2013

Reasoning with qualitative preferences for optimization of component-based system development

Zachary James Oster
Iowa State University

Reasoning with qualitative preferences for optimization of component-based system development

by

Zachary James Oster

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Iowa State University

Ames, Iowa

2013

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

Many people and organizations have supported and guided me throughout my graduate education, so it is fitting that I should recognize a few of them here. Without their support, my time in the graduate program at Iowa State would not have been as fulfilling or as successful.

First, I thank the National Science Foundation for their partial support of my research through grants CNS0709217, CCF0702758, and CCF1143734. I also thank Glenn Luecke and the rest of the High Performance Computing Group in the IT Services department at Iowa State for introducing me to research in high-performance computing, as well as the support their assistantship (in cooperation with Cray, Inc.) provided.

As important as funding is, it is much more important to have good mentors, and I think I have had one of the best in my major professor Samik Basu. His guidance, patience, and openness have been vital for my growth as a researcher and teacher. I also appreciate the time and advice given by the other members of my committee. Ganesh Ram Santhanam has been a valuable colleague, mentor, and source of inspiration throughout my graduate work; his research on preference reasoning enabled much of the work presented in this dissertation. Shashi Gadia kindly agreed to substitute for another committee member during my final defense of this dissertation on very short notice; I thank him for his generosity on that day and for sharing his wisdom at many other times during my years at Iowa State.

My colleague Michelle Ruse deserves special mention here. Michelle and I have given each other a great deal of advice and encouragement throughout our journeys to the Ph.D., and I have her to thank for helping me through my first independent teaching experience.

Finally, I thank my family for their support of my academic pursuits throughout the years. In particular, I thank my wife Carrie for all of her love, support, and understanding, and I look forward to the wonderful things we will do together in the future.

expresses requirements in two different

ABSTRACT

A component-based system is a set of entities that work together in well-defined ways to satisfy a given requirement specified by the stakeholders for the system. This requirement can be modeled as a set of combinations of traits, which represent acceptable alternatives for providing the required functionality. A system satisfies its requirement if and only if it provides one of the required sets of traits in its entirety. Beyond the requirement, system stakeholders may also have preferences with respect to optional functionality that could be provided by a system, tradeoffs between non-functional properties, or other system design options. This work focuses on integrating support for both qualitative preference reasoning and formal verification into the component-based system design process in order to choose a set of components for the system that, when composed, will (1) satisfy the stakeholders' requirement for the system and (2) provide a set of traits that is optimal with respect to the given preferences. Our primary research objective is to develop a generic, modular, end-to-end framework for developing component-based systems of any type which are correct according to the system requirement and most preferred with respect to the stakeholders' preferences. Applications of the framework to problems in Web service composition, goal-oriented requirements engineering, and other areas will be discussed, along with future work toward integrating multi-stakeholder preference reasoning and partial satisfaction of traits into the framework.

CHAPTER 1. INTRODUCTION

1.1 Component-Based Systems: An Overview

A *component-based system* (e.g., [39, 40]), sometimes called a *compositional system*, can be defined as a set of *components* that are selected from a (possibly large) set of available *entities* and then composed according to one or more composition function(s) to form a system that satisfies a given *requirement* specified by the *stakeholders* in the system. These stakeholders include the owners and operators of the system, but may also include users, developers, marketers, maintenance staff, management, and others as needed in the setting where the system is to be used. Along with research specifically oriented toward component-based software systems (see [38, 39, 40, 47] for some surveys), component-based software development has manifested itself in several important ways in recent years. Web service composition [2], in which small, publicly available services with well-defined and relatively simple interfaces are combined to form useful online software applications, has been used increasingly widely by Web and intranet application developers (e.g., [87]). The widespread adoption of software product line engineering [23] in industry is another notable example of the success of a component-based approach to software development. Both of these approaches to component-based software system development are discussed further in Section 2.1.

Component-based software development is not limited to a few specific paradigms. In fact, most software system development problems can be modeled as instances of component-based system development. As part of a standard system development process, software engineers decompose the overall problem to be addressed into one or more layers of subproblems. Each subproblem is then solved by one or more "components", which may be either previously developed software modules or entirely new code written specifically to solve the subproblem in question. These components are then linked together in some way to form a unified software system.

