivation behind it, describing the problem that it targets and stating the contributions this thesis aims to offer.

The next chapter, chapter 2, provides an overview of related work in the areas of software maintenance, software modelling and impact/dependency analysis. This overview targets methodologies and results that inspired/assisted this work, as well as tools leveraged for the completion of the current thesis.

Chapter 3 provides the reader an insight of the structure and function of the proposed framework. That chapter describes the functionality of every major component of the proposed framework, its inner structure, as well as the interfaces with which the different components of the framework can work together. It is safe to state that chapter 3 will give the reader the adequate information to understand the overall process of the framework's function and the details of the structure of the individual units that it is composed of.

Chapter 4 describes the data models that are leveraged to model dependencies between software artifacts in the proposed framework. In that chapter, both the Dependency Multi-Graph and the Dependency Tree are presented and explained in details. Furthermore, the exact algorithm of the generation of a Dependency Tree from the Multi-Graph is presented and explained through examples.

Next chapter, 5 depicts how the framework leverages the aforementioned dependency models to perform dependency analysis in software systems. The algorithm of how a Goal Tree is analyzed and decomposed into atomic predicates to be verified or not is presented along with an example of Goal Tree verification.

Chapter 6 provides the architecture of a prototype implementation of the framework described in this thesis. It also provides case studies of how the prototype software can be used in analyzing dependencies of real-word service component architecture (SCA) software systems.

Chapter 7 summarizes the thesis explaining how the thesis contribution is achieved and giving further ideas/guidance for future work from other researchers. Last chapter gives the references that inspired/assisted the work of the current thesis.

Chapter 2

Related Work

In the related research technical literature we can identify a number of related work areas. This chapter briefly discusses a number of related work in the general area of software engineering that inspired or assisted the present thesis to be completed. Related pieces of work that will be discussed span into the areas of software modelling, software maintenance, model driven engineering and impact analysis. This chapter will present not only methodologies in those areas, but also tools that can be leveraged by researchers or practitioners in software engineering.

2.1 Modelling Frameworks

2.1.1 Modelling Software Systems with EMF

In the area of modelling frameworks, Eclipse Modelling Framework (EMF), [11], is the "one-stop shop" for researchers and practitioners who require a reliable and feature-rich tool for software modelling and manipulation of the software models. The term "manipulation" in this case involves generating the appropriate code to support the software model, store the model in XMI, [23], or relational databases and transform models using languages like the Atlas Transformation Language, [35]. EMF relies on OMG's Meta Object Facility (MOF) Meta-Modelling language, [48] which was introduced to model the Unified Modelling Language (UML), [49].

UML is a diagram-based modelling language to model object-oriented software systems, or systems in general. With UML, stakeholders can describe the structural and behavioural specification of a system. Well-known and very common diagrams for the first case are class diagrams and component diagrams. Class diagrams are very common in modelling object-oriented systems. Class diagrams offer the means to model object-oriented concepts such as classes of objects, inheritance between classes, references of classes from other classes and member encapsulation. Component diagrams offer a more abstract view of the system that is designed, by showing different components and the interactions among them, via provided and required interfaces. In such a context, components can be implemented via a number of classes or even a number of other components, thus forming a composite component. For modelling the behaviour of a system, UML offers different types of diagrams, most common among them are sequence diagrams and state diagrams. Sequence diagrams show the lifetime of an object in the system and its interactions with other objects, during their lifetimes. Interactions in this context, refer to messages (e.g. method invocations) either synchronous or asynchronous. State diagrams show the state of each objects in the system and the transition to other states, based on inputs received from other objects and/or the environment.

MOF is a meta-model trying to define UML itself. It offers all the possible means to model concepts such as class of objects (NamedElements), methods and operations of classes of objects (StructuralFeature) etc. MOF comes in two variants, the Essential MOF (EMOF) and the Complete MOF (CMOF). EMOF is regarded as a subset of MOF that has the ability to relieve stakeholders from complex modelling concepts when designing simple meta-models to be developed via code. Concepts of EMOF are closely related to concepts of Object-Oriented modelling. On the other hand, CMOF is the full featured MOF meta-model that allows for complex development of meta-models for languages and new paradigms. EMF utilizes its own meta-model, Ecore for modelling object-oriented software related concepts. Ecore derives from EMOF and models lower-level concepts than CMOF, in order for code generation to be more straightforward and manageable. Apart from code generation, EMF offers a way to store model instances in XMI or databases and a Java API in order to allow the programmatic handling of the models (initialization, manipulation etc.). EMF and Ecore were leveraged by this thesis for generating code for UML models and model manipulation through the Java API of EMF.

2.1.2 SOA systems and SCA

Service Oriented Architecture, [29, 43], or SOA for short, is a new software paradigm that facilitates the composition of large software systems from software services that can be interfaced dynamically, thus reducing the coupling that some object-oriented systems demonstrate. The essence of Service Oriented Architecture is the existence of a *Service*

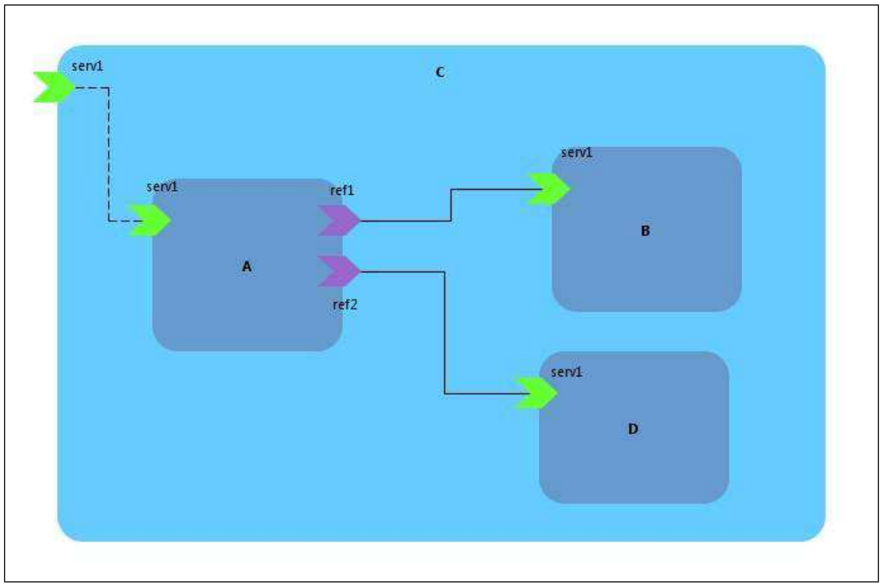


Figure 2.1: Example of a SCDL graphic diagram

Registry where *Service Providers* can publish their offered services and *Service Consumers* can find and utilize the published services. This relieves the developers for having to maintain references to objects as in the previous object-oriented model.

Service Component Architecture, [42, 16], is a software specification of a platform that implements the SOA software development paradigm. It facilitates the interfacing and wiring of services by providing an innovative configuration mechanism and its supporting language, Service Component Description Language (*SCDL* for short).

The SCA specification consists of two pillars, the *Assembly Model* [47] and the *Policy Framework* [46], along with extensions to these pillars, especially the former. Assembly model is used to define the structural composition of an SCA system and its offered and consumed services. The major artifact in the assembly model is the *Component*. Components are usually a piece of independent business functionality that the system offers. It is the smallest piece of SCDL modelling that can have its own implementation (through a Java class or an html page with javascript code etc.). It offers *services* that other components or clients can consume and can have *references* to other services that it consumes. Usually, the services of a component's implementation conforms to a specific interface, modelled by either a Java interface or a Web Service Definition Language (WSDL) file. This interface defines the parameters and return type of the services that the component offers.

Components are bundled into *Composites*. Composites are artifacts that facilitate the comprehension of the system's deployment, as it is usual for a composite to bundle components that will be deployed on the same server/computer. Nevertheless, composites exist to group components that offer common business functionality and/or use similar computer resources (files, databases etc.). Composites can offer services that are forwarded to services offered by inner components, called "promoted" services, although this is optional. SCDL models can be also illustrated using Composite Diagrams, that offer a diagrammatic demonstration of SCDL xml files (although, more limited).

Figure 2.1 shows such a Composite Diagram. A composite named *C* with three components, namely *A*, *B* and *D* is illustrated. Component A offers one service that is promoted to be offered by the composite, as well. It also has two references, *ref1* and *ref2*. The first reference refers to a service offered by component *B* and the latter references a service offered by component *D*. Graphic representation of SCDL files usually is not very illustrative in terms of bindings and wires between services. In figure 2.2 an xml SCDL file is illustrated. It models a composite named "composite" that has only one component named "component". The component is implemented via the Java class "componentImpl" that can be found in the package "pack". The component offers one service named "service" and has a reference to a service called "reference". The *binding.ws* tag depicts that the referenced SCA service can be reached in the mentioned uri and the suffix "ws" shows that it should be called as a web service (through HTTP using SOAP). Generally, SCA specification supports a number of ways for remote method invocation and specific implementations of it offer an even wider range of binding types (such as json-rpc). The *callback* tag defines the callback interface to be used in asynchronous calls. The callback interface can be reached at the specified uri of the following "binding.ws" tag and can be called as a web service.

Another important pillar of the SCA specification is the Policy Framework. It allows stakeholders to model certain policies that should be applied to the binding of services with each other. Figure 2.3 shows an example of a service binding that requires a security policy to be applied when other services or clients invoke the service named "service". "Security" is an *intent* that the underlying SCA runtime platform which performs the service wiring should implement. Intents are the names of policies that a binding or wire between services can require. *PolicySets* are a collection of intents that a binding or wire offers. SCA Policy Framework contains a number of already defined intents and implementations of the SCA specification provide the appropriate policySets that conform to them. Additionally, the user has the possibility to model policies using the WS-Policy specification, [6].

```
<composite name="composite">
<component name="component">
  <implementation.java class="pack.componentImpl"/>
    <service name="service"/>
    <reference name="reference">
      <binding.ws uri="http://localhost:8086/WebApp/Reference"/>
      <callback>
        <binding.ws name="callback"
          uri="http://localhost:8084/WebApp/ReferenceCallback"/>
      </callback>
    </reference>
</component>
</composite>
```

Figure 2.2: SCDL xml file example

```
...
    <service name="service">
        <binding.ws uri="..." requires="security"/>
    </service>
...
</component>
</composite>
```

Figure 2.3: SCDL xml file policy example

2.1.3 Systems Modelling and Modelling Management

Apart from the tools and paradigms, in the area of systems modelling a number of related works can be identified. In [13] a framework for specifying functional and non-functional requirements as a collection of soft goals is presented. The framework allows for the logical decomposition of complex requirements into simpler ones and ultimately allows for the association of requirements with design decisions. In [71] a framework to trace aspects identified during goal-oriented requirements analysis is presented. The framework allows for the validation of the resulting system in light of shareholders' cross-cutting concerns and whether the weaved system with aspects indeed improves system qualities measured by the degree of goal satisfaction.

In the area of model management, in [55] a formal multi-modeling framework that allows specialized model relationship types to be defined between software models that constitutes a new type of model called a macro-model is presented. Macro-models can be used to support Model Driven Engineering and model management and maintenance activities. In [53], an approach that allows for consistency checking of distributed models is discussed. (i.e., models developed by distributed teams). The approach is based on the construction of a merged model before checking for model consistency between models and based on consistency predefined relations. In [8] meta-data management is addressed using model management techniques where schemas and interface definitions are treated as first class objects with basic management operations. A catalogue of various model synchronization schemes has been presented in [1] where model transformations are classified according to their external behaviour characteristics.

Generating code from models is a contemporary phenomenon in software engineering. Legacy code with little or no specification/design documentation can be undergone reverse engineering approaches that will allow the extraction of architectural elements. An example of a tool that allows for the visualization of architectural diagrams extracted through analysis of the build process, is the "Landscape Editor", *LSEdit*, [60]. Using a fact extractor reverse engineering tool like *LDX*, [67], it is possible to extract information for the dependencies between artifacts of an application, in different levels of abstractions (dependencies of functions from variables, of source code from specific libraries etc.). Those dependencies can be visualized in the window of LSEdit and undergo a number of layout transformations. Interestingly, there are promising results in using such fact extraction and visualization tools for identifying code clones, [15].

2.2 Software Maintenance

Software maintenance is a large sub-area under the umbrella of Software Engineering. A plethora of diverse technologies have been studied by researchers in their attempt to improve software maintenance, as it is one of the most important aspects of software development. From Formal Concept Analysis and Model Synchronization, through software monitoring, impact and dependency analysis, this section discusses a relevant to this thesis and important part of the wealth of knowledge produced by software scientists and researchers in this area.

2.2.1 Impact and Dependency Analysis

From the beginning of software, dependency analysis was used for improving the performance of software, especially in compilers. Dependence graphs or dependence trees were and remain important models that heavily contribute to low-level dependency analysis and to compiler optimization, [36]. In general, dependence graphs were introduced to model dependencies between software components. In the context of compilers they allowed modelling dependencies between statements of software code. Generally, dependence graphs in the area of compiler optimization is used to pinpoint flow dependencies between *read* and *write* statements. The analysis can categorize the dependencies into *flow dependency*, *anti-dependency* and *output dependence*, [27] and dependence graphs can visualize these types of dependencies. This analysis allows for rearranging the statements in order to produce execution flows that could be parallelized in multi-piped hardware architectures [27], or parallelized in different threads of programs.

Generally, dependence graphs that model low-level software behaviour can be divided into control flow graphs and data flow graphs. In [17] the “Program Dependence Graph” (PDG) is presented, allowing the modelling of both control dependencies and data dependencies. Program Dependence Graph is a more abstract than an the abstract syntax tree and can be used for vectorization and parallelization of the code. Program Dependence Graphs model sequential programs, the nodes containing a set of statements of code and the edges modelling control or data dependencies between nodes. Later, in [56], the Parallel Program Dependence Graph was introduced to allow for the modelling of programs that were parallel and enhanced the PDG model with edges that either modelled parallel execution, or posed constraints in the sequential execution of code.

As software engineering progresses, the complexity and the size of the code increases. Contemporary systems consist of large numbers of inner parts, mostly components that may or may not be accompanied by their implementation source code. This is the reason why the capability to model software systems in higher level and more abstract ways is needed. In compatibility testing, [70] proposes the "RATCHET" approach, where the dependencies between different components are modelled in a Component Dependency Graph (CDG) with *AND* and *XOR* nodes. According to the "RATCHET" approach, software developers/designers create the Component Dependency Graph for the system they want to test and identify a component C (node in CDG) they want to be tested for compatibility with other nodes. Subsequently, "RATCHET" identifies all components directly dependent to C and generates the tests for compatibility checking of all the versions of C to all the versions of other components directly dependent to it. In [69], "RATCHET" approach is taken one step further, by allowing stakeholders to decorate the Component Dependency Graph with priorities between configuration management instances (versions of each component) to be checked. The upgraded "RATCHET" system performs assortment of the test plans to be checked, based on the given preferences.

2.2.2 Model Driven Engineering for Software Maintenance

While software systems' size continues to grow, Model Driven Engineering was introduced to allow for easier development of software, based on modelling techniques that might involve, for example, the aforementioned Unified Modelling Language. As models and modelling languages started receiving attention from researchers, practitioners and software stakeholders, the need of impact analysis on changes in models was apparent.

The first step in impact analysis is to be able to extract the dependencies in a model, either local or dependencies that span across multiple models that describe a software system. In model synchronization area, [30] changes due to maintenance in a software system is viewed as synchronization problem between different models. The models can be in the same or different levels of abstraction. The approach involves encoding the models via a graph, EMOF meta-model called Graph Metamodel for Synchronization (GMS) and providing a transformation algorithm between models M_1 and M_2 that are encoded using GMS. The transformation algorithm provides the proper synchronization between M_1 and M_2. In [31] and [33], formal concept analysis (FCA, [18]) is leveraged to identify dependencies between models of different abstraction levels. Along with the approach in [30], it can automate the process of identifying relations between models of the same software system in different abstraction levels, and facilitate the synchronization of the affected portions of the models, during evolution changes in the software system.

Great role in software maintenance and impact analysis plays the identification and management of inconsistencies of software artifacts when they change in time. Two constraint based consistency management frameworks for incremental maintenance of software artifacts are presented in [52],[24]. These frameworks rely on the existence of a well defined set of constraints for ensuring the consistency of the interlinked models and are capable of incremental resolution of in-consistencies for such cases.

2.2.3 Software Monitoring

Software monitoring is an important part of the maintenance phase of a software system. It allows for polling and extracting information from the system that could lead to identification of problems or even prevention of failures, therefore increasing software reliability. In the area of monitoring frameworks, [59] proposes a collection of monitoring systems. These are integrated with available hardware and software mechanisms for COTS-based systems. [9] proposes an event-based temporal object model that keeps track of selected values within the history of a data object.

Failure detection is a significant part of software monitoring. In [25] an approach to automatically identify failures of a software system is presented. The approach monitors the external behaviour of the software system and allows for cross-referencing it with the formally specified in communicating extended finite state machines (CEFSM). As a result, deviations of the system's external behaviour from the prescribed one (i.e. software failures) can be identified. Aspect Oriented Programming (AOP) is used in [61] to interweave pieces of software code that measure metrics used to infer the "health" of the software system under test. The approach defines such "health indicators", such as the ratio of successful system calls over the total number of system calls (indicator called "System Exceptions"). It also discusses a number of design patterns for the aspect oriented code, in an effort to formalize a useful design of metrics and indicators measurement code.

This approach differentiates from the aforementioned related work in two points; the dependency models used and the runtime adaptivity during the triggering and verification of the changes in the software system under maintenance. Approaches that deal with dependencies between components of a software system view the "dependency" of two components as one, unclassifiable entity. This work will try to model different types of dependencies between software components or artifacts and possibly two components will depend on each other via a diversity of types of relations. Based on the context of the software maintenance phase, only a subset of all the possible types of dependencies will be taken into account, for the verification of changes into the software system under maintenance. Furthermore, the proposed approach will provide an adapting mechanism for the verification policies to be changed at runtime. Additionally, the framework will also guide the monitoring of the system under maintenance based on the policies that need to be verified at each specific point in time. As a result, the proposed approach contains two stages of adaptive behaviour, which is not seen in other similar software systems, to the best of our knowledge.

Chapter 3

System Architecture

This chapter discusses the architecture of the proposed framework, i.e. its major components, the way those components interact with each other and the process that the framework follows in the event of dependency checking.

3.1 Major Components

Figure 3.1 depicts the component diagram of the framework's architecture. It shows the major components of that architecture and the interfaces used to pass messages between these components. The architecture implements a hybrid architectural style composed of a mainly blackboard/repository architectural style, along with a publish/subscribe style which gives an event-handling nature to the control flow of the framework. The main components of this architecture are:

Blackboard: The main component of the architecture. Its purpose is to hold the current state of the framework, update it and notify the appropriate components via the *PublishSubscribe* component.

Event Handler: This component is responsible for handling the events produced from the other components of the framework. It logs these events and forwards them to the appropriate component, *Blackboard* or *PublishSubscribe.*

PublishSubscribe: The purpose of this component is to facilitate the subscription of other components to specific types of events. It holds a list of those subscriptions

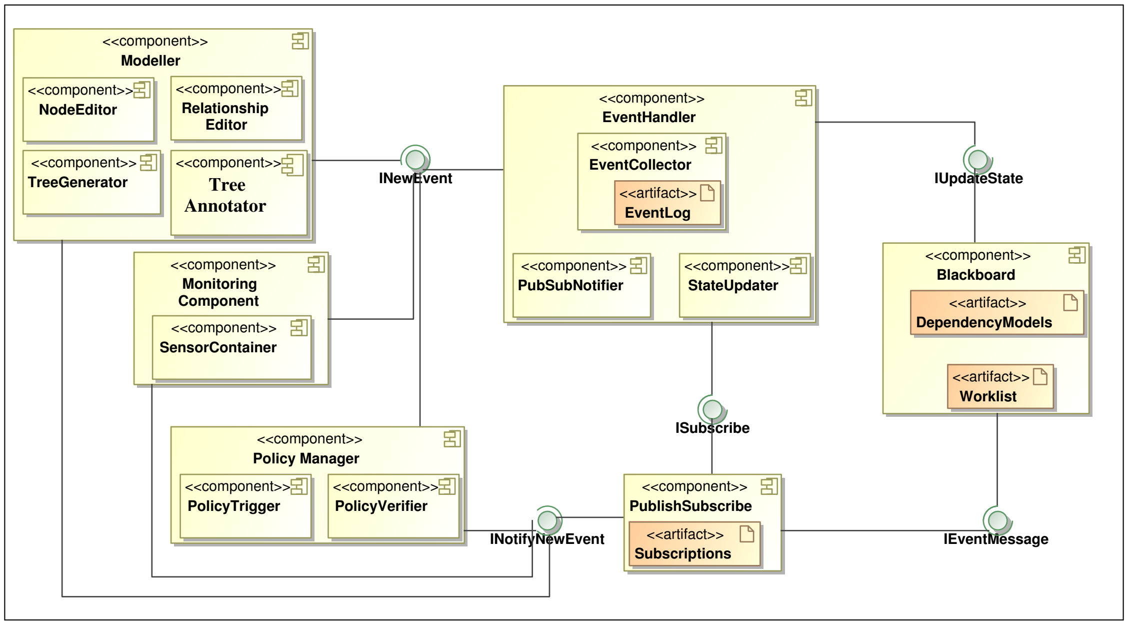


Figure 3.1: Component Diagram of the proposed framework's architecture

and upon the reception of a specific event, it forwards the event to the component it has subscribed for it.

Modeller: This component is responsible for modelling dependencies between entities of the system under maintenance. It creates the Dependency Multi-Graph model and, based on the context of the current maintenance scenario, generates the Dependency Tree model to be used by the *Policy Manager* (see chapter 4).

Monitoring Component: This component uses a set of sensors that monitor the system under maintenance for important information. Such information is needed for verifying or denying the validity of a software maintenance event.

Policy Manager: This component is responsible for the verification or denial of the validity of a software maintenance event. As an example, software maintenance events include the update, removal or addition of a Web Service in Service Oriented Architecture systems. *Policy Manager* leverages information gathered from the *Monitoring Component's* sensors and uses the dependency models produces from the *Modeller* component, in order to perform the validation check.

Next section discusses the aforementioned components in details and describes how these components communicate with each other.

3.2 Description of Components and Interfaces

3.2.1 Blackboard

This is the major component of the framework's architecture. Its main purpose is to store the *state* of the framework. The *state* of the framework refers to the dependency models that have been created and the list of policies that are triggered to be verified. *Blackboard* component stores these two pieces of information into the *DependencyModels* and *Worklist* artifacts, respectively. Next two sections discuss these artifacts and the interfaces that the Blackboard uses, in detail.

Artifacts of the Blackboard

As seen in figure 3.1 the *Blackboard* component consists of two artifacts, namely the *DependencyModels* and the *Worklist* artifact.

The first artifact stores the part of the framework state that refers to the dependency models. Those dependency models are created from the *Modeller* component and passed to the *Blackboard* as events, through the *Event Handler* component. There are two types of dependency models used by the proposed framework, a dependency multi-graph that models all dependencies, and a dependency tree that models only a subset of all the potential dependencies (see chapter 4). *DependencyModels* artifact stores both models, but only the dependency tree can be populated to other components. The reason behind this design decision is that *Policy Manager* component (which cares about the current set of dependencies that should be verified) will perform verification based on the set of dependencies that are relevant to the latest software maintenance operation, i.e. only the dependency tree is needed. However, the dependency multi-graph is stored in the *Blackboard* to allow for comprehensive traceability of the framework's state at each point in time.

The second artifact is a list of "dependencies" (that will be also called "policies") that should be verified in order for the framework to prove the validity of the last performed software maintenance operation. Those dependencies are practically the edges of the dependency tree stored in the *DependencyModels* artifact, annotated with specific strategy that will guide the verification process (see chapter 5). It is the responsibility of the *Policy Manager* to trigger the appropriate verifiers to perform verification of the elements inside the *Worklist*.

Interfaces provided by the Blackboard

Blackboard component provides the *IUpdateState* interface. It is used by the *Event Handler* component when a new event that should change the state of the framework arises. Typically, the interface is used when the *Modeller* generates a new dependency tree from the dependency multi-graph or when the *Policy Manager* adds a new, to-be-verified policy in the *Worklist*.

3.2.2 Event Handler

This component serves as the intermediate of the other components, along with the *PublishSubscribe* component. Its control flow involves the following steps.

1. Collect a new event from other components via the *INewEvent* provided interface.
2. Log this event into an appropriate storage container or database (handled by the *EventLog* artifact).
3. Determine the appropriate recipient of the event, depending on the event's type. There are two options here.
   (a) The event is a request from a component to subscribe to a specific type of events. The event should be forwarded to the *PublishSubscribe* component.
   (b) The event relates to the state of the framework. This type of events should be forwarded to the *Blackboard* component.

Sub-components of the Event Handler

Event Handler consists of three sub-components, namely *Event Collector*, *PubSub Notifier* and *State Updater*.

Event Collector is the sub-component responsible for collecting the events generated by other components and logging them properly, for traceability reasons. This sub-component consists of one artifact, the *EventLog* which represents the logging mechanism used for the recording of the observed events in the system under maintenance. After logging the event, this sub-component forwards it to one of the other two sub-components, based on the type of the event.

PubSub notifier handles case 3a of the aforementioned control flow. It is notified by the *Event Collector* sub-component when a request for subscription to a specific type of events has been received. Subsequently, it forwards this request to the *PublishSubscribe* component, using the *ISubscribe* interface.

StateUpdater is responsible for case 3b of the above control flow. It is notified by the *Event Collector* sub-component when an event that changes the state of the framework has been received. It forwards the event to the *Blackboard* component, through the *IUpdateState* interface.

Interfaces provided by the Event Handler

Event Handler component provides the *INewEvent* interface. This interface is used from *Policy Manager*, *Modeller* and *Monitoring Component* components of the framework. Those components can generate events related to a new state of the framework (change in the dependency models, a new observation about the system under maintenance, a new policy in the work-list) or events that request their subscription to one of the state update event types. When one of those three components has a new event to populate, it does so using the *INewEvent* interface. Afterwards, the event is logged and transfered to the appropriate recipient, as described above.

3.2.3 PublishSubscribe

This component handles subscription of other components to events they are interested into. It receives those subscription request events from the *PubSub Notifier* sub-component of the *Event Handler*. Examples of such request is the subscription of *Policy Manager* to monitoring events (from the *Monitoring Component*) that relate to the verification of policies in the current work-list. It is clear that the *PublishSubscribe* component acts also as a delegate component between the *Blackboard* component and the three main components of the system, namely *Policy Manager*, *Monitoring Component* and *Modeller*. As such a delegate component, it is used to propagate appropriate messages to the three aforementioned components, when the state of the *Blackboard* changes.

It can be argued that subscription types of events are part of the framework's state and should be stored inside the *Blackboard*. However, the current design separates the concern of the event-driven control flow of the framework from the concern of storing the framework's state in a uniform manner. The event-driven control flow is mainly handled by the *Event Handler* and *PublishSubscribe* component, whereas *Blackboard* is responsible for interpreting those events into changes of the framework's state.

Artifacts of PublishSubscribe

PublishSubscribe component consists of one artifact, namely the *Subscriptions* artifact. Generally, it can be seen as a mapping between the interested component and the type of the event this component is interested into. The users of the proposed framework are free to implement their own data structure that models this mapping.

Interfaces provided by PublishSubscribe

PublishSubscribe component provides two interfaces to communicate with other components of the framework:

ISubscribe: This interface facilitates the submission of request events from the *Event Handler* component.

IEventMessage: This interface is used from the *Blackboard* component to propagate state-changing events to the subscribed components (through the *PublishSubscribe* component).

3.2.4 Monitoring Component

This component is responsible for monitoring the system under maintenance and providing valuable information needed for it. Information needed from the system under maintenance may involve resources usage (network, hard disk, physical memory . . .), web service availability (demo calls to web services, polling, ping . . .) or time/space and reliability measurements of the system under maintenance (time elapsed, number of failures . . .). This information is gathered by a set of sensors controlled by the *Monitoring Component* component and encoded into events offered to the other components to the framework.

Sub-components of Monitoring Component

Monitoring Component consists of a *Sensor Container* sub-component which is responsible for handling all the sensors attached to the system under maintenance. Those sensors can be activated/deactivated based on policies in the current work-list. It is the responsibility of the *Sensor Container* to handle these activations and deactivations, whereas *Monitoring Component* interprets events received from other components and generates events targeting other components of the framework.

Interfaces provided by Monitoring Component

This component provides only one interface, namely *INotifyNewEvent* interface. It is the same interface used by *Modeller* and *Policy Manager* components. This interface facilitates the population of events from these three components to the framework's state (*Blackboard*) or to the subscriptions *PublishSubscribe* component stores.

3.2.5 Policy Manager

The triggering of appropriate policies and their verification is the responsibility of the *Policy Manager* component. The term **policy** refers to the algorithm/methodology used to verify or deny the validity of a dependency after a maintenance operation in the system under maintenance has been performed. The verification of the policy may be subject to information gathered by the *Monitoring Component* and is controlled by the dependency tree model created by the *Modeller* component (see chapters 4 and 5).

Policy Manager component is not aware of the dependency multi-graph for the system under maintenance. Instead, it tries to verify the policies of the dependency tree model. The reason behind this decision is the fact that the dependency multi-graph contains all the possible relations between software entities of the system under maintenance. But under a specific maintenance context (e.g. interface compatibility), only one part of those dependencies can cause problem in the system. The generation of the dependency tree saves *Policy Manager* for the effort of checking for every possible relation and allows it to focus on the important relations, based on the context.

Furthermore, *Policy Manager* component is interested into two types of events. The first type is the events that relate to the dependency tree model at a specific point in time. The second type is the events that relate to information gathered by the *Monitoring Component* and can be leveraged to verify or deny policies. It is worth to note that *Policy*

Manager will subscribe only to events that relate to the active policies and not to policies irrelevant to the current state of the verification process.

Sub-components of Policy Manager

Policy Manager encapsulates two sub-components, namely *Policy Trigger* and *Policy Verifier*. The first is responsible for triggering the appropriate policies to be verified. Normally, the set of those policies derives from the dependency tree model produced by the *Modeller* component and stored in the *Blackboard*. The second sub-component is responsible for verifying or denying the policy, by using information from the sensors of the *Monitoring Component*. *Policy Manager* can instantiate a large number of *Policy Verifier* objects at any given point in time. Each *Policy Verifier* is responsible for a set of policies (that will be later called strategy). It is the responsibility of the *Policy Manager* to combine the intermediate results of each *Policy Verifier* to the final result. There are two outcomes in this combination process:

1. All the *Policy Verifiers* agree upon the result. The final result matches the agreeing results (if all the verifiers verify, then the final result is verification, if all the verifiers deny, then the final result is denial).
2. There are controversial results from all the *Policy Verifier* objects. The *Policy Manager* flags the final result as controversial and gives a warning to the user of the framework.

Interfaces provided by the Policy Manager

Policy Manager provides the *INotifyNewEvent* interface, which is explained in section 3.2.4.

3.2.6 Modeller

The purpose of this component is to model the dependencies of the system under maintenance. There are two models used by this framework, a dependency multi-graph and a dependency tree that derives from the multi-graph. Both models are created once, during the framework's initialization. The second one is created when under the assumption of a specific software maintenance context (e.g. service invocation) so it is a subset of the first one. The purpose of the dependency tree is to provide a focused view on the potentially troublesome dependencies, given the context of the last maintenance operation. More information on the two dependency models will be given in chapter 4.

Sub-components of Modeller

The mission of the Modeller component is to model the multi-graph and generate the dependency tree from it. Modelling the multi-graph is done by the *NodeEditor* and *RelationEditor* sub-components, while generating the dependency tree is done by the *TreeGenerator* and *TreeAnnotator* components.

A multi-graph consists of *nodes* and *parallel edges*. Nodes denote specific software artifacts of the system and edges that connect them denote dependencies between the nodes. Hence, the multi-graph consists of two sets, $G(V_G, E_G)$, a set V_G of the nodes (or vertices) and a set E_G of edges between them. The process that *Modeller* follows to construct these two sets is mentioned below.

1. *NodeEditor* sub-component reads the appropriate software artifacts (usually high-level artifacts, such as deployment configuration or SCDL files) of the system under maintenance and produces a list of nodes. Each node corresponds to each of the software entities identified in the system under maintenance.

2. *RelationEditor* sub-component reads the list of nodes created from the previous sub-component and parses again the software artifacts (usually lower-level now, such as implementation artifacts) to identify dependencies between the nodes, hence creating the edges of the multi-graph.

The aforementioned two-step process is general for modelling dependencies and creating the multi-graph for any type of software system. The user of the framework is free to specialize the functionality provided by the *Modeller* to meet the requirements of a software domain of interest. For the purposes of the current work, chapter 6 depicts a prototype usage of the proposed framework, for Service Component Architecture software systems. In the case of SCA systems, the artifacts that *Modeller* uses can be the contribution zip files, the configuration of an SCA domain, or a *SCDL* file, a configuration of a component who produces or consumes web services. The web services that the SCA system offers or consumes can play the role of the software entities modelled by the nodes of the multi-graph. The dependencies that those web services can have with each other (e.g. invocation, data dependencies) can form the edges of the multi-graph. The process is explained in chapter 6, in detail.

The third sub-component of the *Modeller*, *TreeGenerator*, takes into account only a specific subset of all the types of dependencies in the multi-graph. Based on the characteristics of those dependencies and the multi-graph, *TreeGenerator* identifies the related nodes and edges and produces the dependency tree model. The exact algorithm is given in chapter 4.

Last but not least, *Modeller* invokes the *TreeAnnotator* component. The dependency tree produced by the previous component contains conjunctive branches between different edges. However, based on information at the implementation level, certain branches can be annotated as disjunctive, as at runtime the verification of a branch might imply that it is not worth checking the alternative branch. As an example, consider the case where there is a *invocation relationship* between a component C_1, a component C_2 and a component C_3. Let us assume that the first component can call the other two, so there is an edge between C_1 and C_2 and one between C_1 and C_3. *TreeGenerator* component will model these two dependencies via a tree where C_1 is the parent and has two children, C_2 and C_3. The edges will be conjunctive with each other, meaning that the framework must verify that both invocations from C_1 can be made with no problems. Interestingly, if the two invocations are made into two alternative blocks of code (e.g. C_1 calls C_2 if a "boolean" condition is met, or C_3 otherwise) then the system can check only the subtree that applies. This is done when the two branches, and their subsequent sub-trees, have been annotated as disjunctive by the framework.

TreeAnnotator is the *Modeller's* sub-component that transforms certain branches to disjunctive, based on information from implementation artifacts (source code files). Detailed discussion on the *Modeller's* modelling algorithms and the trees will be given in chapter 4.

3.3 Overall Process

Previous sections gave the component diagram of the framework's architecture, figure 3.1, and explained the major components and its inner structure. This section will try to shed light on how the framework works, what is the exact flow of events handled between its components. An example of the process flow of the framework is illustrated in figure 3.2. For simplicity, an initial stage where all the components subscribe for particular types of events is omitted. The process illustrated starts from the moment where the *Modeller* models the dependency multi-graph. It is worth to note that each message shown in figure 3.2 is named after the corresponding interface without the leading "I" (see figure 3.1). The messages of sequence diagram 3.2 are explained one by one in the following list.

1. *Modeller* models the dependency multi-graph and generates a new event captured by the *Event Handler*.

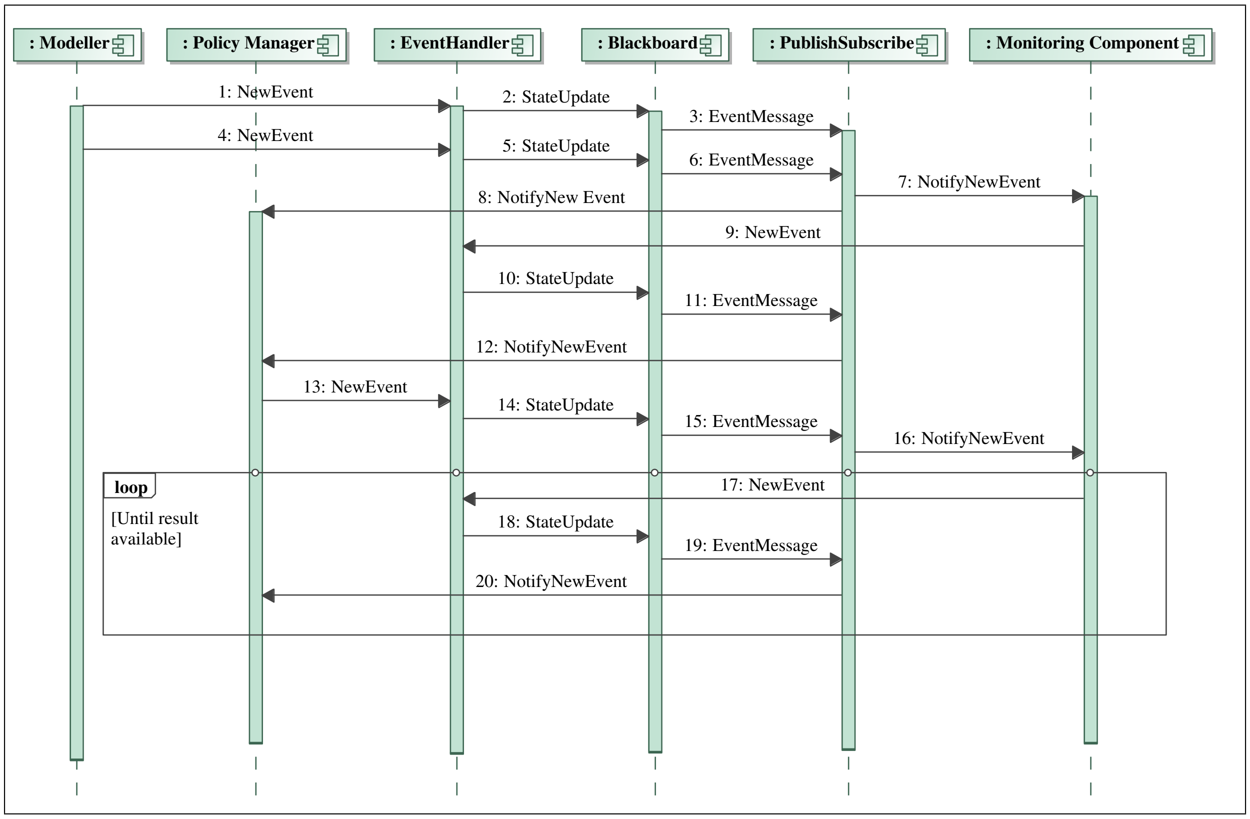


Figure 3.2: Sequence Diagram of the overall process

2. *Event Handler* logs the event and forwards it to the *Blackboard* as a state update.

3. *Blackboard* stores the dependency multi-graph and notifies the *PublishSubscribe* component of the new event. (No component registers for updates in the multi-graph as it is stored only for traceability purposes, so no component is notified from *Publish-Subscribe*)

4. *Modeller* generates a dependency tree from the multi-graph created before. This tree generation is done based on a user-indicated subset of important dependencies. *Modeller* notifies *Event Handler* that a dependency tree is created.

5. *Event Handler* logs the event and forwards the generated tree to the *Blackboard* as a state update.

6. *Blackboard* forwards the state update to the *PublishSubscribe* component.

7. *PublishSubscribe* notifies *Monitoring Component* that a new tree is generated via the *INotifyNewEvent* interface. *Monitoring Component* will enable all sensors that can trace changes to the tree's nodes.

8. *PublishSubscribe* notifies *Policy Manager*, as well. *Policy Manager* requires the information of the dependency tree to initialize policies that will be potentially triggered in the future.

9. *Monitoring Component* produces a new event that a component of the system under maintenance has changes (i.e. a node in the dependency tree has changed). *Event Handler* receives this new events, logs it and forwards it to the *Blackboard* as a state update.

10. *Blackboard* marks the changed node (the dependency tree nod that corresponds to the component changed in the software under maintenance) and notifies the *Publish-Subscribe* component of the change.

11. *PublishSubscribe* receives the information that a node has changed and needs to be verified. For this type of messages, *Policy Manager* is interested, so a notification message to that component is prepared.

12. *PublishSubscribe* sends a new event notification to the *Policy Manager* component. The later inspects the node changed and its *Policy Trigger* sub-component allocates the appropriate *Policy Verifier* sub-components along with the appropriate policies.

13. The information of the policies to be verified is populated from the *Policy Manager* via the *Event Handler* component and the *INewEvent* interface.

14. As usual, *Event Handler* component logs the event (the policies to be triggered) and forwards the state update to the *Blackboard*. *Blackboard* adds the policies into the work-list.

15. Via the *IEventMessage* interface, *PublishSubscribe* is notified about the new additions to the work-list in the *Blackboard*.

16. *PublishSubscribe* notifies *Monitoring Component* about the new additions in the work-list. *Monitoring Component* will enable all sensors that will assist in the verification of the policies in the work-list.

17. This message initiates a loop of messages that will be executed until the final verification of the maintenance operation. The loop starts with a new event from the *Monitoring Component* that updates the framework about information gathered from its sensors. This event is populated to the *Event Handler*.

18. *Event Handler* logs the event and forwards the state update to the *Blackboard* component.

19. *Blackboard* notifies *PublishSubscribe* with the new information from the *Monitoring Component's* sensor(s).

20. *PublishSubscribe* notifies *Policy Manager* component that new information regarding the verification of the policies is available. The appropriate *Policy Verifier* subcomponent will receive the information and try to infer the final result (acceptance of the maintenance operation). Steps 17-20 will be repeated until information for all the triggered policies is gathered and a result from *Policy Manager* is available.

The process will continue with the *Policy Manager* inferring the results from the *Policy Verifier* objects and publishing them to the *Blackboard* which will remove policies from the work-list, as appropriate.

As a conclusion, this chapter introduced a blackboard/repository architecture for the proposed framework and illustrated how this framework works. The major advantage of the framework is that it provides the foundation for dependency analysis on software systems. It can be extended to incorporate a diversity of software system types, as long as the extension provides the appropriate extensions to model the targeted software system with the dependency multi-graph. Another major advantage of the framework is its separation of concerns, due to the message passing interface that was described above.

Chapter 4

Dependency Modeling

This chapter discusses how the proposed framework models dependencies between software artifacts in modern systems. It stands as an introduction to the next chapter that will explain how these models are used for analyzing these dependencies with respect to software maintenance operations.

Software contains dependencies between the entities that it is composed of, regardless of the size of the source code of an application. Single source code files can demonstrate dependencies between functions, e.g. using the same variables. Multiple source code file software projects can demonstrate an even broader range of dependencies, between entities of different source code files. It is also common for software systems to use *configuration artifacts*, such as property files, to fine tune their behaviour. Configuration management like this can inject even more dependencies into large scale software systems. And, of course, this is intensified in the case of distributed software systems, such as service oriented systems or cloud computing applications.

Each and every case of software system can demonstrate a wide variety and diversity of dependencies between its entities. However, those dependencies can have common properties and be classified or handled based on those properties. This is the approach of the current thesis as it tries to model those dependencies by using a common set of properties and leverage those properties in the analysis of the dependencies during the maintenance phase. This chapter is organized as follows.

The first section discusses the *Dependency Multi-Graph Meta-Model* which describes all the possible dependencies between artifacts of a software system. Next, a *Dependency Tree Meta-Model* is described, which contains only the dependencies that are interesting to analyze when checking for a particular attribute of the software system. For example, if the system has to be checked for interface compatibility, then the framework must check for dependencies that have to do with invocation contracts (such as parameters and return types) and not with the usage of configuration files. In order to build this specialized dependency tree from the dependency graph a number of traversal policies are introduced. Those traversal policies lead to the aforementioned dependency tree by using a generation algorithm presented in the end of this chapter.

4.1 Dependency Multi-Graph Meta-Model

The first model used by the proposed framework is a Dependency Multi-Graph that tries to model all types of dependencies that can be found in a software system of interest. Its nodes (or vertexes) are modelling the software components (or artifacts) of the system and its edges the relations between them. Therefore, because of the diversity of ways that components/artifacts can be related/dependent to each other, the multi-graph was selected as the first dependency model. The difference of a multi-graph from a graph is that two nodes can be connected with more than one edge of the same direction and it fits the modelling of the various ways that components can depend with each other in software. Additionally, each multiple edge will be annotated with the exact relation which models how exactly the source artifact depends on the target one.

Formally, the multi-graph used in this framework can be defined using three sets $DMG = \{N_G, E_G, R_G\}$, where:

$\mathbf{N_G}$: The set of nodes that model the software artifacts/components of interest.

$\mathbf{E_G}$: The multi-set of multiple or parallel edges.

$\mathbf{R_G}$: The set of relations. Each relation is an annotation to an edge, specifying how the nodes depend on each other.

This dependency multi-graph is modelled using UML Class diagram, in order for the framework to process it. The *Dependency Multi-Graph Meta-Model* can be seen in figure 4.1. The top-level element is the *MGraph* class which represents the multi-graph. The *MGraph* class is composed of a number of objects of the class *MGraphElement*, as shown by the composition association between these two entities. The composition means that objects of the type *MGraphElement* can exist only as a part of the *MGraph* class, i.e. multi-graph elements cannot exist as autonomous entities outside the scope of a multi-graph. The multiplicity of this association dictates that one multi-graph can be composed of more than

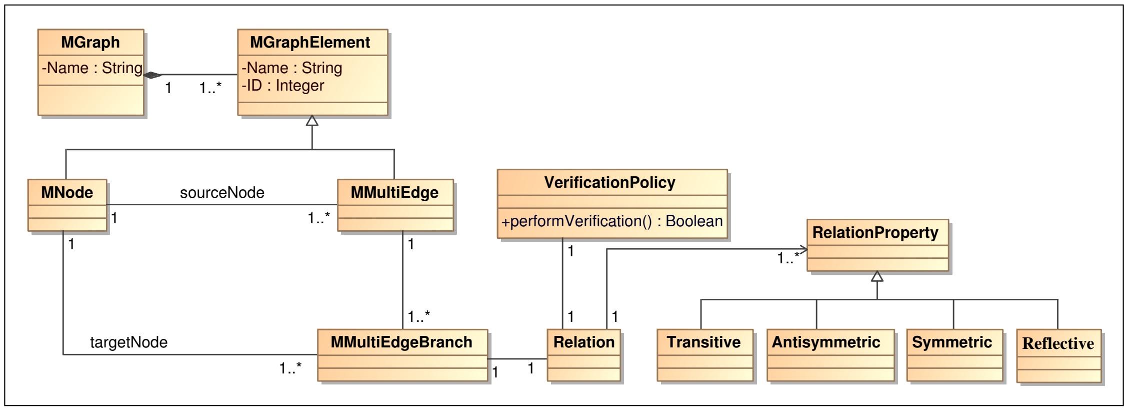


Figure 4.1: Dependency Graph Meta-Model

one multi-graph elements, which is an obvious fact. An *MGraphElement* can be either a node or a multi-edge, modelled by the *MNode* and the *MMultiEdge* classes, respectively.

MNode models a typical node of a graph. In the context of the proposed framework, this class of objects is used to model software entities that can demonstrate dependencies between them. For example, *MNode* can be a function of a source code file, because functions in a program can depend, for example, on the results of other functions. In service oriented software systems, *MNode* can model Web Services that can depend on the results of other Web Services, potentially hosted in different servers. In general, the nodes of a multi-graph model software artifacts that are important in software maintenance (i.e. they can be removed, changed or deleted during a maintenance operation).

MMultiEdge models dependencies of one *MNode* to numerous other nodes. The purpose of this class of objects is to model the parallel edges of multi-graphs. In multi-graphs, two nodes can be connected by multiple edges, and that is why *MMultiEdge* can have one source node (*sourceNode*) and multiple target nodes (*targetNode* via *MMultiEdgeBranch*). The reason behind this design decision is the fact that one software artifact (node) in modern systems might interact in various ways with another software artifact. As a result, the two aforementioned artifacts may have more than one type of dependencies with each other, hence, *MMultiEdgeBranch* is needed to illustrate this fact.

MMultiEdgeBranch depicts one of the parallel edges of the multi-graph. Its purpose is to model one of the (possibly) many dependencies that two nodes have with each other. As a result, it is the element that associates with the appropriate information for the verification of the validity of the dependency during checking. In the model of figure 4.1, *Relation* is the element which models dependencies between two software artifacts (nodes). Each *MMultiEdgeBranch* associates with only one *Relation* as explained above.

Relation class is the general class that models the possible types of dependencies two nodes can have. Such dependencies can be invocation, data dependency etc. Generally, the framework doesn't have to be aware of the exact type of the dependency or its semantics for the designer of the system under maintenance. Instead, it has to be aware for a set of general type of properties those dependencies should have.

These properties are modelled using the *RelationProperty* class, which can be one of the following four:

Reflective: This type of *Relation* models self-dependency that a software component might demonstrate. A practical example of such a dependency is recursive calls, where a function invokes itself with another argument. This type of *Relation* is of no significant interest during software maintenance (because when a node is changed it is highly unlikely that it will cause problems to itself) but this *RelationProperty* exists in the model for completeness.

Antisymmetric: This type of *Relation* models directed, "one-way" dependencies. If there are two components, c_1 and c_2, and an antisymmetric *Relation* r_a, then if there is a dependency $r_a(c_1, c_2)$ the dependency $r_a(c_2, c_1)$ cannot exist. The framework leverages this type of *Relation* in specifying a one-way verification strategy for the *VerificationPolicy* attached to the *Relation*.

Symmetric: This type of *Relation* models bidirectional dependencies. If there are two components, c_1 and c_2 and a symmetric *Relation* r_s then the existence of the dependency $r_s(c_1, c_2)$ guarantees the existence of the dependency $r_s(c_2, c_1)$. The framework applies two-way verification strategy for this type of *Relation*.

Transitive: This is the type of dependency that can be transferred to the target nodes. If there are two components, c_1 and c_2 and two dependencies $r_t(c_1, c_2)$ and $r_t(c_2, c_3)$, then component c_1 is dependent to component c_3 with the same type of dependency, i.e. $r_t(c_1, c_3)$. The framework links the verification policies of those type of dependencies, as the next sections will demonstrate.

Figure 4.1 shows and association of the *Relation* class to objects of the type *VerificationPolicy*. These objects are responsible to verify or deny the dependency modelled by the edge they are attached to. The aforementioned *RelationProperty* is used by the framework to attach the correct verification strategies that will govern the correct triggering of *VerificationPolicy* objects. Those strategies can be seen in the next dependency model.

4.2 Dependency Tree Meta-Model

Figure 4.2 depicts the Dependency Tree Meta-Model. Its purpose is to model Dependency Trees that offer a view of the relations between software artifacts given a specific context during the maintenance phase. Section 4.1 defined the Dependency Multi-Graph as three sets.

The dependency tree that derives from a dependency multi-graph can be also defined with three sets $DT = \{N_T, E_T, R_T\}$, where N_T is the set of nodes, E_T is the set of edges (two nodes can be connected with only one edge now) and R_T is the set of relations attached to each edge. Generally, it can be safely stated that equations (4.1a), (4.1b) and (4.1c) are all true for every pair of dependency multi-graph and a (derived from it) dependency tree. It's worth to note that equations (4.1b) and (4.1c) dictate that the sets of edges and relations of the tree should not be equal to their counterpart sets of the multi-graph. The purpose for this decision is that the tree models one portion of the dependencies modelled by the multi-graph. So, it will include a portion of multi-graph's E_G and R_G sets.

$$N_T \subseteq N_G \tag{4.1a}$$
$$E_T \subset E_G \tag{4.1b}$$
$$R_T \subset R_G \tag{4.1c}$$

The Dependency Tree consists of *DTNodes*. Each node corresponds to its counterpart node of the multi-graph. Each node of the tree can be either an *AtomicNode* or a *CompositeNode*. *AtomicNodes* do not have outgoing edges with any of the interesting dependencies in the multi-graph. On the other hand, *CompositeNodes* depend on other nodes of the multi-graph. In figure 4.2 the *container* design pattern, [34], is used for modelling the *CompositeNode* which composes of other (atomic or composite) nodes.

The composition of a *DTNode* from other *DTNodes* can have two meanings. The first is *Disjunction* and the second is *Conjunction*. In the first case, the parent *DTNode* depends on only one of the children *DTNodes* at a specific point in time. In the second case it depends on all the children *DTNodes*. This disjunction/conjunction dependency of the parent to its children nodes is modelled via the *BranchType* class which is realized by the *Disjunction* and *Conjunction* classes.

Similarly to the multi-graph, tree edges can be annotated with *VerificationPolicies* that implement the appropriate function to verify or deny the dependency. However, the tree is used for guiding the process of the verification, and therefore, it must contain the appropriate information. This information is modelled in the *VerificationPolicyControlStrategy*.

Instances of this class are attached to *VerificationPolicy* objects. There are three different types of *VerificationPolicyControlStrategy* objects, each one corresponding to one of the *RelationProperty* found in the aforementioned multi-graph, section 4.1.

Unidirectional: It corresponds to the *Antisymmetric* type of *RelationProperty* objects. This verification strategy checks the directed edge, from the source node to the target node. This is the reason why it associates with only one *VerificationPolicy*.

Bidirectional: Its counterpart is the *Symmetric RelationProperty*. If a dependency is symmetric, then there are two policies to be checked, bidirectionally from source to target node and the opposite, in order for symmetric dependency to be verified. As a result, the corresponding type of strategy is associated with another one *VerificationPolicyControlStrategy* object, whose *VerificationPolicy* object is responsible to verify the "opposite" relation.

Linked: It serves as the verification strategy for *Transitive RelationProperty* objects. For the verification strategy, the policies are linked in a doubly linked list. The reason is that if a dependency is transitive, then the framework must check all the dependency edges that relate to the transitive closure of the *Transitive* relation on the multi-graph.

More on the verification algorithm and the role that the aforementioned strategies play in it, will be given in chapter 5.

4.3 Traversal Policies

Figures 4.1 and 4.2 show two object classes, *RelationProperty* and *VerificationPolicyControlStrategy* that highly relate with each other. They are used for guiding the generation of the dependency tree from the multi-graph and guiding the verification process of the policies, respectively. This section will show how *RelationProperty* objects guide the traversal of the multi-graph, in order to produce a dependency tree. Next section will demonstrate the process by providing the algorithm and discussing examples of dependency tree generation.

4.3.1 Reflective

A relation is reflective according to the following definition.

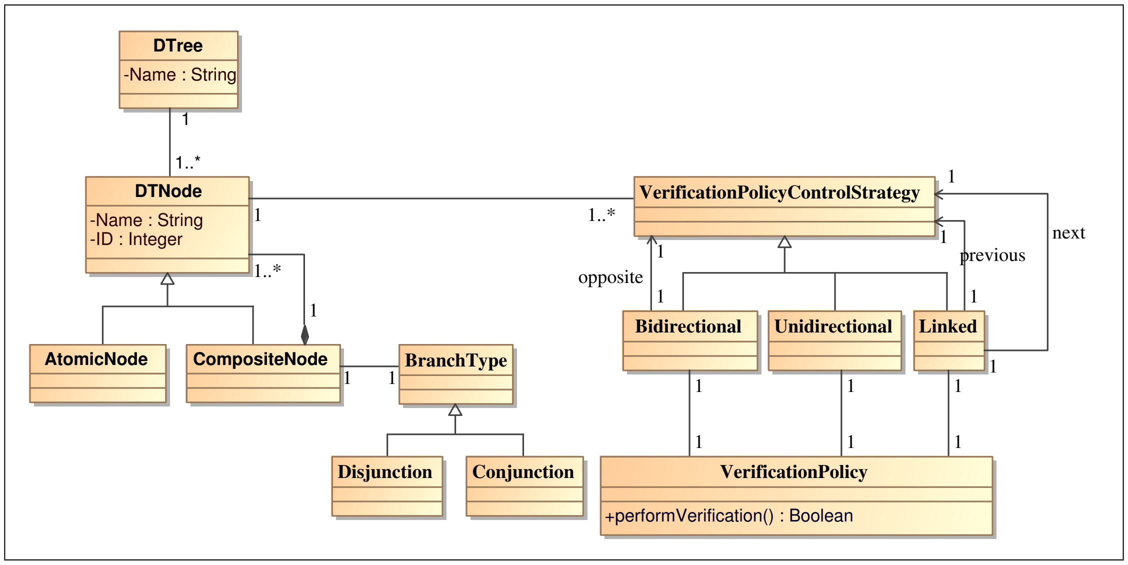


Figure 4.2: Dependency Tree Meta-Model

Definition 4.1 *Assuming $n_1 \in N_G$ is the node and $r^r \in R_G$ the reflective relation, then $r^r(n_1, n_1)$ is true if node n_1 depends on itself.*

The above case is not important for maintenance operations, as explained before. For the completeness of verification algorithms, its case is handled as the case of an antisymmetric relation, presented below.

4.3.2 Antisymmetric

A relation between two nodes is antisymmetric according to the following definition.

Definition 4.2 *Assuming $n_1, n_2 \in N_G$, are the nodes and $r^{as}(n_1, n_2) \in R_G$ the antisymmetric relation, then $r^{as}(n_2, n_1) \notin R_G$.*

The above definition means that only node n_1 depends on node n_2 and not the opposite. During the multi-graph's traversal this means that only the edge $e(n_1, n_2)$ must be found in the resulting tree. Moreover, the appropriate verification strategy will be "unidirectional" from node n_1 to node n_2. Therefore, when the traversal algorithm visits such an edge in the multi-graph, it adds the edge $e^{as}(n_1, n_2)$ and a *Unidirectional* verification strategy is attached to it.

4.3.3 Symmetric

A relation between two nodes is symmetric according to the following definition.

Definition 4.3 *Assuming $n_1, n_2 \in N_G$ are the nodes and $r^s(n_1, n_2) \in R_G$ the symmetric relation, then $r^s(n_2, n_1) \in R_G$.*

As a result, when there is a symmetric dependency, if node n_1 depends on node n_2, then node n_2 depends on node n_1, as well. For the graph traversal process, there will be only one edge $e^s(n_1, n_2)$. However, the algorithm will attach a bidirectional verification strategy on that edge. As a result, the underlying framework will have to check both the dependency of node n_1 on node n_2, as well as the opposite dependency. This will be done by using the two *VerificationPolicyControlStrategy* objects that should be linked with each other.

4.3.4 Transitive

A relation between two nodes is transitive according to the following definition.

Definition 4.4 *Assuming $n_1, n_2, n_3 \in N_G$ are the nodes and $r^t(n_1, n_2), r^t(n_2, n_3) \in R_G$ the transitive relations, then $r^t(n_1, n_3) \in R_G$.*

Transitive relations depict a more profound dependency. They show dependencies that can span throughout the different components modelled in the multi-graph. In the verification context, a transitive relation means that the strategy must cover the full path of edges of the same transitive relation. As an example, let us consider a set of $n_1, n_2, n_3, n_4, \ldots, n_m$ nodes, a set of $e_1, e_2, e_3, e_4, \ldots, 3_{m-1}$ edges and the transitive relation r^t. The transitive dependencies can be seen in formula (4.2).

$$n_1 \xrightarrow[r^t]{e_1} n_2 \xrightarrow[r^t]{e_2} n_3 \xrightarrow[r^t]{e_3} n_4 \xrightarrow[r^t]{e_4} n_5 \ldots n_{m-1} \xrightarrow[r^t]{e_{m-1}} n_m \tag{4.2}$$

In the event that one of the above nodes changes due to a maintenance operation, the verification strategy should check all the above path of edges, as well as the transitive closure, in order to verify the validity of the change.

4.4 Dependency Tree Generation

Previous sections of the present chapter discussed and described the meta-models for the dependency multi-graph and the dependency tree. As stated previously, the multi-graph models all the dependencies that nodes of a system can have. The dependency tree allows for a specific view of those dependencies, with respect to the verification analysis to be done for the system under maintenance. This chapter describes how the dependency tree derives from the dependency multi-graph and gives examples of the derivation process.

4.4.1 Algorithm

Algorithm 1 Dependency Tree Generation from Dependency Multi-Graph

```
Input: MultiGraph $g$, InitialNode $n_S$, RelationSet $relations$
Output: Tree $t$
1: $t.add(n_S)$
2: for all $e \in n_S.getMultiEdges$ do
3:   for all $b \in e.getBranches$, where $((r = b.getRelation) \in relations$ and $b.getTarget \notin visited)$ do
4:     if r.getRelationProperty=(Antisymmetric or Reflective) then
5:       $t.createEdge(n_S, b.target)$
6:       $attachUnidirectionalStrategy(b.getPolicy, n_S, b.target)$
7:     else if r.getRelationProperty=Symmetric then
8:       $t.createEdge(n_S, b.target)$
9:       $attachBidirectionalStrategy(b.getPolicy, n_S, b.target)$
10:    else if r.getRelationProperty=Transitive then
11:      $t.createEdge(n_S, b.target)$
12:      $attachLinkedStrategy(b.getPolicy, n_S, b.target)$
13:      $treegen(g, b.target, relations)$
14:      $markVisited(b.target)$
15:    end if
16:    return $t$
17:  end for
18: end for
```

Algorithm 1 illustrates an abstraction of the process that is followed to generate a dependency tree from the dependency multi-graph. Input of the algorithm is the multi-graph, an initial node of it and a set of relations. Output is the generated tree. Essentially, the algorithm iterates over all edges of the initial node that are annotated with one of the relations found in the *relations* set. If such an edge exists, then a new edge between the initial node and the target node is created. Function *createEdge* is responsible for adding the new edge to the tree, along with the new node. After adding the new node, a verification strategy is attached to the policy that verifies the edge. As explained in section 4.3, there are three options in this case.

1. If the relation is *Reflective*, the attached policy is *Unidirectional.*
2. If the relation is *Antisymmetric*, the attached policy is *Unidirectional.*
3. If the relation is *Symmetric*, the attached policy is *Bidirectional.*
4. If the relation is *Transitive*, the attached policy is *Linked.*

In case number 4 there is one more step needed. As mentioned in section 4.3 all nodes that are connected with the target node using the same transitive relation should be taken into account. As a result, the same algorithm is recursively executed with the same relation set. Function *attachLinkedStrategy* assures that if the strategy of the incoming edge to source node is of type *Lined*, then the newly formed strategy will be linked to the doubly-linked list.

Disjunctive Nodes

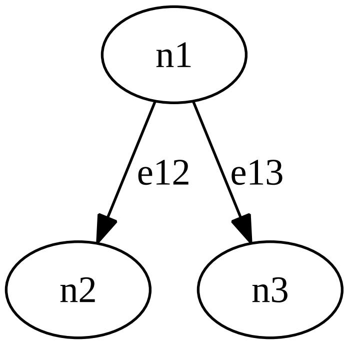
The Dependency Tree Meta-Model of figure 4.2 contains a feature that algorithm 1 does not take into account. This feature is the *Disjunction* or *Conjunction BranchType* that a *CompositeNode* can have in the dependency tree. This feature will assist the framework to free resources when there is no need to check one of the two subtrees in a disjunctive *CompositeNode*, as will be shown in chapter 5. All *CompositeNodes* created by algorithm 1 are of type *Conjunction. TreeAnnotator* sub-component of *Modeller* is responsible to attach *Disjunction* nodes, when this is applicable. As an example, assuming one node n_1 that invokes methods offered by two other nodes, n_2 and n_3. Therefore, node n_1 depends on the other two, via two edges $e_{12}(n_1, n_2)$ and $e_{13}(n_1, n_3)$ and one type of relation/dependency, $r_{invokes}$. Also, let us assume that the code in question of node n_1 looks like in figure 4.3.

It is clear that at runtime, only one of the dependencies depicted by edges e_{12} and e_{13} will be present in the system, based on the value of the boolean expression "b". In the above example, algorithm 1 will create node n_1 as a *CompositeNode* with *Conjunction BranchType. TreeAnnotator* will identify, based on implementation information, that the

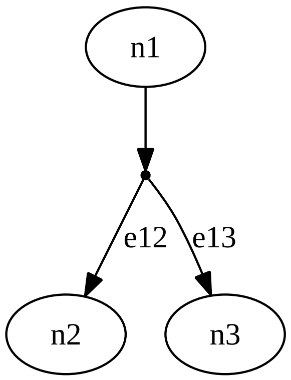
```
function n1(boolean b){
        if(b){
                n2()
        }else{
                n3()
        }
}
```

Figure 4.3: Code sample for disjunctive dependency

dependencies are exclusive with each other and change the *BranchType* to *Disjunction*. Figure 4.4a shows the tree as derived from the *TreeGenerator* and figure 4.4b the tree after being processed by the *TreeAnnotator* sub-component. The former sub-component identifies the two edges as exclusive with each other and creates a *helper CompositeNode* that is unnamed, of type *Disjunction* and represented by a '•' symbol.



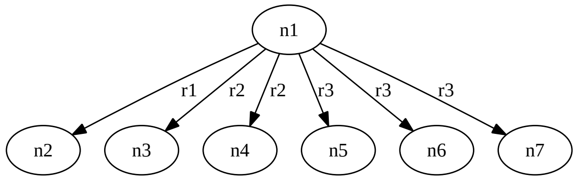
(a) Original Tree - Conjunction



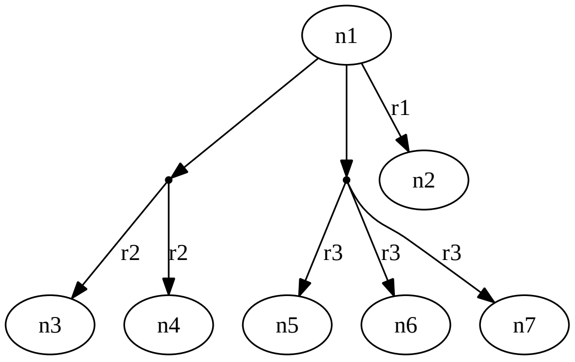
(b) Tree after TreeAnnotator execution - Disjunction

Figure 4.4: Conjunction to Disjunction node annotation from TreeAnnotator

In a more complex example, let us examine the case of node n_1, when it has the following relations with nodes $n_{2\ldots7}$: $r_1(n_1, n_2), r_2(n_1, n_3), r_2(n_1, n_4), r_3(n_1, n_5), r_3(n_1, n_6), r_3(n_1, n_7)$. Let us assume that multiple instances of relations r_2 and r_3 are exclusive with each other, i.e. n_1 can depend only on n_3 or n_4 for r_2 and only one of n_5, n_6, n_7 for r_3. *TreeAnnotator* should create two helper *Disjunction* nodes, one with nodes n_3 and n_4 as its children and one with nodes n_5, n_6 and n_7 as its children nodes. The two aforementioned helper nodes and node n_2 should be places as children of node n_1. This is illustrated in figure 4.5.



(a) Original Tree - Conjunction



(b) Tree after TreeAnnotator execution - Disjunction

Figure 4.5: Conjunction to Disjunction node annotation from TreeAnnotator - Complex Example

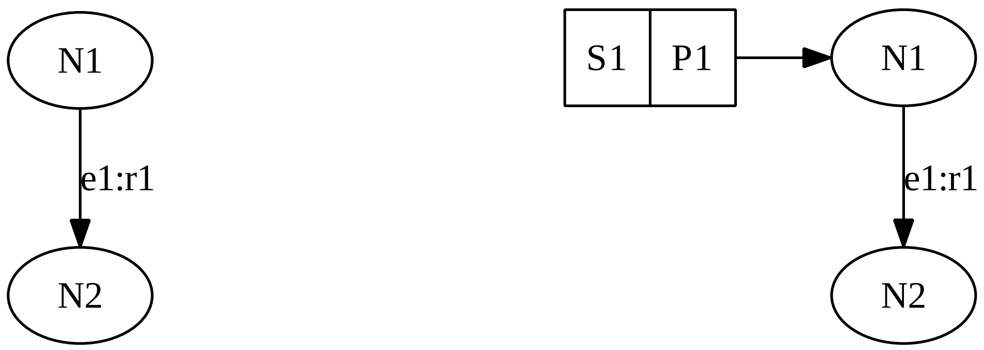
Next section will try to explain the aforementioned algorithm 1 by providing examples of tree generations from multi-graphs.

4.4.2 Tree Generation Examples

This section will try to explain the process of the goal tree generation, by using examples. First, three examples of simple cases will show how strategies are attached to dependency tree. Then, complete examples will be given to demonstrate the generation of a tree from a multi-graph of 10 nodes.

Simple Cases

Antisymmetric Relation. As a first simple example, let us consider a simple software system that consists of two components, n_1 and n_2. The two components relate with each other with the antisymmetric relation r_1, $r_1(n_1, n_2)$. The multi-graph for such a system can be seen in figure 4.6a. The resulting tree can be seen next to it, in figure 4.6b. There is only one edge from node 1 to node 2. The rectangle named "S1" is the *VerificationPolicy-ControlStrategy* object attached to node n_1 (which depends on node n_2). Additionally, the rectangle named “P1” is the *VerificationPolicy* attached to the aforementioned strategy object. In this simple case there is only one *VerificationPolicy* attached to a *Unidirectional* strategy. This policy will check for the dependency of node 1 on node 2.



(a) Graph (b) Tree

Figure 4.6: Multi-Graph and derived tree for system with one antisymmetric relation

Symmetric Relation. As a second example, let us consider the same simple software system with the two components. The only thing that differs from the previous case is that the two components are connected with a symmetric relation r_2, $r_2(n_1, n_2)$. In this case, the *VerificationPolicyControlStrategy* should maintain a link between the “straight” $(n_1 \rightarrow n_2)$ and the “opposite” $(n_2 \rightarrow n_1)$ *VerificationPolicy* objects. The multi-graph and the resulting tree in this case are illustrated in figures 4.7a and 4.7b, respectively. It is worth to note that in the case of symmetric relation there are two *VerificationPolicyControlStrategy* objects created, one for node 1 and one for node 2. *VerificationPolicyControlStrategy* “S1” that belongs to node 1 associates with *VerificationPolicy* “P1” which checks the dependency of node 1 on node 2. It also associates with *VerificationPolicyControlStrategy* “S2”, whose attached *VerificationPolicy* “P2” is responsible to check the dependency of node 2 on node 1 (the “opposite”). The exact opposite situation is true for *VerificationPolicy-ControlStrategy* “S2” which has *VerificationPolicy* “P2” as the “straight” one, and “P1” as the “opposite”. *Note:* Dotted lines in figure 4.7b depict *opposite* association between two *Bidirectional VerificationPolicyControlStrategy* objects; see figure 4.2.

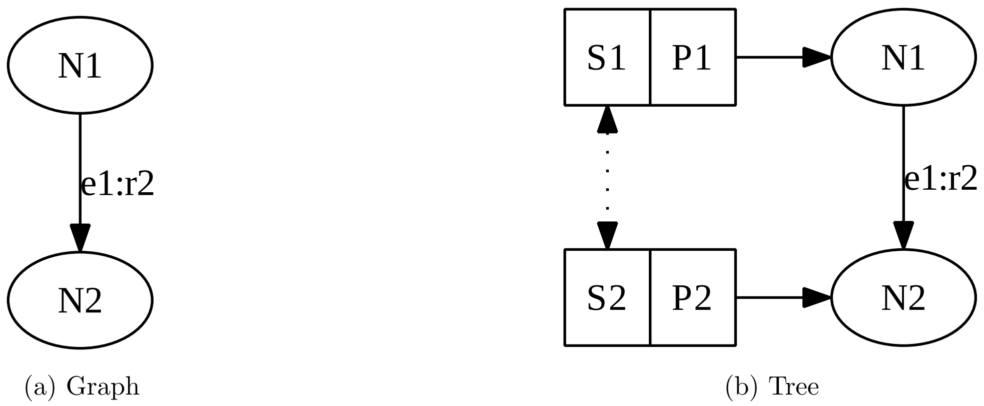
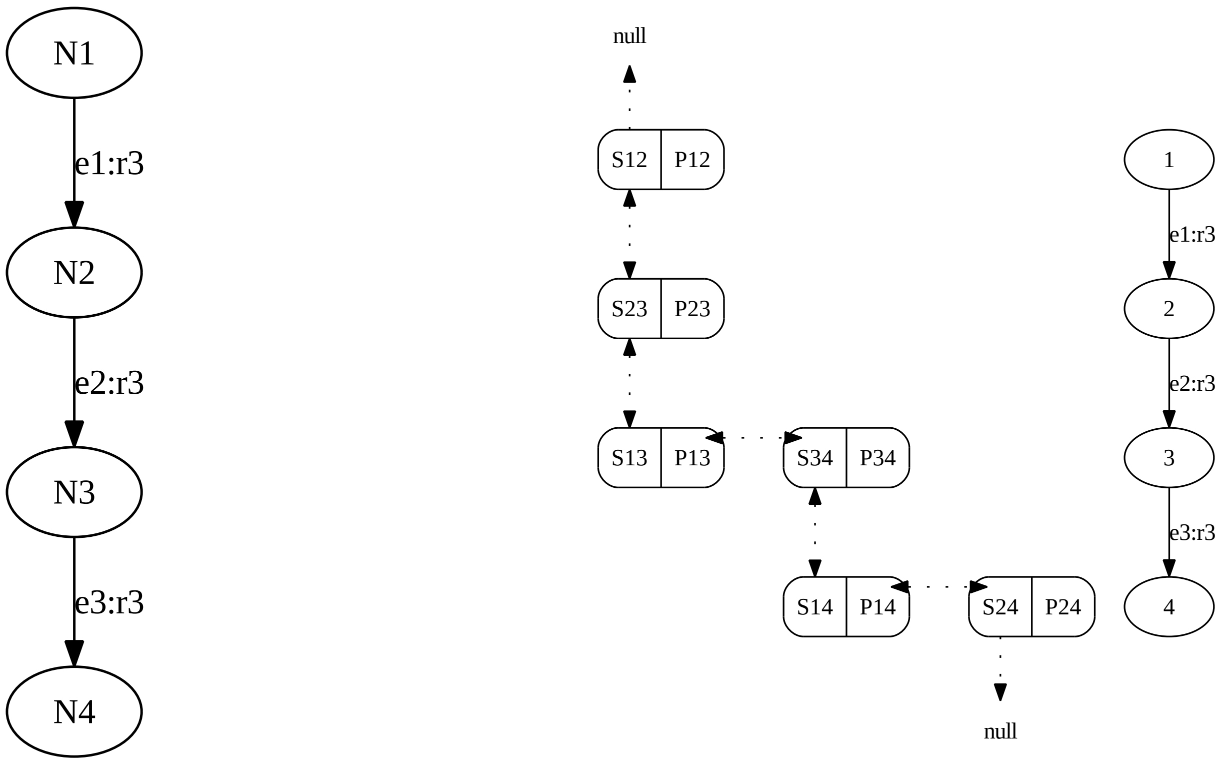


Figure 4.7: Multi-Graph1 and derived tree for system with one symmetric relation

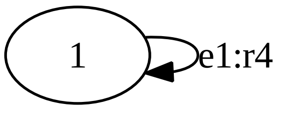
Transitive Relation. The third simple example involves a system whose components all relate via a transitive dependency. The system consists of four components that all relate with each other with the transitive relation r_3, as shown in figure 4.8a. The traversal of this multi-graph results in the tree shown in figure 4.8b. The notation for this example is a little different from the previous figures, because of its complexity. For example, *VerificationPolicyControlStrategy* "S12" refers to the dependency that node 1 and node 2 have. Furthermore, links between nodes and policy/strategy objects are not shown, but each strategy/policy bundle corresponds to the node in the same height in the figure, e.g. "S13" and "S34" belong to node n_3. It is worth noting that there are some *VerificationPolicyControlStrategy* and *VerificationPolicy* objects like "S13" and "P13" refer to imaginary edges that can be found in the transitive closure (e.g. node 1 and node 3 are not connected in the multi-graph or tree, but they relate because of the transitivity of the relation r_3).



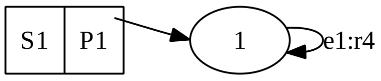
(a) Graph (b) Tree

Figure 4.8: Multi-Graph and derived tree for system with one transitive relation

Reflective Relation. The forth example involves a software component which has a dependency on itself. An example of such a dependency is a recursive call. As seen in figure 4.9a, component 1 has a recursive relation with itself, r_4. The tree generated in this case is similar to the one illustrated in figure 4.9b. The attached *VerificationPolicy-ControlStrategy* is *Unidirectional* and contains the proper *VerificationPolicy* to check the recursive call (or the reflective relation, for that matter).



(a) Reflective Relation - Graph



(b) Reflective Relation - Tree

Figure 4.9: Multi-Graph and derived tree for system with one reflective relation

Complete Tree Generation Example

Previous section illustrated the simple cases of tree generation when the system contains only one relation and there are a few (2 or 4) components. This section will provide a complete example of tree generation from a multi-graph. The setting for this example is as follows.

Let us assume a hypothetical software system S which consists of ten components, $C_1, C_2, \ldots, C_{10}$. The types of dependencies that components of this software system can have are six and can be found in table 4.1. There are two symmetric relations (r_1 and r_2), two transitive relations (r_3 and r_4) and two antisymmetric relations (r_5 and r_6). Additionally, the dependency multi-graph for the aforementioned hypothetical software system is illustrated in figure 4.10.

In this example, algorithm 1 is called with the same multi-graph of figure 4.10, node n_2 and the relation set $R = r_2, r_4, r_6$ as the input. The output in this case is illustrated in figure 4.11.

For figure 4.11 only the strategies are presented and the corresponding policies attached to it are not illustrated, but they exist in the actual model. Each strategy is named after the source and target node in the multi-graph of figure 4.10 as $S\{X\} - \{Y\}$. For example, *VerificationPolicyControlStrategy* named $S6 - 7$ refers to the dependency that nodes 6 and 7 have and is modelled with edge e_7. If there are more than one *VerificationPolicyControlStrategy* with the same source and target nodes, then a symbol is appended to the name, for illustrative purposes. This happens with nodes 2 and 6. They relate directly via edge e_6, but also indirectly via the transitive closure of the path $n_2 \xrightarrow[r_4]{e_1 1} n_5 \xrightarrow[r_4]{e_1 2} n_6$. Therefore, the second relation is named $S2 - 6a$.

The *VerificationPolicyControlStrategy* objects are attached to the nodes in the same "row". If it is not obvious from the figure, then arrows connect the *VerificationPolicyControlStrategy* object with the corresponding node. This is done for nodes n_7 and n_{10}. Furthermore, adjacent *VerificationPolicyControlStrategy* objects denote that they are part

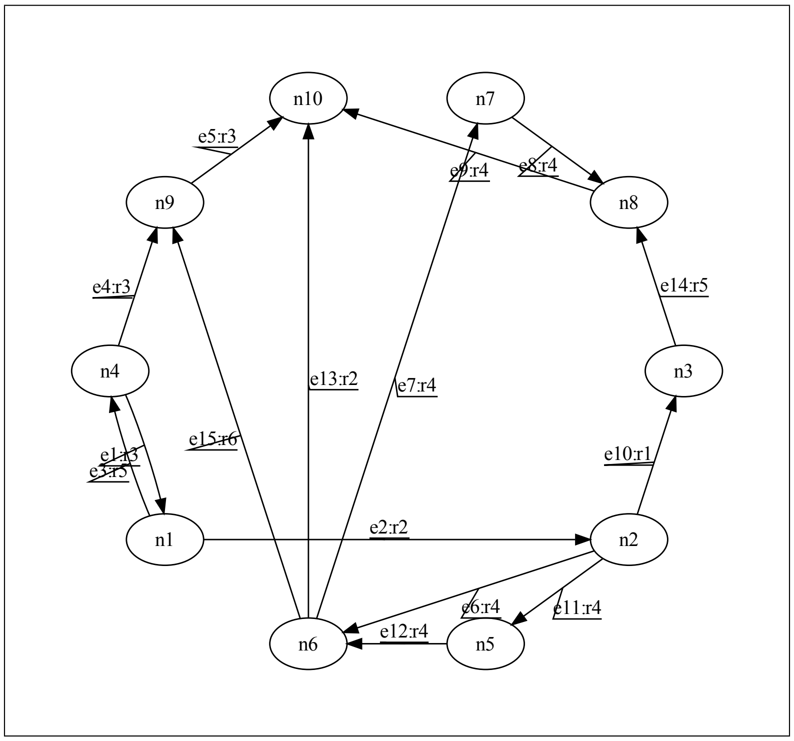


Figure 4.10: Dependency Multi-Graph for the “complete” example

Table 4.1: Relations used for the tree generation examples

Relations	
Name	Property
r_1	Symmetric
r_2	
r_3	Transitive
r_4	
r_5	Antisymmetric
r_6	

of the list, e.g. S2-7, S5-7 and S7-8 they are all part of the *Linked VerificationPolicyControlStrategy* list for the transitive closure of relation r_4. S5-7 is the "next" *VerificationPolicyControlStrategy* of S2-7, and the "previous" of S7-8 etc.

This example contains examples from all three major categories of relation types between nodes. *VerificationPolicyControlStrategy* $S6 - 9$ is *Unidirectional* because of the antisymmetric nature of relation r_6. There are two *Bidirectional VerificationStrategyControlPolicy* objects, namely $S6 - 10$ and $S10 - 6$. The reason for that is the dependency between nodes n_6 and n_{10} via the edge e_{12}. The relation modelled with this edge is r_2 and according to the table 4.1, relation r_2 is symmetric.

Additionally, here is a big linked list of *VerificationPolicyControlStrategy* objects. This is a result of the path linked with the *transitive relation* r_4, $n_2 \xrightarrow[r_4]{e_{11}} n_5 \xrightarrow[r_4]{e_{12}} n_6 \xrightarrow[r_4]{e_7} n_7 \xrightarrow[r_4]{e_8} n_8 \xrightarrow[r_4]{e_9} n_{10}$. The aforementioned lined list contains all the implicit and explicit relationships that are found in the transitive closure of the path.

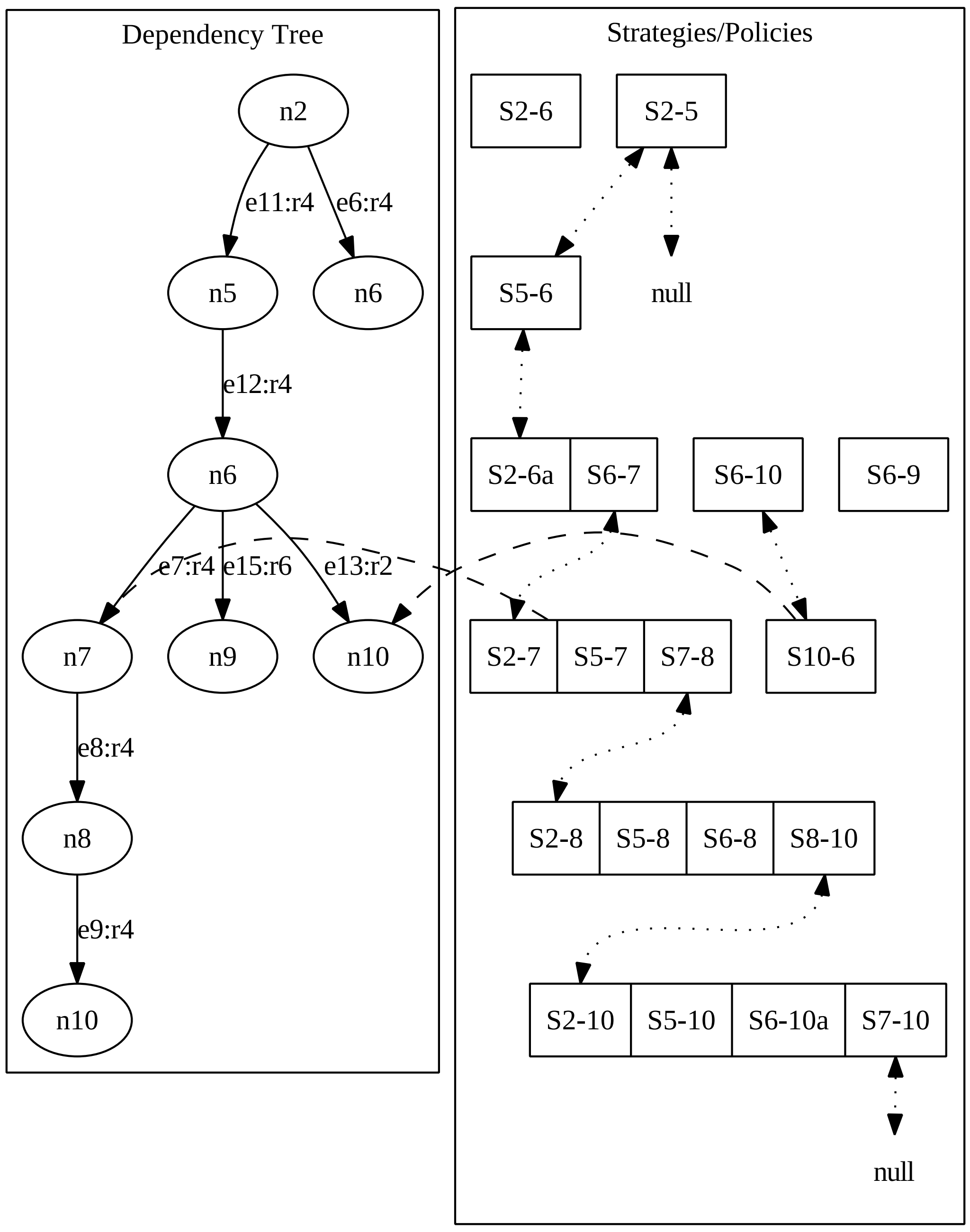


Figure 4.11: Resulting dependency tree for the “complete” example

Chapter 5

Dependency Analysis

Chapters 3 and 4 explained how the proposed framework is structured and how it models dependencies between the artifacts of a software system. This chapter discusses how this dependency modelling can be leveraged to check consistency of maintenance operations (artifact update).

Generally the process of the dependency analysis consists of two steps. The first step is to identify the affected portion of the dependency tree model in the event of a change in the software system. During this step, the framework decides which policies should be triggered in order to verify dependencies that are possibly affected by the aforementioned change. Based on the type of the *VerificationPolicyControlStrategy* objects of the portion that is changed, an appropriate number of policies is triggered and put in the work-list.

The second step is to trigger the appropriate sensors or artifacts that have the ability to verify or deny the policies in the current work-list. In general, this process is handled by the *Policy Manager* and the *Monitoring Subsystem. Policy Manager* guides the process of the policies verification, while *Monitoring Subsystem* leverages its sensors to feed the framework with information that can help the verification process.

This chapter presents the algorithms associated with the aforementioned steps and gives an example of their execution, based on the hypothetical software system of chapter 4.

5.1 Dependency Tree Analysis

5.1.1 Policy Triggering

The approach that the framework follows to trigger new policies to be verified is very simple. As mentioned earlier in this text, nodes of the dependency tree denote artifacts of the software system and edges dependencies between these artifacts. Upon the update of one artifact, the framework identifies the node of the dependency tree that was changed. Afterwards it gathers all its incoming edges, i.e. all the components that depend to it, and all the *VerificationPolicy* objects attached to it, i.e. all the dependencies the changed node has to other nodes/components. For each incoming edge there are three possibilities, based on the type of the relation that the edge models.

Antisymmetric: When an edge is antisymmetric, then only one policy is triggered, the one that verifies the dependency between the source and the target node.

Symmetric: When an edge is symmetric, then the verification strategy assures that there will be two policies triggered, one to check the dependency of the source node to the target node, and one for the opposite dependency.

Transitive: As mentioned in chapter 4, transitive relations are all linked into one doubly-linked list of verification strategies. If an incoming or outgoing edge of the changed node belongs into such a nested strategy, then the full path of the policy "nest" should be taken into account during the verification process.

Additionally, the framework triggers all *VerificationPolicy* objects that are attached to the node which represents the component that changed. In this case, there is the additional complexity of the *Disjunction* helper nodes that are children of the changed node. In such a case, the framework should perform *path selection*, in order to choose which *VerificationPolicy* in a *Disjunction* to be triggered. Generally, this involves knowledge of the system at runtime, like variable values. In algorithm 2, function *pathSelection* is responsible to choose which *VerificationPolicy* to be triggered. Algorithm 2 depicts how *VerificationPolicy* objects are inserted into the framework's work-list, when the component modelled by node n_c, has changed. The algorithm starts by triggering all the *VerificationPolicy* objects attached to the node which models the changed component. In this case, if a *VerificationPolicy* (which is attached on a *VerificationPolicyControlStrategy* object) is on a *Disjunction* helper node, then path selection is performed, to choose which one will be triggered. Otherwise, the attached *VerificationPolicy* is triggered. Second part of algorithm 2 triggers

Algorithm 2 Policy Triggering

```
Input: Tree $t$, NodeChanged $n_c$
Output: Correct VerificationPolicy Objects are triggered
 1: for all VerificationPolicyControlStrategy $v \in n_c.getStrategies()$ do
 2:     if $v$ belongsTo Disjunction node then
 3:         pathSelection($v$)
 4:     else
 5:         triggerHelper($v$)
 6:     end if
 7: end for
 8: for all edge $e \in (n_c.incomingEdges)$ do
 9:     triggerHelper($e.getSourceNode()$)
10: end for
```

all the *VerificationPolicy* objects attached to parents of the "changed" node. Path selection in this case is not required, because the correct path is dictated by the "changed" node. Algorithm 1 shows how *triggerHelper* function works in triggering the appropriate *VerificationPolicy* objects, based on the type of the *VerificationPolicyControlStrategy*. In

Algorithm 3 Policy Triggering - Helper

```
Input: VerificationPolicyControlStrategy $v$
Output: Trigger VerificationPolicy objects
 1: if $v$ = Unidirectional then
 2:     trigger($v.getPolicy()$)
 3: else if $v$ = Bidirectional then
 4:     trigger($v.getPolicy()$)
 5:     trigger($v.getOpposite().getPolicy()$)
 6: else if $v$ = Linked then
 7:     triggerTransitive($v$)
 8: end if
```

case of line 1, only the *VerificationPolicy* attached to the argument *VerificationPolicyControlStrategy* is triggered. In case of a bidirectional *VerificationPolicyControlStrategy* as in line 3, then both the *VerificationPolicy* attached to it and the one attached to the "opposite" *VerificationPolicyControlStrategy* are triggered. Function *triggerTransitive*, in the block of line 6, is responsible for identifying and triggering all the policies inside the path of the linked strategy. In algorithm 4, function $trigger(VerificationPolicyControlStrategy)$

Algorithm 4 Transitive Trigger

```
Input: VerificationPolicyControlStrategy $v_{control}$( of type “Linked”)
Output: All appropriate policies are triggered
 1: $v_{previous} \leftarrow v_{control}$
 2: {Add all previous strategies}
 3: while $(v_{current} = v_{previous}.getPrevious) \neq null$ do
 4:     $trigger(v_{current})$
 5:     $v_{previous} \leftarrow v_{current}$
 6: end while
 7: $trigger(v_{control})$
 8: $v_{previous} \leftarrow v_{control}$
 9: {Add all next strategies}
10: while $(v_{current} = v_{previous}.getParent) \neq null$ do
11:     $trigger(v_{current})$
12:     $v_{previous} \leftarrow v_{control}$
13: end while
```

is responsible to add the argument policy to the work-list in the blackboard component. The algorithm has two parts, one traverses all policies backwards until a null “previous” is found and triggers all the policies found. The second part traverses all the policies forwards, triggering all “next” policies. It is worth noting that there should not be duplicates of the policies in the blackboard’s work-list. This can be guaranteed by using appropriate data structures that do not allow duplicates.

Policy Triggering Examples

The functionality of the two aforementioned algorithms will be demonstrated by using three examples all based on the example multi-graph and dependency tree of section 4.4.2. The multi-graph of that examples is illustrated in figure 4.10 and the derived tree in figure 4.11.

Example 1. For the first example the case of an update of the node 9 will be discussed. In the event that node 9 changes, the *Policy Manager* component will trigger the correct policies using algorithm 2. Node 9 does not depend on other components (no outgoing edges from it in figure 4.11), but Node 6 depends on it, as modelled by the e_{15} edge. Algorithm 2 will take into account the incoming edge to node 9 and it will trigger the related *VerificationPolicy* P6-9, that is attached to the *VerificationPolicyControlStrategy* object S6-9. Figure 5.1 show the *changed* node in *red* and the *triggered* policies/strategies in *green*. Because relation that relates nodes 6 and 9 is r_6 and antisymmetric, the *VerificationPolicyControlStrategy* is *Unidirectional* and the triggered *Policy* is one.

Example 2. For the second example the case of an update of node 10 will be presented. Node 10 relates to node 6, as seen by edge e_{13}, via the *Symmetric Relation* r_2. As a result the *VerificationPolicyControlStrategy* objects attached to nodes 6 and 10 are *Bidirectional* and connected with each other. The update of node 10 will trigger both the *Policy* objects attach to the two aforementioned *VerificationPolicyControlStrategy* objects, as seen in figure 5.2. The framework will understand that *VerificationPolicyControlStrategy* S10-6 is *Bidirectional* and will traverse its “opposite” S6-10. Finally, there will be two *Policy* objects triggered, P10-6 for the dependency of 10 on 6 and P6-10 for the dependency of 6 on 10.

Example 3. The third example for this section will demonstrate the case when a node who relates on other nodes via a *Transitive Relation* is updated. As figure 5.3 depicts, if node 5 changes due to a maintenance operation, then the corresponding *Policy* attached to the *VerificationPolicyControlStrategy* object S5-6 must be triggered. This *Policy* verifies the relation that node 5 has with node 6 based on edge e_{12} and *Relation* r_4. Because r_4 is *Transitive*, the corresponding *VerificationPolicyControlStrategy* is of type *Linked.* The framework will traverse all the previous and next *VerificationPolicyControlStrategy* objects in the doubly linked list and will trigger all the attached policies. This will guarantee that there is no transitive dependencies that are not checked upon the update of node’s 5 component.

Example 4. The last example for this section will demonstrate how *pathSelection* function works. This example assumes a software system with four nodes, n_1, n_2, n_3 and n_4 that relate with each other with relations $r_1(n_1, n_2), r_2(n_1, n_3)$ and $r_3(n_1, n_4)$, where at each point in time only one of the relations of n_1 to nodes n_3 and n_4 can be active. This is why the example assumes that relation r_2 is an invocation relation and the code of the component that node n_1 models, looks like in figure 5.4. Obviously, this system will be modelled by a dependency tree where there will be a *Disjunctive* helper node with n_3 and n_4 as its children and this helper node, along with node n_2 will be children of the *Conjunctive* node n_1. This example will demonstrate two cases.

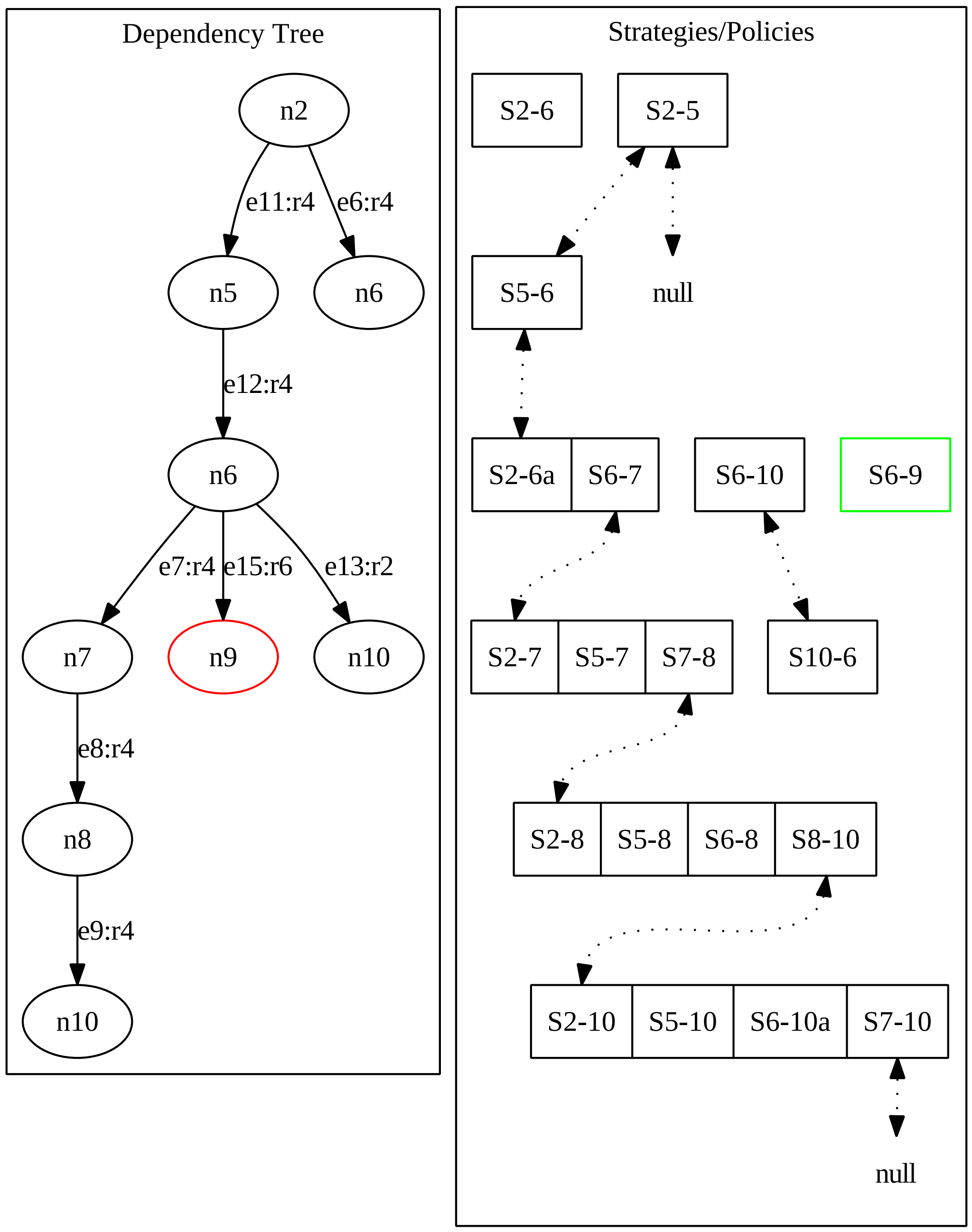


Figure 5.1: Triggering an antisymmetric relation

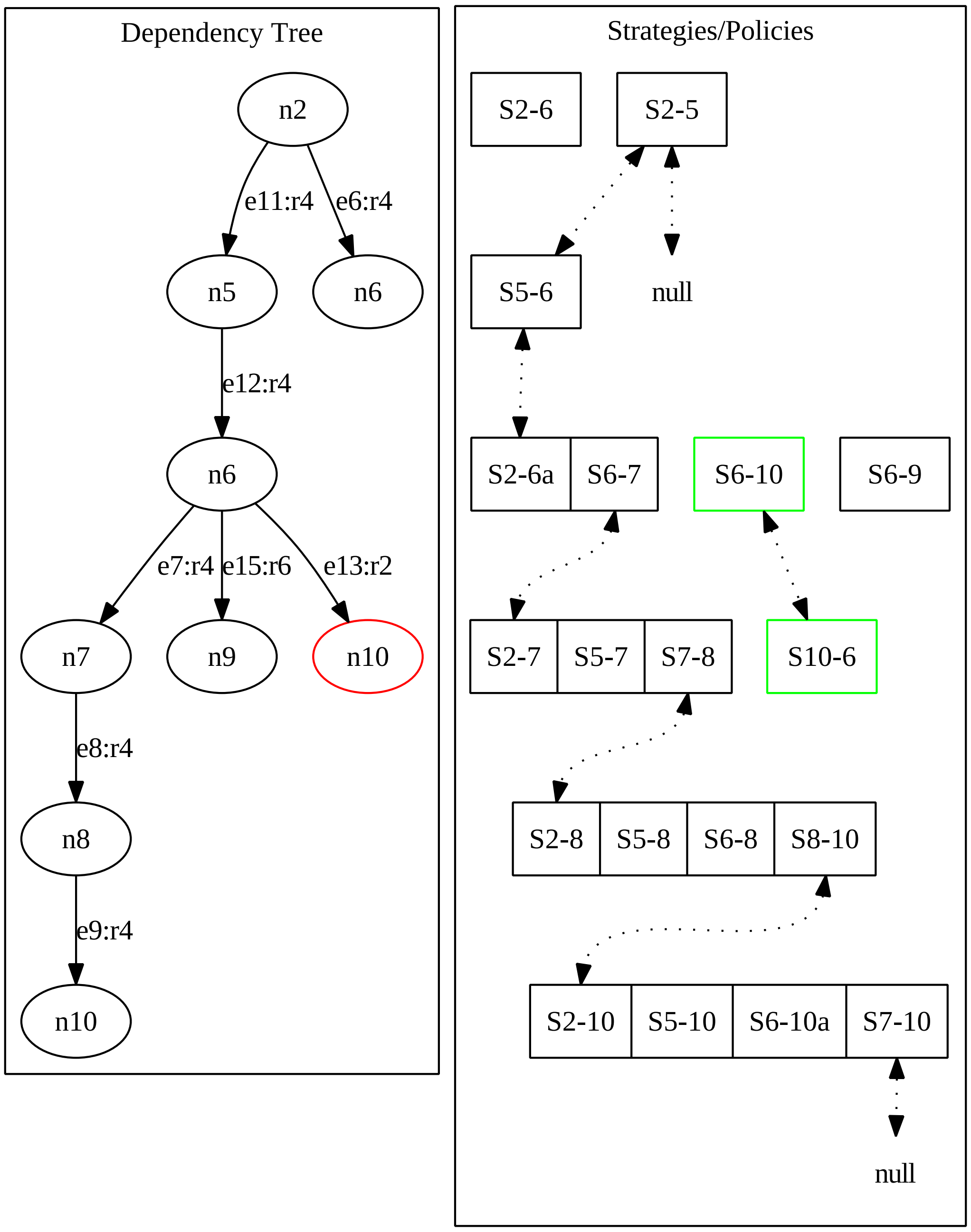


Figure 5.2: Triggering a symmetric relation

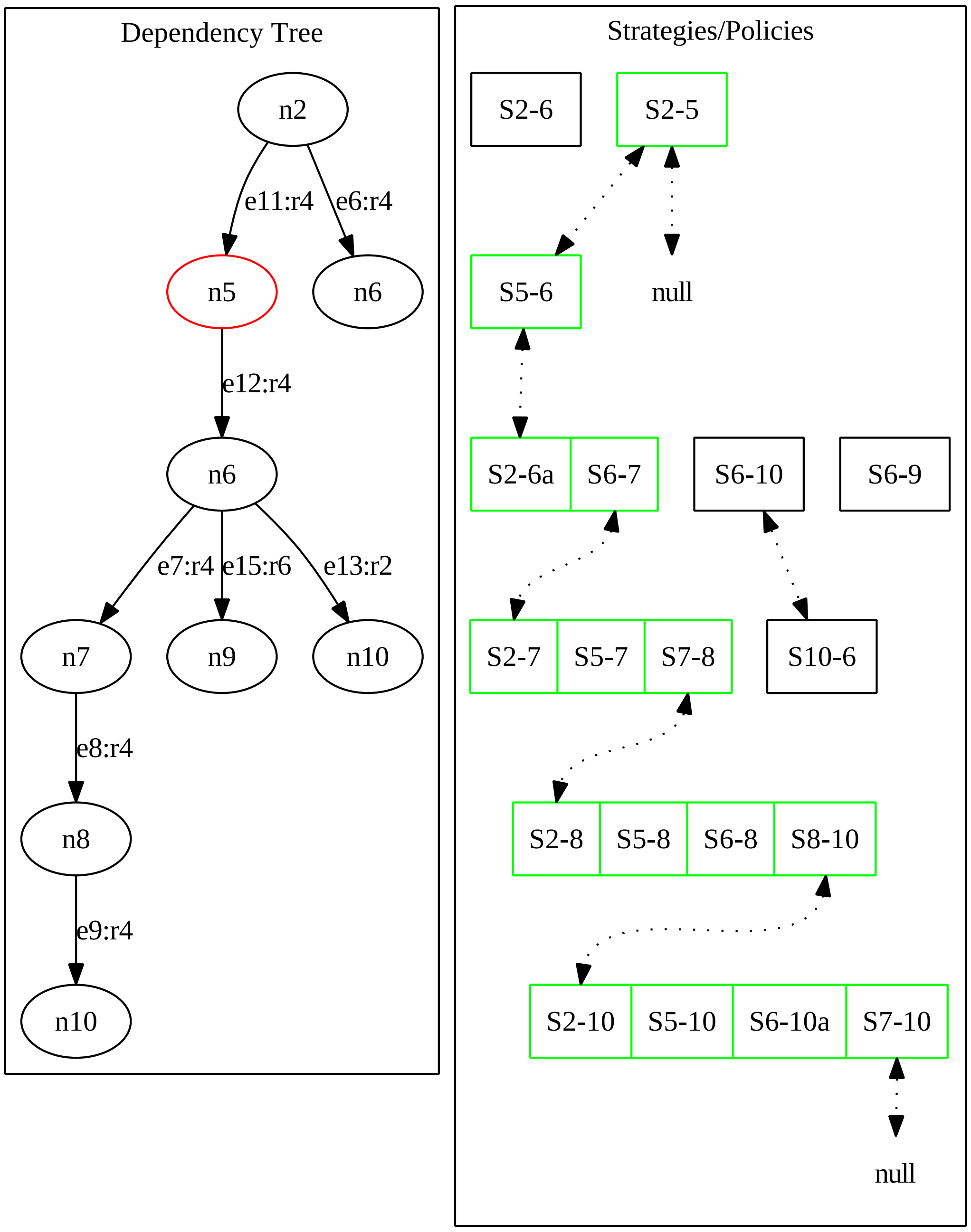
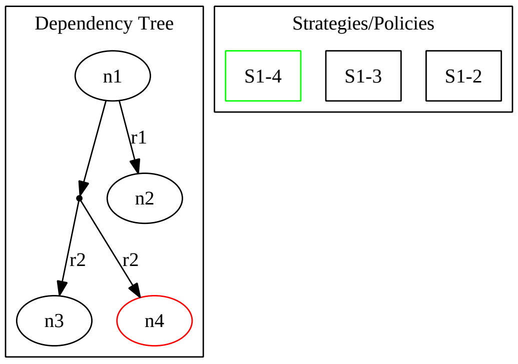


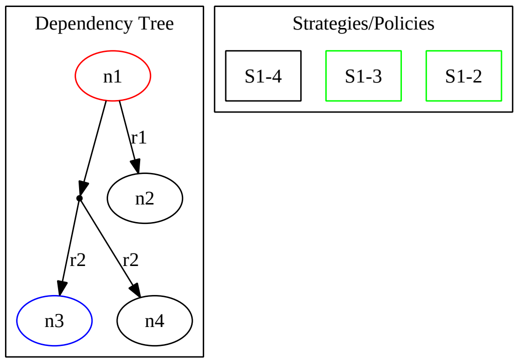
Figure 5.3: Triggering a transitive relation

```
function n1(boolean b){
        if(b){
                n3()
        }else{
                n4()
        }
}
```

Figure 5.4: Code for exclusive calls



(a) Example 4a - No path selection needed



(b) Example 4b - Path selection needed

Figure 5.5: Example 4 - Triggering dependencies near *Disjunction* nodes

Example 4a. In this example, the component that node n_4 models, is changed. In this case, there is no need to select a path, as it is straightforward that the *VerificationPolicy* that verifies the relation $r_2(n_1, n_4)$ should be triggered, because this is the relation risky to cause problem in the system. This is illustrated in figure 5.5a.

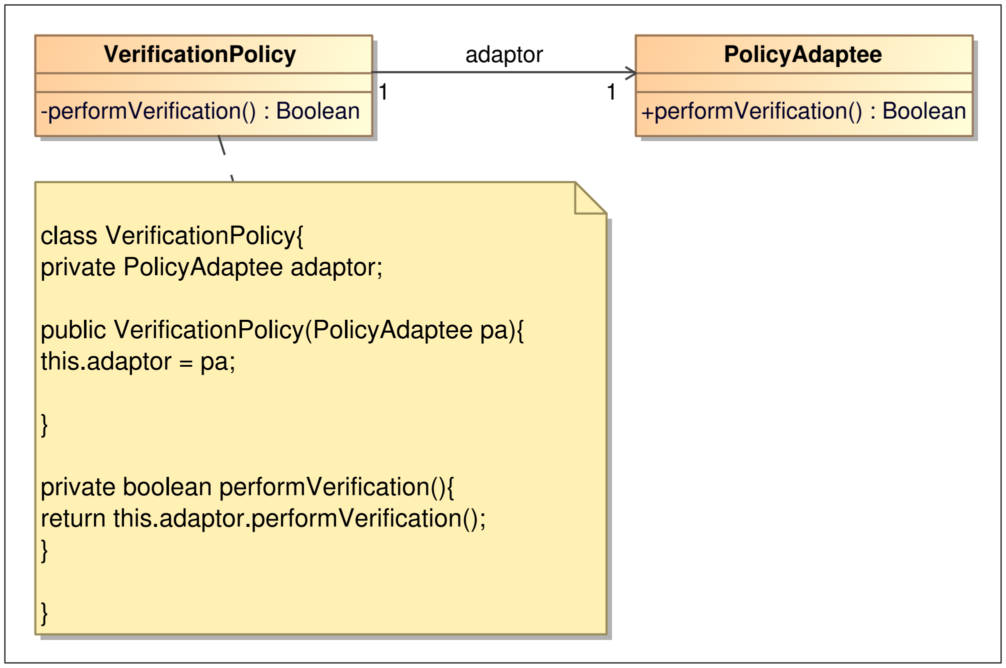
Example 4b. In this example, the component that node n_1 models, is changed. In this case, there has to be a path selection. This is why node n_1, depending on the runtime state of the system, depend on either node n_3 or n_4. Assuming that $b = \textbf{true}$, then the *VerificationPolicy* that refers to the dependency of node n_1 on node n_3 should be selected. This is illustrated in figure 5.5b. Of course, the *VerificationPolicy* that corresponds to the relation of node n_1 on node n_2 should be triggered.

5.1.2 Verifying Triggered Policies

The previous section demonstrated how the framework identifies the correct policies to be triggered, based on the verification strategies in the dependency tree. This section will show how these policies are verified by the *Policy Verifier* sub-component of the *Policy Manager*, shown in figure 3.1.

During the tree generation process, each node is attached with a number of *VerificationPolicyControlStrategy* and *VerificationPolicy* objects, based on the types of the *Relations* it has with other nodes. Each *VerificationPolicy* object is responsible for verifying the dependency of the node to other nodes, i.e. the *VerificationPolicy* objects attached to each node verify it's "outgoing" relations. *Modeller* component adds the appropriate *VerificationPolicy* object during the tree generation phase. Based on the type of the *Relation* on which it is attached, the *VerificationPolicy* object must execute different methodology on verifying it. As a result, the *VerificationPolicy* object is designed using the *Adapter Design Pattern*, [34], as illustrated in 5.6. Each *VerificationPolicy* object contains a reference to objects of type *PolicyAdaptee*, which is the object that actually implements the algorithm/methodology to perform the verification. Upon its initialization using the default constructor, *VerificationPolicy's* "adaptor" reference is bound to a specific *PolicyAdaptee* instance which can verify the type of the *Relation*. When the policy is triggered, the method *performVerification* is invoked to deny or verify the dependency. As depicted in figure 5.6, this invocation actually invokes the *performVerification* of the *PolicyAdaptee* object, therefore the correct verification code is executed. This allows for flexibility in the framework's implementation, as the *VerificationPolicy* object is not obliged to implement the *performVerification* method and the actual implementation of the *performVerification* method can be changed at runtime, simple by changing the "adaptor" reference in the appropriate *VerificationPolicy* object.

The previous paragraph demonstrated the mechanics behind the verification of an individual *VerificationPolicy*. The purpose of the present chapter is to present how *Policy Manager* component provides the verdict for a node update. Section 5.1.1 already demonstrated which *VerificationPolicy* objects are triggered upon the update of a node. During each update, *Policy Manager* keeps track which *VerificationPolicy* objects have been triggered by assigning a *key* to all *VerificationPolicy* objects that relate to a specific node update and instantiating a new *Policy Verifier* sub-component for each node update. All the *VerificationPolicy* objects invoke their *performVerification* method and the boolean result is populated to *Policy Verifier* sub-component. The later sub-component awaits for all the triggered policies to populate their result. Afterwards, it provides its final verdict based on the results it collects from the individual *VerificationPolicy* objects. If one or


VerificationPolicy
-performVerification() : Boolean
adaptor
1
1
PolicyAdaptee
+performVerification() : Boolean

```
class VerificationPolicy{
private PolicyAdaptee adaptor;

public VerificationPolicy(PolicyAdaptee pa){
this.adaptor = pa;

}

private boolean performVerification(){
return this.adaptor.performVerification();
}

}
```

Figure 5.6: VerificationPolicy Adapter pattern

more *VerificationPolicy* return *false* then the final verdict is that the node change is *denied*, otherwise the change is *verified*.

Although the actual implementation is event-driven, algorithm 5 shows all the logic behind the verification process of a node's update. The algorithm's input is the list of triggered policies, as constructed by the *Policy Trigger* sub-component. The result is a *boolean* value regarding the verification or denial of the node's update.

Algorithm 5 Verification

```
Input: List of triggered VerificationPolicy objects, vplist
Output: Boolean verdict
1: for all VerificationPolicy $vp \in vplist$ do
2:    if $vp.performVerification()$ = false then
3:       return false
4:    end if
5: end for
6: return true
```

5.2 Example

This section will present a complete example of policy triggering and policy verification. This example assumes the same dependency tree as the examples of section 5.1.1 and that nodes n_5 and n_9 are changed. Based on the previous discussion, the triggered policies can be seen in figure 5.7. The policies triggered as a result of n_5 node's update are painted in *green* and those that are affected by the change in node n_9, in *blue*. In this example, *Policy Manager* component would have instantiated two *Policy Verifier* objects, one for the "green" *VerificationPolicy* objects and one for the "blue" *VerificationPolicy* objects. *Policy Verifier* named **PV1** is assigned for the "green" ones and *Policy Verifier* named **PV2** is assigned for the "blue" one. If the results of each individual *VerificationPolicy* is like the ones stated in table 5.1, then **PV1** will output **false** and deny the change of node's n_5 component and **PV2** will provide verdict **true**, verifying the change of node's n_9 component.

Table 5.1: Results from *VerificationPolicy* objects of section's 5.2 example

Policy Verifier	**VerificationPolicy**	**Result**
PV1	S2-5	*true*
	S5-6	*true*
	S2-6a	*true*
	S6-7	*false*
	S2-7	*true*
	S5-7	*true*
	S7-8	*true*
	S2-8	*true*
	S5-8	*true*
	S6-8	*true*
	S8-10	*true*
	S2-10	*true*
	S5-10	*true*
	S6-10a	*true*
	S7-10	*true*
PV2	S6-9	*true*

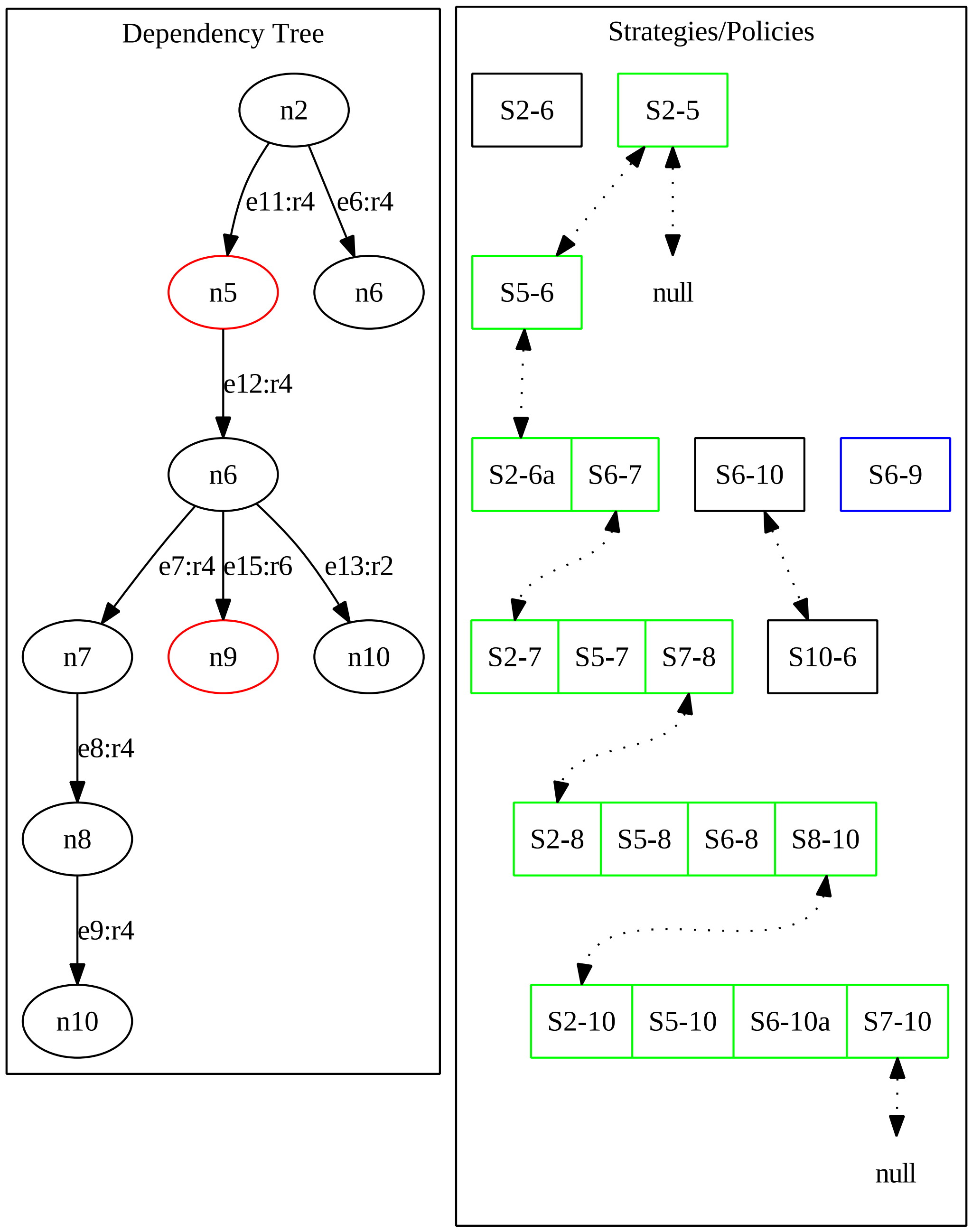


Figure 5.7: Dependency Tree and triggered policies for section 5.2

Chapter 6

Case Studies

This chapter discusses the usage of the proposed framework and methodologies as presented in chapters 3, 4 and 5, by applying them to a case-study pertaining to a SOA application and by providing a performance evaluation.

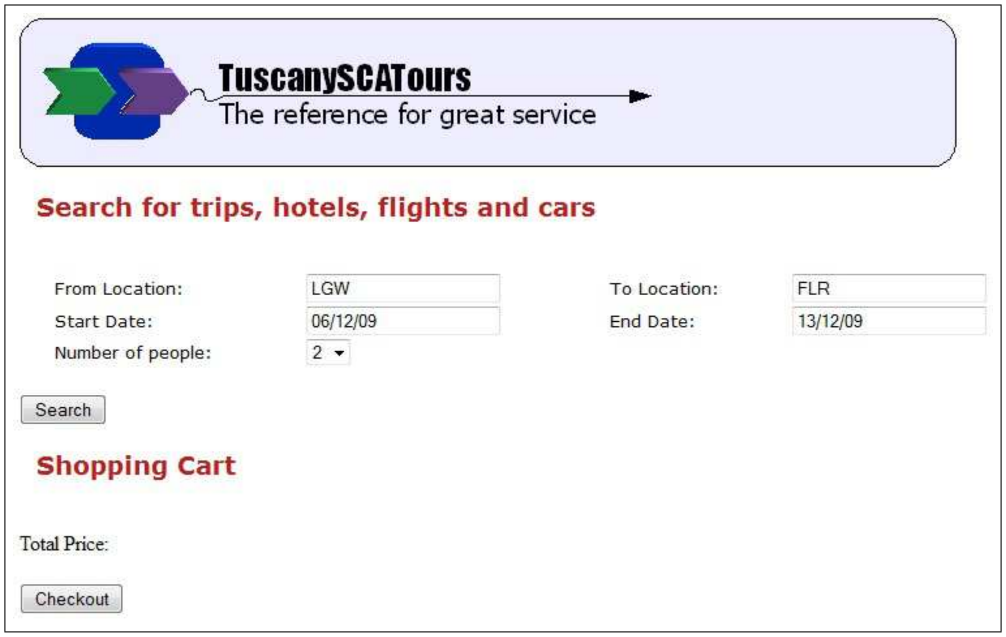
The case study involves a web application designed and implemented using the Service Component Architecture (SCA) paradigm. This case study will demonstrate how the components of the framework presented in chapter 3 can be specialized for a specific type of software systems and how the algorithms presented in chapters 4 and 5 can be leveraged for dependency analysis in SCA systems.

This chapter will close by providing experimental results on the performance of the algorithms mentioned in chapters 4 and 5. The results demonstrate that the proposed methodology of this thesis can be applied in the maintenance phase of real-world applications.

The first section of this chapter discusses the internal structure and the characteristics of the system under maintenance, as well as the extensions made to the framework to incorporate the SCA system. Next section provides the case studies related to the SCA system under maintenance and the chapter concludes with the performance measurements for the policy verification algorithms.

6.1 System Under Maintenance

The sample SCA system that was chosen as being the system under maintenance is a Java Travel Sample application, originally presented in [62], which is a part of the Apache



Search for trips, hotels, flights and cars

From Location:	LGW	To Location:	FLR
Start Date:	06/12/09	End Date:	13/12/09
Number of people:	2		

Figure 6.1: SCA Tours Travel Sample home page

Tuscany, SCA Java project, [63]. This travel sample application is a web based system which allows users to search for flights, hotel bookings and car rentals and plan a trip, either pre-packaged or customized by the user choosing the three aforementioned components individually. It features an 'html' page where the customer can insert departure and destination location for the flight. The system searches for available flights with the specified criteria and proposes packaged trips or customized options to the user. The home web-page of the sample application is illustrated in figure 6.1, while the SCA design of the corresponding system is depicted in figure 6.2.

The SCA application consists of 8 *composites* and 15 *components* in total. These composites and components are as follows:

1. **fullapp-ui:** This composite contains the front-end component of the system. It is responsible for getting input from the user, initiating the search for the appropriate trip(s), notifying the user of the search results and managing purchase of trips from the user. It contains two components, *TuscanySCAToursUI* and *SCATours*.

   (a) *TuscanySCAToursUI* is the user's entry-point to the system, the web-page shown in figure 6.2. It is implemented as a "widget", i.e. via html and javascript

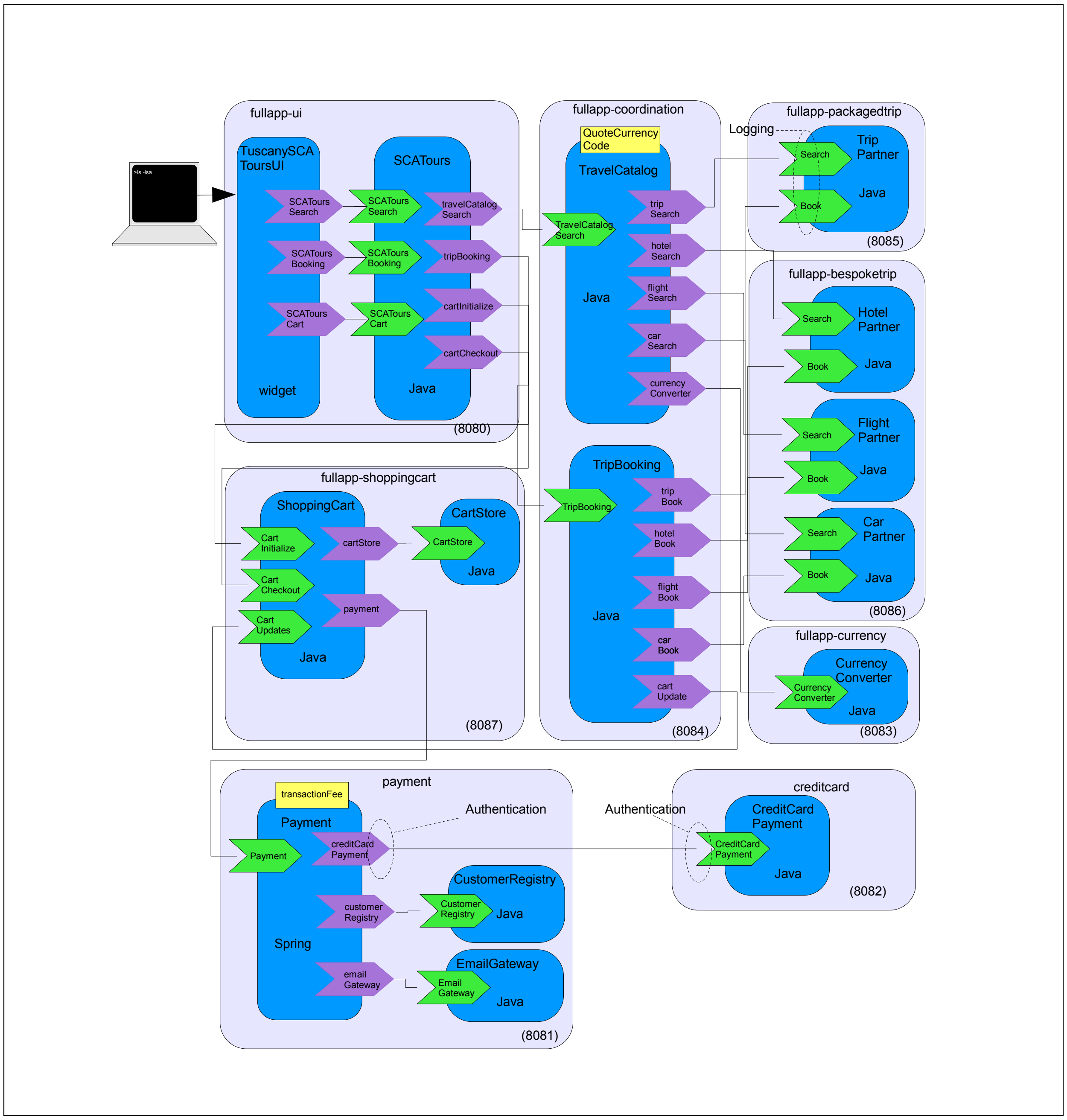


Figure 6.2: Travel Sample SCA design, [62]

code. This component has three references to the second component of the **fullapp-ui** composite, *SCATours*. Being the front-end, *TuscanySCAToursUI* component forwards user's requests for travel search, travel booking and cart management to the *SCATours* component which is responsible to call other component to perform the three operations.

   (b) *SCATours* is the component with the Java code that implements the logic behind the user's requests and data received from the html forms of widget component *TuscanySCAToursUI*. It offers the three aforementioned services, by forwarding the invocations to the corresponding composite each time. This component serves as an intermediate between the presentation of the data to the user (*TuscanySCAToursUI*) and the components and composites that implement the system's business logic.

2. **fullapp-coordination:** This composite contains the appropriate components that coordinate the process of booking a custom-made or pre-packaged trip for the user, from search to actual booking. It consists of two components, *TravelCatalog* and *TripBooking*.

   (a) *TravelCatalog* component handles search requests from the user. It provides one service, *TravelCatalogSearch* and references five other services. It references the search services from *TripPartner* component (for pre-packaged trips) and from *HotelPartner*, *CarPartner* and *FlightPartner* components (**Partner* components), for custom-made trips. It also references the *CurrencyConverter* service for currency conversion during the display of the trip's cost.

   (b) *TripBooking* component handles booking requests from the user. It offers one service, *TripBooking*, and references the *Book* services from the four aforementioned **Partner* components, for booking a pre-packaged trip or a user-compiled trip. It also references the *CartUpdates* service from *ShoppingCart* component, to update the cart of the user with new trip bookings.

3. **fullapp-packagedtrip:** The composite which is responsible to search and book pre-packaged trips for the user. It consists of only one component, the *TripPartner* component.

   (a) *TripPartner*. The component offers two services, *search* and *book*, for users to search for or book pre-packaged trips, respectively.

4. **fullapp-bespoketrip:** The composite which is responsible to search and book individual parts (flight,hotel,car) of a user-made trip. It consists of three components:

FlightPartner, *HotelPartner* and *CarPartner*. If users do not want a pre-packaged trip, they use the services offered by this composite's components to create their trip.

   (a) *FlightPartner*. This component offers *search* and *book* services for flights.
   (b) *HotelPartner*. This component offers *search* and *book* services for hotel bookings.
   (c) *CarPartner*. This component offers *search* and *book* services for car rentals.

5. **fullapp-currency:** This composite is responsible for converting the currency of the trip packages posted by the two aforementioned composites to the currency requested by the user. It consists of only one component, the *CurrencyConverter* component.

   (a) *CurrencyConverter*. This component offers a service with the exact same name. The service takes an amount of money and converts it from its currency to a desired currency. The desired currency is configured through the *QuoteCurrencyCode* property of the *TravelCatalog* component.

6. **fullapp-shoppingcart:** This composite handles and maintains the shopping cart of the customer. It consists of the *ShoppingCart* and the *CartStore* components.

   (a) *ShoppingCart* is the main component to handle updates, creation or removal of shopping carts. It offers three services, *CartInitialize*, *CartCheckout* and *CartUpdate*. The first service is invoked when the user initiates a search for trips. The second is invoked when the user chooses to purchase the pre-packaged or custom-made trip and the last one is invoked when there is an update to the cart (user chose a new trip, removed a trip from the cart etc.). Furthermore, the component references two services, *CartStore* to store the shopping cart of a user and *Payment* to process the payment for a shopping cart the user wishes to purchase.
   (b) *CartStore* is a component that stores the shopping cart during the customer's browsing through the website. The life-cycle of a shopping cart that is stored in the *CartStore* component starts when the user initiates a search for a trip. The *CartStore* component follows the updates to the cart as the customer chooses trips they want to add to the cart or remove from the cart. Finally, the life-cycle of a shopping cart ends when the user decides to purchase the trips that exist inside a cart.

7. **payment:** This composite is responsible for handling payment of the booked trips from the user. It consists of three components, *Payment*, *CustomerRegistry* and *EmailGateway*.

(a) *Payment.* This component handles payment processing from the customer. It offers the *payment* service and references the *creditCardPayment* service to process credit card payments, the *CustomerRegistry* service to register the user as a customer and the *EmailGateway* service to e-mail the invoice to the customer.

(b) *CustomerRegistry.* This component is responsible to store customer information. It stores names, e-mail addresses and credit card information for customers of the SCATours system.

(c) *EmailGateway.* This component provides one service, the *EmailGateway* service which sends e-mails to the customers about the status of their payment(s).

8. **creditcard:** This composite is responsible for "simulating" the processing of credit card payments from the customer. It consists of one component, the *CreditCardPayment* component.

   (a) *CreditCardPayment.* This component offers the *CreditCardPayment* service which authorizes or not a credit card payment, based on the information of the credit card (type, number etc.). It is worth to note that the wiring of this service with the reference from the *Payment* component must comply to the authentication policy, so every communication of sensitive data (credit card) is done only with legitimate parties.

The aforementioned discussion describes a system of approximately 3.2 KLOC found in 80 Java classes contained in 40 Java packages (plus a few more written in other programming languages, such as html or javascript). It functionality can be summarized with the next steps:

(a) The user visits the web-site where the SCATours application resides. A cart is initialized for the user.

(b) The user searches for departure and destination locations for the trip. The search is forwarded to the *TravelCatalog* component which in turn forwards the request to the *Partner components. Via a callback interface, *TravelCatalog* informs the two components of the **fullapp-ui** composite of the results and the web-page is refreshed.

(c) The user has the opportunity to select either a pre-packaged trip or make a custom-made trip. When user hits the "book" button, *TripBooking* updates the cart with the trips the customer wishes to purchase. The user is presented with the final price and requested to confirm the purchase (checkout).

(d) The user can hit the check-out and when it does so the payment is processed, an e-mail is send to the customer about the purchase, the *TripBooking* component books the trip's elements (by calling the appropriate "Book" services from the **Partner* components) and *CartStore* component removes the cart. The web-page is refreshed and ready to get a new order from the customer.

The system that was briefly described above will be used as the system under maintenance for the proposed framework and the proposed approach of this thesis.

6.2 Case Study - Web Services Dependency Analysis on SCA Models

This case study views services offered by SCA systems as "first class citizens". Nodes of the dependency multi-graph and dependency tree will model services offered by the SCATours web application, as seen in figure 6.2. The following dependencies between services were chosen to modelled.

1. *Synchronous Invocation*, r_1. This dependency denotes the case where a service has a reference to another service and invokes the other service synchronously via a specific interface. It is a transitive dependency. The *VerificationPolicy* objects that verify this kind of dependencies checks the interface that the called method implements, versus the actual call statement in the callee service (number and type of parameters, type of result).

2. *Asynchronous Invocation*, r_2. This dependency denotes the case where a service calls asynchronously another service and implements a specific callback interface in order to receive the result in the future. It is a transitive and symmetric dependency (two strategies will be created for this type of dependencies). The *VerificationPolicy* objects that verify this kind of dependencies will check the interface and the callback interface as described for the r_1 type of dependencies.

3. *Binding*, r_3. This dependency denotes the binding of two services with each other. It means if the protocol, address and port of two wired services match. For example, if service s_1 uses service s_2, then service s_1 should use the protocol service b implements for receiving messages and know the address and port where s_2 can be reached at. It is an asymmetric dependency.

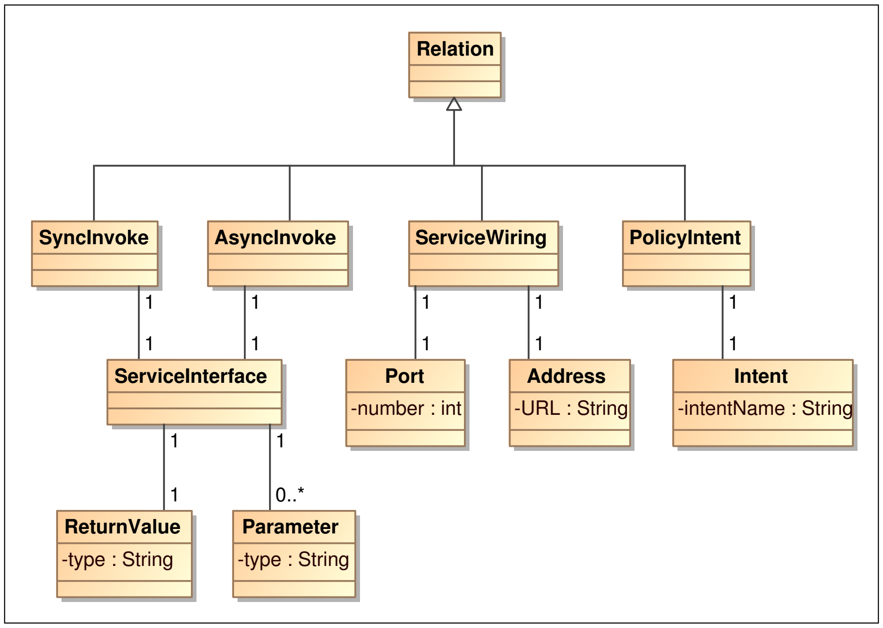


Figure 6.3: Relations for Case Study

4. *Policy Requirement*, r_4. This dependency models the case where a service requires a specific SCA policy intent to be applied to the communication channel. This is the case of the *Payment* service that requires authentication of messages. This dependency is antisymmetric.

Those dependencies all extend the general *Relation* class defined in chapter 4. Figure 6.3 illustrates how this extension happens. *SyncInvoke* relation refers to the transitive relation r_1 and *AsyncInvoke* refers to relation r_2. Those two relations associate with a *ServiceInterface* object. The semantics of this object are straightforward, as it models the return type and number of methods and parameters used for the invocation of a web service from another. It is the information that will be passed to the *VerificationPolicy* objects after the generation of the tree, in order to check the invocation dependencies. *ServiceWiring* refers to r_3 type of relations. As it checks for wiring of the services, it maintains the information of *URL Address* and *port* where a service can be reached. This information is used also for checking the service wiring by *VerificationPolicy* objects. Finally, *PolicyIntent* refers to r_4 type of relations and the associated *Intent* object keeps the name of the required intent (also to be checked by the *VerificationPolicy* objects). For these dependencies to be identified, the *Modeller* component of the framework must read the appropriate xml files that configure the composites of the SCA application and extract

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           name="fullapp-coordination">

    <component name="TravelCatalog">
        <implementation.java class="com.tuscanyscatours.travelcatalog
         .impl.TravelCatalogImpl"/>
        <service name="TravelCatalogSearch"/>
        <reference name="hotelSearch">
            <binding.ws uri="http://localhost:8086/Hotel/Search"/>
            <callback>
                <binding.ws name="callback"
                            uri="http://localhost:8084/Hotel/SearchCallback"/>
            </callback>
        </reference>
            .
            .
            .
</composite>
```

Figure 6.4: Excerpt from the composite xml file of fullapp-coordination

the appropriate information from them. During the generation of the tree, the *Modeller* component must read implementation information to attach the appropriate policies to each node of the dependency tree. For example, in figure 6.4 an excerpt from the SCDL file of fullapp-coordination composite is shown. It features the *TravelCatalog* component and the *TravelCatalogSearch* service it offers. Furthermore it has a reference to the *hotelSearch* service with a web service binding to it and a callback interface, showing that the invocation is asynchronous. The corresponding multi-graph will contain two edges from node “TravelCatalogSearch” to node “HotelPartner/Search”, one edge modelling the r_2 dependency between these elements and one modelling the r_3. The multi-graph of the system with respect to the four aforementioned relations between web services can be seen in figures 6.5, 6.6 and 6.7. The separation of the multi-graph was done for illustration purposes. In figure 6.5 we can see that all search requests are asynchronous and the called services use a callback interface to communicate the results back to the callee. Another significant example is service *Payment* in figure 6.7 which requires *authentication* policy to be applied and that is the reason why *CartCheckout* service and *Payment* service also depend via a r_4 relation. It is worth noting that although two nodes are connected only with one line in these figures, they are actually multiple/parallel edges. To clarify this, let us consider the *Payment* and *CreditCardPayment* services in figure 6.7. Although it appears that two edges connect these two edges, the fact is that there are three edges that connect those two services, each one for one relation from r_1, r_3 and r_4. In the figure they appear as one edge to avoid overwhelming the figure from redundant illustrative information.

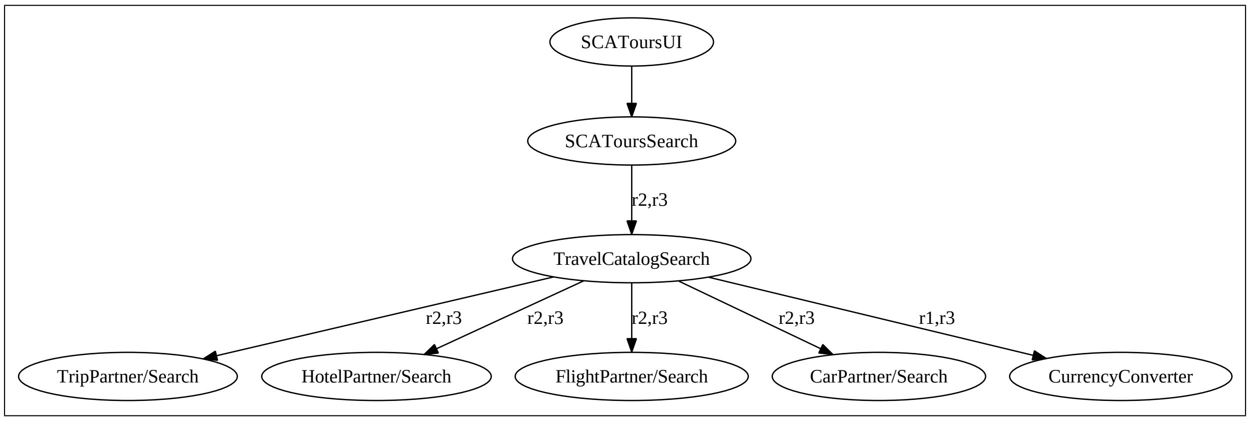


Figure 6.5: Multi-Graph for Case Study (1 of 3)

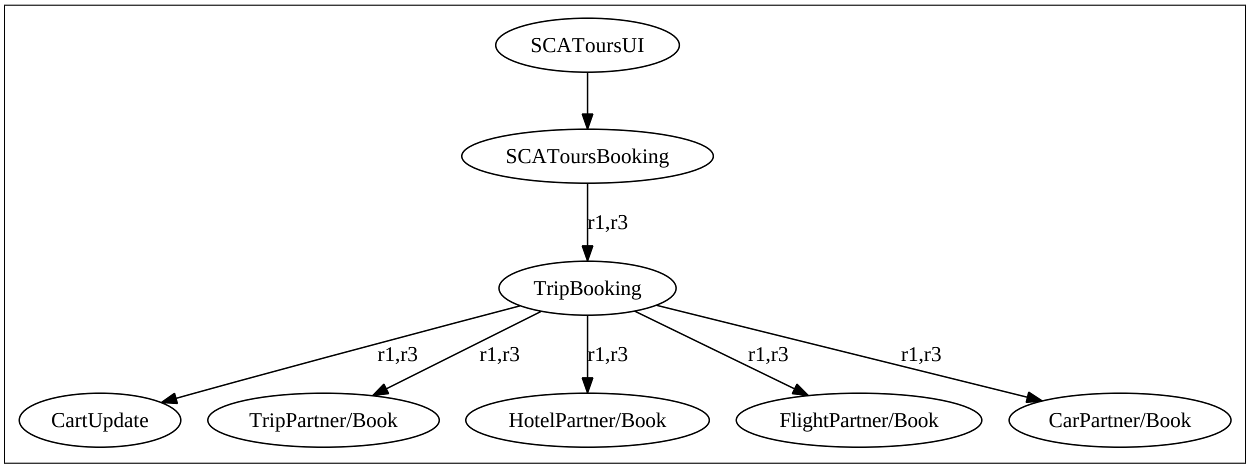


Figure 6.6: Multi-Graph for Case Study (2 of 3)

There are three interesting dependency trees that can be generated from the aforementioned multi-graph. The first tree relates to relations r_1 and r_2 and can be used to guide checks of invocation/interface compatibility between the web services. The second tree

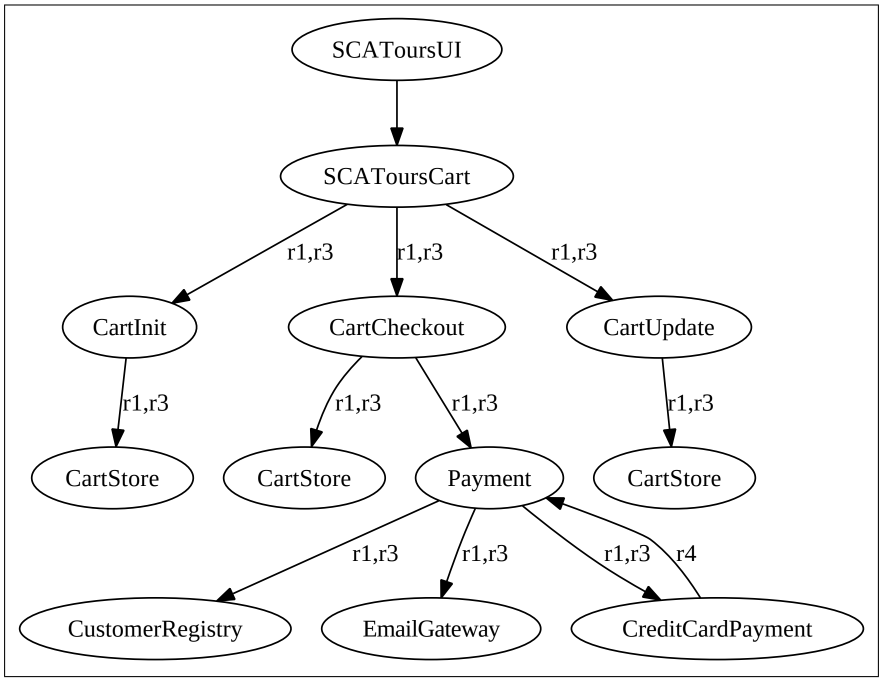


Figure 6.7: Multi-Graph for Case Study (3 of 3)

relates to relation r_3 and can be used to check the wiring and binding of the web services with each other. Finally, the third tree takes into account relation r_4 and involves only the nodes corresponding to services *Payment* and *CreditCardPayment*.

6.2.1 Invocation/Interface Dependency Process and Analysis

Process

In order for the framework to facilitate in dependency checking for web services the following process is followed:

1. *Modeller* component reads SCDL xml files and identifies dependencies between SCA services offered by different components, allowing it to create the dependency multi-graph.

2. *Modeller* generates the dependency tree based on the set of the aforementioned synchronous and asynchronous invocation relations, r_1 and r_2.

3. *Tree Annotator* annotates disjunction nodes, as explained before.

4. Upon the change of an SCA component, the framework has to identify the list of services that belong to it, in order to pinpoint the nodes that are changed from the dependency tree and execute the algorithms mentioned in chapter 5. This process can be seen in algorithm 6.

Algorithm 6 Identification of the changed SCA service nodes

```
Input: componentChanged: SCA Component
Output: Nodes corresponding to the change, n
1: serviceList = componentChanged.getServices()
2: n = new list of nodes
3: for Service:s in serviceList do
4:     n.add(getCorrespondingNode(s))
5: end for
6: return n
```

Analysis

Taking into account only relations r_1 and r_2 from the aforementioned multi-graph, one can check for invocation dependencies between web services, as the latter evolve through maintenance procedures. For demonstrating this aspect of the case study, let us consider the part of the SCA system that provides the functionality of searching and booking for the trip. The two parts of the derived dependency tree with respect to this functionality are illustrated in figures 6.8 and 6.9. Those two trees can be used by the framework to guide the verification policy in the event that a component (and its offered services) gets updated during the maintenance phase of SCA Travel Sample.

In figure 6.9, there is a *Disjunction* node, because each point in time the system cannot book both a pre-packaged trip and a user tailor-made trip. The invocation of the book service in *TripPartner* component and any of the book service in the other *Partner components is done in different blocks of code and that is why the dependency tree of this figure has this specific layout. Each node in the trees of figures 6.8 and 6.9 is attached with two strategies, one for the transitive property of r_2 and the other for the symmetric part of it. As a result, when the component that offers a service gets updated, the strategies that relate to all the search calls through the system, as well as the callback interface calls, are triggered. The algorithm for identifying which policies to be triggered in a SCA

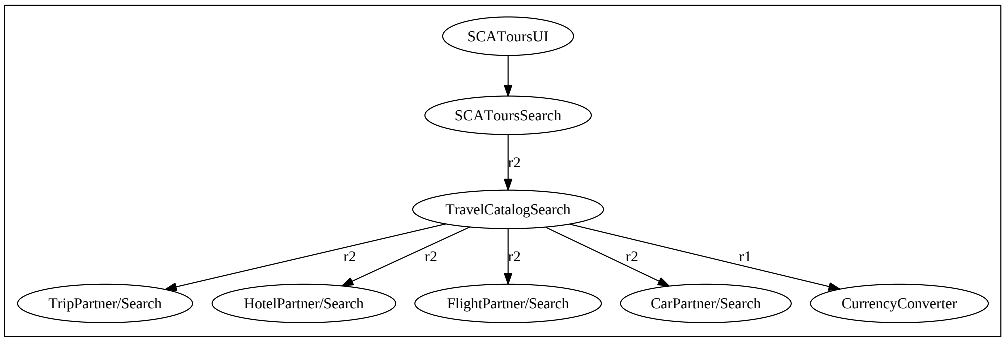


Figure 6.8: Dependency tree for relations r_1 and r_2 of case study - (1 of 2)

component update is algorithm 6. For the component that is changed, the algorithm gets all the services offered by it. Consequently, it identifies the node in the tree that each of these services belong. It returns this list of nodes and the strategies of the nodes in the returned list are triggered using the algorithm presented in chapter 5.

For example, if *TravelCatalog* component gets updated, then the framework matches this update with the update of node *TravelCatalogSearch*, because this service is offered by the aforementioned component. Based on the discussion of chapter 5, in the event of an update of *TravelCatalog* component, the transitive closure of the paths *SCAToursSearch* $\rightarrow$ *TravelCatalog* $\rightarrow$ **Partner/Search* will be taken into account, along with the dependency *TravelCatalog* $\rightarrow$ *CurrencyConverter* and the "opposite" dependencies **Partner/Search* $\rightarrow$ *TravelCatalog.*

6.2.2 SCA Policy dependencies analysis

As discussed earlier in this chapter, *Payment* service has to provide "authentication" of the outgoing messages, because *CreditCardPayment* service requires it to protect sensitive data of credit cards for the users. An excerpt from the composite xml file of the *credticard* composite that shows this requirements can be seen in figure 6.10. This composite file defines the *creditcard* composite with its internal *CreditCardPayment* component, as shown in figure 6.2. The *CreditCardPayment* component offers one service, namely the *CreditCardPayment* and as seen in figure 6.10 it requires the" authentication" intent. This service is referenced by the *Payment* service of the *Payment* component. Therefore, the *Payment* component should provide authenticated outgoing messages. This relation between the two services is modelled via the r_4 relation presented earlier in the present text.

The corresponding tree, that derives from the multi-graph of figures 6.5, 6.6 and 6.7

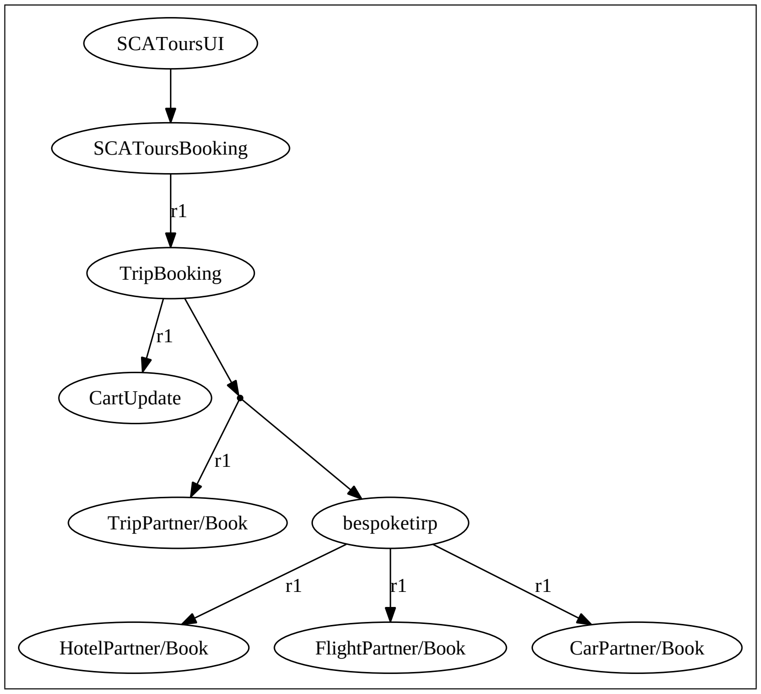


Figure 6.9: Dependency tree for relations r_1 and r_2 of case study - (2 of 2)

with respect to relation r_4 can be seen in figure 6.11. The dependency tree contains only two nodes, the *Payment* and *CreditCardPayment*. The edge that connects them in this case models only the relation r_4. The *VerificationPolicy* objects that will be attached to node *Payment* will have the responsibility to check whether the node *Payment* conforms to the policy intent requirements of *CreditCardPayment*.

6.3 Performance

The previous sections outlined how the framework can be specialized and used for the analysis of numerous dependencies SCA artifacts in a medium-range SOA application can demonstrate. This section presents performance data obtained during experimentation with the code that implements the algorithms presented in chapter 5. The reason behind the decision of choosing those algorithms is their importance for the framework's functionality and the frequency that they are executed compared to the other functions of the framework. It is obvious that the policy triggering and policy verification algorithms presented in chapter 5 are the ones that will be executed more intensively than the others

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="http://tuscanyscatours.com/"
           name="creditcard">

    <component name="CreditCardPayment">
        <implementation.java class="com.tuscanyscatours.payment.creditcard
         .impl.CreditCardPaymentImpl" />
        <service name="CreditCardPayment">
            <interface.wsdl interface=". . ." />
            <binding.ws uri=". . ." requires="authentication"/>
            . . .
        </service>
    </component>

</composite>
```

Figure 6.10: Excerpt from creditcard.composite xml file

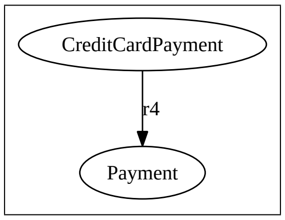


Figure 6.11: Tree derived for Case Study I, with respect to r_4

during the framework's lifetime. This is true as the multi-graph generation algorithms and the dependency tree generation algorithm will be executed once, during the framework's initialization and in rare cases where a change to those two models needs to be done (e.g. when the user of the framework wishes to take more dependency types into account).

6.3.1 Experimentation process and results

Using a set of randomly generated dependency trees we measured the performance of the verification algorithm, in order to calculate the overhead it adds to the classic tree traversal algorithms. A number of trees of various complexities were generated. Two parameters were taken into account for generating the dependency trees: (a) number of nodes and

(b) number of strategies/policies attached per node. Those two factors are the important ones that affect the system's performance during policy triggering and verification. In the simulation that took place, all the strategies/policies attached to all the nodes of the tree were triggered and requested verification. In order to simulate the worst case scenario for the framework's performance, the strategies and policies were manufactured to always return **"true"**, therefore the process of policy verification did not stop in the middle of the tree's strategies/policies to pinpoint a faulty dependency, but rather went through all of them, that is the worst case scenario.

Figure 6.12 presents the graph of the time taken (in milliseconds) to verify all the policies, versus the number of nodes in the tree. The figure contains four graphs, based on the policies attached to each node for every simulation ran. The simulation ran for dependency trees of 10 to 1000 nodes and 5, 10, 50, 100 policies attached per node. The data from the simulation prove that the proposed approach can be used for dependency verification in medium to large scale systems. The obtained data indicate that it takes less than 10 seconds to complete the verification process for trees with up to 1000 nodes and 10 strategies/policies attached per node (approximately 10000 policies). This is a promising result that the framework can work with much more complex SOA systems. The only discouraging result is a "spike" in the case of 100 policies per node in a tree of 1000 nodes. This scenario involves 100000 policies to be triggered and verified which forces the framework to allocate a huge amount of memory and use swap space heavily. Generally, the complexity of the algorithm grows with the number of nodes in the tree and even more with the number of policies attached to the nodes. Therefore, the proposed approach cannot be used in a fine-grained manner (i.e. the trees cannot be abstract syntax trees where each node is a program statement in Java), but it is applicable for interface compatibility and configuration management analysis in modern, real-world web applications.

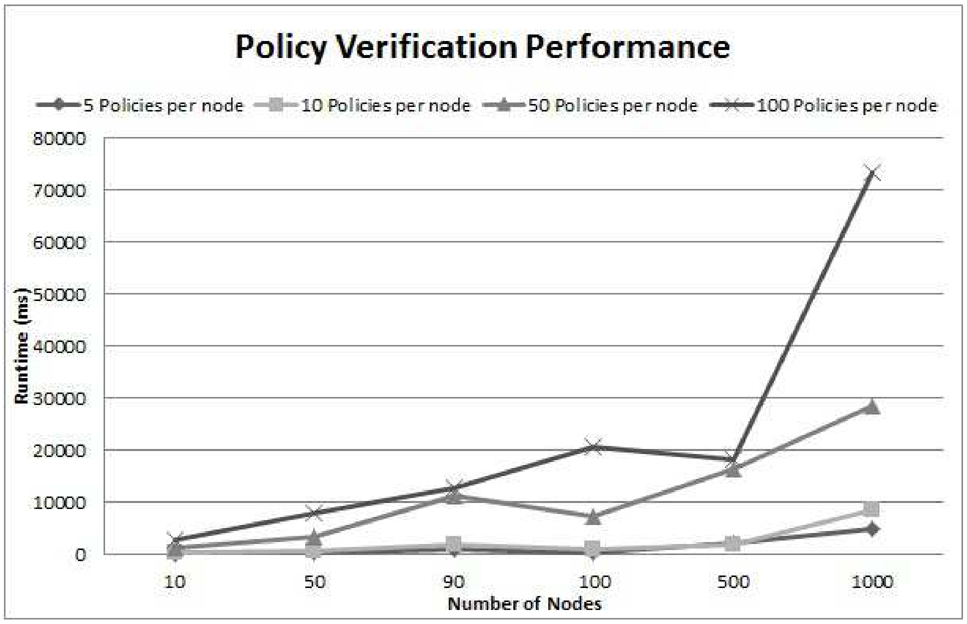


Figure 6.12: Simulation Data

Chapter 7

Epilogue

7.1 Summary

This thesis presented a methodology and a supporting framework to leverage component dependency modelling as well as, system relation properties to guide the verification process during the maintenance phase of a software application. Work found in the technical literature reveals that components in large software systems can demonstrate a vast number and type of dependencies with each other, yet those dependencies are usually modelled uniformly. The proposed approach is to maintain two dependency models for the system, a dependency multi-graph and a dependency tree. The former is used to model all types of dependencies that software components in a system can have with each other. The latter is a subset of the former and takes into account only types of dependencies that are relevant in a specific software maintenance context, e.g. there is no need to check for data dependencies when someone changed only the interface of a component. Based on this concept, the thesis was founded in three pillars.

First, a generic architecture framework to incorporate the modelling and verification algorithms was proposed. The architecture is hybrid, using elements from the Publish/Subscribe and Blackboard architectural styles. All the components of the framework were designed to be as much independent as the framework's functionality dictated. A *Blackboard* component contains the two dependency models and the information on which policies are triggered for verification. Other components can poll *Blackboard* to learn about the state of the verification process and/or retrieve information from the dependency models. A *PublishSubscribe* component is used for components to subscribe to specific types of events and to notify those components when such an event occurs. An *EventHandler* component manages logging and recording of the events occurred in the system under maintenance or in the framework. A *PolicyManager* component manages the verification process and the triggering of specific policies in the context of a specific maintenance operation in the system under maintenance. A *Monitoring* component is proposed last, to facilitate information harvesting from the system under maintenance.

Second, the thesis presented two dependency models, namely the dependency multi-graph model and the dependency tree model. The former is used to represent every possible type of dependency that components of a specific type of software system can demonstrate. For example, in an SCA application, the multi-graph might contain dependencies caused by invocations of services from other services, by wiring the services together or by the requirements in authentication or security in messages exchanged between services. The dependency tree, on the other hand, reflects one view of the multi-graph and contains only a specific set of relations that are relative to one type of dependency analysis to be performed in the system under maintenance. For example, interface compatibility analysis in the aforementioned system relates only to the first two types of dependencies and not on the third one. Furthermore, an algorithm for generating dependency tree(s) from the dependency multi-graph was presented, along with examples to illustrate its functionality.

Third, the thesis presented a methodology to trigger specific policies and verify conformance with the corresponding dependencies, when a software system undergoes maintenance operation. More specifically, in the event of an update or a change in a specific component of the system, the algorithms presented can pinpoint the exact node that changed in the two dependency models. Based on the verification policies attached to the nodes of the tree and the types of the relations involved (transitive, symmetric, antisymmetric, reflective), an algorithm to trigger the appropriate verification policies is proposed. Consequently, the verification process is presented, up to the point where *PolicyManager* component deducts the final "verdict" for the update operation.

Finally, the proposed approach has been evaluated for its performance and tractability. First, the reference architecture was extended to accommodate all the necessary algorithms and artifacts needed to perform dependency checking in the medium scale SCA system, SCA Travel Sample. The former was checked for consistency of dependencies between its web services and deployment configuration artifacts, during maintenance phase. The complete process to adapt the framework and its underlying concepts to the case of SCA systems was presented, along with the concrete example of the SCA Travel Sample. Additionally, a simulation took place to evaluate the performance of the proposed approach in systems with complexity of up to a thousand services and a few hundred components.

7.2 Conclusion

As a conclusion, the proposed approach showed that it is possible to maintain a "global-view" of a wide-spectrum of dependency types among the components of a system, while having the ability to extract only a subset of this "global-view" and perform dependency checks during the maintenance phase. Being able to maintain a "global-view" model of the system is quite important, as it enhances software comprehension and reduces the time that developers or designers have to consume in understanding dependencies in large software systems that consist of thousands of lines of code. However, dependency checking requires detailed information on how the software code works. The proposed work is a step toward this.

Additionally, the proposed approach demonstrated how the knowledge of generic types of relations (symmetric, transitive ...) can assist in preparing governing strategies for the dependency verification process. The proposed approach combines the aforementioned dependency models with a generic methodology to trigger dependency verification policies, based on characteristics of dependencies that are not tied to a specific platform, technology, programming language or type of software system under maintenance.

Furthermore, this thesis demonstrated how the framework can be specialized in order to incorporate all the appropriate information needed for the verification process in the case of SOA applications that are designed and developed under the SCA paradigm. The basic framework was enhanced by an identification process (algorithm 6) to allow for the population of a list of nodes corresponding to the changed SCA components. The proposed framework is independent of such a selection process. This case study showed that the proposed approach can function for specific type of software systems, with some effort from experts in those types of systems.

Finally, the proposed approach can be used from software designers or developers that require a system which provides "global-view" of the dependencies and has the ability to guide the software verification process, or by software maintainers to perform "what-if" analysis when a software maintenance induced change is planned or applied. The sole requirement needed from the stakeholders of this approach is to substantiate the types of relations that can exist in their software systems of interest and provide the appropriate methodologies for building the dependency models and verifying the aforementioned relations.

7.2.1 Limitations

As mentioned in chapter 6 there are some limitations in this proposed approach. The major one is the combined number of nodes and policies the framework can handle. Figure 6.12 show that there is a "spike" in the performance of the triggering and verification of 100 policies per node and 1000 nodes. The implication of this result is that the approach would be impractical to be used in a fine-grained level, for example where the dependency models will try to model program statements as nodes of the multi-graph or the tree. Nevertheless, in a more coarse-grained level, where the two dependency models will encode large software components, the approach works fine and it is safe to state that it performs well.

Another limitation of the system is that it requires the user of the framework to extend the "relation" class of objects in order for it to perform the identification and verification process. Regardless the fact that this approach makes the framework much more expendable and ready to accommodate specific types of software systems, a list of common, well-known and generic relation types could be given in advance, along with the relation properties (symmetric, antisymmetric, transitive ...) they have.

7.3 Future Work

There are different avenues this work can be improved or extended in the future.

First, one can extend the breadth of types of relations being considered, as well as the scope of the analysis of such relations. In modern systems, there are also ways that dependency types can interact with each other. For example the existence of one type of dependency between two components might exclude the existence of another type of dependency between the two same nodes. One type of dependencies might imply the existence of another type of dependencies. Research on this area might extend the current framework and its policy verification properties.

In the governance of the policy verification, a number of reasoning tools can be leveraged for identifying the policies to be triggered and verifying the triggered policies. Such tools might involve SAT solvers, caching of previous results for performance improvement and using program slicing techniques to identify verification work-flows that can be executed in parallel.

Finally, in the area of systems and tooling, effort can be put into integrating the aforementioned framework and its extension for SCA systems to existing SCA runtimes such as Apache Tuscany, OW2 Frascati and fabric3. Work that would extend the framework into software paradigms such as Cloud Computing can be of significant value for providing feedback for the proposed framework and applying the methodology to a wider range of software products.

References

[1] Micha Antkiewicz and Krzysztof Czarnecki. Design space of heterogeneous synchronization. In *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 3–46. Springer-Verlag, 2008. 11

[2] Michal Antkiewicz, Krzysztof Czarnecki, and Matthew Stephan. Engineering of framework-specific modeling languages. *IEEE Trans. Software Eng.*, 35(6):795–824, 2009.

[3] L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, M. Segnan, and D.T. Dupre. Enhancing web services with diagnostic capabilities. In *Web Services, 2005. ECOWS 2005. Third IEEE European Conference on*, page 10 pp., nov. 2005.

[4] Y. Asnar, P. Giorgini, and J. Mylopoulos. Risk modelling and reasoning in goal models. *University of Trento, Tech. Rep. DIT-06-008*, 2006.

[5] R. Aydogan and H. Zirtiloglu. A Graph-Based Web Service Composition Technique Using Ontological Information. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 1154–1155, july 2007.

[6] Siddharth Bajaj, Don Box, Dave Chappell, Phillip Hallam-Baker Maryann Hondo Francisco Curbera, Glen Daniels, Chris Kaler, Dave Langworthy, Anthony Nadalin, Nataraj Nagaratnam, Hemma Prafullchandra, Claus von Riegen, Daniel Roth, Jeffrey Schlimmer, Chris Sharp, John Shewchuk, Asir Vedamuthu, and mit Yalinalp. Web services policy framework 1.2, 2006. 9

[7] S. Basu, F. Casati, and F. Daniel. Toward web service dependency discovery for soa management. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 2, pages 422–429, july 2008.

[8] Philip A Bernstein. Applying model management to classical meta data problems. *In Proc. Conf. on Innovative Database Research,*, pages 209– 220, 2003. 11

[9] E. Bertino, E. Ferrari, and G. Guerrini. An approach to model and query Event-Based temporal data. In *Proceedings of the Fifth International Workshop on Temporal Representation and Reasoning*, page 122. IEEE Computer Society, 1998. 14

[10] Russel C. Bjork. An example of object-oriented design: An atm simulation, 2004.

[11] F. Budinsky, S.A. Brodsky, and E. Merks. *Eclipse modeling framework*. Addison-Wesley Boston;, 2003. 6

[12] A. Chakrabarti, L. De Alfaro, T. Henzinger, M. Jurdziński, and F. Mang. Interface compatibility checking for software modules. In *Computer Aided Verification*, pages 654–663. Springer, 2002.

[13] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*. Kluwer Publishing, Dordrecht, 2000. 11

[14] Krzysztof Czarnecki. Variability modeling: State of the art and future directions. In *VaMoS*, page 11, 2010.

[15] Ian J. Davis and Michael W. Godfrey. Clone detection by exploiting assembler. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 77–78, New York, NY, USA, 2010. ACM. 11

[16] J. Davis. *Open source SOA*. Manning, 2009. 8

[17] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987. 12

[18] B. Ganter, G. Stumme, and R. Wille. *Formal Concept Analysis: foundations and applications*. Springer, 2005. 13

[19] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani. Formal reasoning techniques for goal models. *Journal on Data Semantics*, pages 1–20, 2003.

[20] P. Giorgini, J. Mylopoulos, and R. Sebastiani. Goal-oriented requirements analysis and reasoning in the Tropos methodology. *Engineering Applications of Artificial Intelligence*, 18(2):159–171, 2005.

[21] B. Gonzales Baixauli, P. Leite, and J. Mylopoulos. Visual variability analysis for goal models. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 198–207. IEEE, 2005.

[22] Denis Gracanin, Shawn A. Bohner, and Michael Hinchey. Towards a model-driven architecture for autonomic systems. *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:500, 2004.

[23] T.J. Grose, G.C. Doney, S.A. Brodsky, and Inc Books24x7. *Mastering XMI: Java Programming with XMI, XML, and UML.* John Wiley & Sons, 2002. 6

[24] John Grundy, John Hosking, and Warwick B. Mugridge. Inconsistency management for Multiple-View software development environments. *IEEE Trans. Softw. Eng.*, 24(11):960–981, 1998. 14

[25] P. Hazy and R. E. Seviora. Automatic failure detection with separation of concerns. In *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on*, pages 173 –181, april 2007. 14

[26] J.L. Hennessy and D.A. Patterson. *Computer architecture: a quantitative approach.* Morgan Kaufmann, 2003.

[27] J.L. Hennessy, D.A. Patterson, and J.R. Larus. *Computer organization and design.* Kaufmann, 1994. 12

[28] R. Holmes and D. Notkin. Identifying Program, Test, and Environmental Changes That Affect Behaviour. In *Proceedings of the International Conference on Software Engineering (ICSE)*, volume 10, 2011.

[29] IBM. IBM Service Oriented Architecture (SOA). `http://www-01.ibm.com/software/solutions/soa/`. [Online; Last accessed: April-2011]. 7

[30] I. Ivkovic and K. Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. 2004. 13

[31] Igor Ivkovic and Kostas Kontogiannis. Using formal concept analysis to establish model dependencies. In *ITCC (2)*, pages 365–372, 2005. 13

[32] Igor Ivkovic and Kostas Kontogiannis. A framework for software architecture refactoring using model transformations and semantic annotations. In *CSMR*, pages 135–144, 2006.

[33] Igor Ivkovic and Kostas Kontogiannis. Towards automatic establishment of model dependencies using formal concept analysis. *International Journal of Software Engineering and Knowledge Engineering*, 16(4):499–522, 2006. 13

[34] R. Johnson, E. Gamma, R. Helm, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. *Addison-Wesley*, 1:1–2, 1995. 33, 57

[35] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006. 6

[36] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM. 12

[37] Terence C. Lau, Tack Tong, Ross McKegney, Kostas Kontogiannis, Igor Ivkovic, Philip Liew, Ying Zou, Qi Zhang, and Maokeng Hung. Model synchronization for efficient software application maintenance. In *ICSM*, page 499, 2004.

[38] Bixin Li. Managing dependencies in component-based systems based on matrix model. In *Proc. Of Net.Object.Days 2003*, pages 22–25, 2003.

[39] S. Liaskos, A. Lapouchnian, Y. Yu, E. Yu, and J. Mylopoulos. On goal-based variability acquisition and analysis. In *Requirements Engineering, 14th IEEE International Conference*, pages 79–88. IEEE, 2006.

[40] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21:466–471, June 1978. 1

[41] Lei Liu, Stefan Thanheiser, and Hartmut Schmeck. A reference architecture for self-organizing service-oriented computing. In *Proceedings of the 21st international conference on Architecture of computing systems*, ARCS'08, pages 205–219, Berlin, Heidelberg, 2008. Springer-Verlag.

[42] Jim Marino and Michael Rowley. *Understanding SCA*. Pearson Education, 1 edition, July 2009. 8

[43] Microsoft. Understanding service-oriented architecture. `http://msdn.microsoft.com/en-us/library/aa480021.aspx`. [Online; Last accessed: April-2011]. 7

[44] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Ernst Ellmer. Flexible consistency checking. *ACM Trans. Softw. Eng. Methodol.*, 12(1):28–63, 2003.

[45] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 58(2):171–180, September 2001.

[46] OASIS. Service Component Architecture Policy Framework Specification. `http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf`, November 2007. [Online; Last retrieved: April-2011]. 8

[47] OASIS. Service Component Architecture Assembly Model Specification. `http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec.pdf`, January 2011. [Online; retrieved: April-2011]. 8

[48] Object Management Group. Meta Object Facility. `http://www.omg.org/mof/`. [Online; Last accessed: April-2011]. 6

[49] Object Management Group. Unified modelling language. `http://www.uml.org/`. [Online; Last accessed: April-2011]. 6

[50] Abrehet Mohammed Omer and Alexander Schill. Dependency based automatic service composition using directed graph. In *Proceedings of the 2009 Fifth International Conference on Next Generation Web Services Practices*, NWESP '09, pages 76–81, Washington, DC, USA, 2009. IEEE Computer Society.

[51] A. Razavi and K. Kontogiannis. Pattern and policy driven log analysis for software monitoring. In *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 108 –111, 28 2008-aug. 1 2008.

[52] Steven P. Reiss. Incremental maintenance of software artifacts. *IEEE Trans. Softw. Eng.*, 32(9):682–697, 2006. 14

[53] Mehrdad Sabetzadeh and Steve Easterbrook. View merging in the presence of incompleteness and inconsistency. *Requir. Eng.*, 11(3):174–193, 2006. 11

[54] Mehrdad Sabetzadeh, Shiva Nejati, Steve Easterbrook, and Marsha Chehik. Global consistency checking of distributed models with TReMer+. In *Proceedings of the 30th international conference on Software engineering*, pages 815–818, Leipzig, Germany, 2008. ACM.

[55] Rick Salay, John Mylopoulos, and Steve Easterbrook. Managing models through macromodeling. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 447–450, L'Aquila, Italy, 2008. 11

[56] Vivek Sarkar. A concurrent execution semantics for parallel program graphs and program dependence graphs. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 16–30, London, UK, 1993. Springer-Verlag. 12

[57] S.R. Schach. *Object-oriented and classical software engineering.* McGraw-Hill, 2002. x, 1, 2

[58] Roberto Sebastiani, Paolo Giorgini, and John Mylopoulos. Simple and Minimum-Cost Satisfiability for Goal Models. In Anne Persson and Janis Stirna, editors, *Advanced Information Systems Engineering*, volume 3084 of *Lecture Notes in Computer Science*, pages 675–693. Springer Berlin Heidelberg, 2004. 10.1007/978-3-540-25975-6_4.

[59] Janusz Sosnowski and Marek Poleszak. On-line monitoring of computer systems. In *Proceedings of the Third IEEE International Workshop on Electronic Design, Test and Applications*, pages 327–331. IEEE Computer Society, 2006. 14

[60] N. Synytskyy, R.C. Holt, and I. Davis. Browsing software architectures with lsedit. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 176 – 178, may 2005. 11

[61] J. Thai, B. Pekilis, A. Lau, and R. Seviora. Aspect-oriented implementation of software health indicators. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pages 97 – 104, dec. 2001. 14

[62] The Apache Software Foundation. SCA Java Travel Sample 1.0. `http://tuscany.apache.org/sca-java-travel-sample-1x-releases.html`, June 2010. [Online; Last accessed April-2011]. xi, 62, 64

[63] The Apache Software Foundation. Tuscany SCA Java. `http://tuscany.apache.org/sca-java.html`, June 2010. [Online; Last accessed April 2011]. 63

[64] A. van Lamsweerde. Goal-oriented requirements engineering a guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262, 2001.

[65] Shuying Wang and M.A.M. Capretz. A dependency impact analysis model for web services evolution. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 359–365, july 2009.

[66] R. Weaver. The Business Value of the Service Component Architecture (SCA) and Service Data Objects (SDO). *IBM Whitepaper, November*, 2005.

[67] J. Wu and R.C. Holt. Linker-based program extraction and its uses in studying software evolution. In *Proceedings of the International Workshop on Foundations of Unanticipated Software Evolution*, pages 1–15. Citeseer. 11

[68] S. Yacoub, B. Cukic, and H.H. Ammar. A scenario-based reliability analysis approach for component-based software. *Reliability, IEEE Transactions on*, 53(4):465–480, 2004.

[69] Il-Chul Yoon, A. Sussman, A. Memon, and A. Porter. Prioritizing component compatibility tests via user preferences. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 29 –38, sept. 2009. 13

[70] Il-Chul Yoon, Alan Sussman, Atif M. Memon, and Adam A. Porter. Direct-dependency-based software compatibility testing. In *ASE*, pages 409–412, 2007. 13

[71] Yijun Yu, Nan Niu, Bruno Gonzlez-Baixauli, William Candillon, John Mylopoulos, Steve Easterbrook, Julio Cesar Sampaio do Prado Leite, and Gilles Vanwormhoudt. Tracing and validating goal aspects. In *Requirements Engineering, IEEE International Conference on*, volume 0, pages 53–56, Los Alamitos, CA, USA, 2007. IEEE Computer Society. 11