Developing a system in a component-based manner can have significant advantages over more traditional software development methods where every piece of software in the system is developed specifically for that system. If the inputs, outputs, behaviors, and other properties of each component are clearly and correctly documented and if the components are designed to be interoperable, then using previously developed software components can save a great deal of time and effort [42, 45]. In theory, large systems can be built almost entirely from pre-existing components or subsystems purchased from vendors (commonly called "commercial-off-the-shelf" or COTS software) with only limited effort required to integrate the components into a cohesive system, although such an approach involves significant challenges and risks in practice [41].

1.2 Challenges in Developing Optimal Component-Based Systems

Although there are notable advantages to component-based system development, it is difficult to develop component-based software systems that fully satisfy their stakeholders' needs. Stakeholders express their needs for a specific system to be developed as a *requirement*, which describes the minimum collection of traits or properties that the system must provide in order to be acceptable to the stakeholders. More often than not, the process of determining the requirement for the system is iterative: a set of sub-requirements or even a prototype system is presented to the stakeholders and rejected, the reasons for the rejection are obtained by the system developers and used to refine the sub-requirement set or prototype, and then the cycle is repeated until a satisfactory system requirement or design is identified (or until a deadline arrives, whichever comes first). This cycle of iterative refinement appears to be a natural and unavoidable part of system requirement and preference modeling [27]. As stakeholders develop a better understanding of their needs through various iterations of the stated system requirement, the requirement can be stated more accurately and in greater detail, possibly being decomposed into additional levels of sub-requirements. The problem is that systems are often constructed and delivered before sufficient iterations of this process have been completed, only to find that the delivered system does not satisfy the stakeholders' needs well enough. Frederick Brooks's observation of this phenomenon led him to advise system developers in [15] to "plan to throw one [system] away; you will, anyhow."

Instead of building a system and then throwing it away, we aim to present alternative sub-requirement sets or system designs for stakeholders to compare, evaluate, and refine before choosing a system to build. To do so, the system designers must be able to correctly model each sub-requirement given by the stakeholders in order to provide a strong guarantee that the completed system fully satisfies them. Unfortunately, current component-based system development methods such as those described in Section 2.1 are often limited to representing all parts of the system requirement using a single formal language. While the one chosen formalism may be exactly suited to modeling and verifying one specific type of trait (e.g., low-level behavioral properties such as deadlock-freeness or fairness properties may be modeled using a property written in a temporal logic), it may be difficult or impossible to express other types of traits using that formalism. Therefore, one important challenge in component-based system development is to *coordinate the specification and verification of sub-requirements for a system using multiple formal methods.* Ideally, component-based system development methods should take advantage of natural linkages between the requirements for a system and the sets of components that can satisfy them, helping the system's stakeholders understand the possible implications of changes to the requirements.

Beyond the basic requirement, system designers must also account for *optional traits* that stakeholders would like the system to possess if possible. These optional traits can include certain non-functional properties, optional features, enhanced implementations of required features, and other aspects that are not necessary to fulfill the basic purpose of the system but are desirable to the stakeholders. Because it is usually impossible to provide all of the optional traits that stakeholders desire in a system, it is important for system designers to work together with stakeholders to determine which sets of optional traits the stakeholders prefer most strongly. The system designers can then attempt to construct a component-based system that provides a preferred set of optional traits while fulfilling the essential requirement for the system.

Unfortunately, many commonly employed methods for representing and reasoning with stakeholders' preferences cannot accurately model the full complexity and nuance of stakeholders' preferences. Such preference reasoning methods often follow a set procedure: (1) ask stakeholders to quantify the relative importance of a number of traits, (2) obtain data to quantify the extent to which each design option provides each trait, (3) execute a multi-criteria decision making (MCDM) or multi-objective optimization algorithm with these two sets of numeric data, and finally (4) provide a single "preference" or "quality" value produced by the optimization algorithm for each value. Among other shortcomings, these approaches to preference reasoning assume both that stakeholders consciously know their true preferences and that stakeholders are able to quantify these preferences within an expected level of precision. Such approaches also rely on the assumption that the given MCDM or optimization algorithm is the correct one to rank the given design options for the problem to be solved. Therefore, another significant challenge in developing optimal component-based systems is to *apply more accurate and flexible methods to model and reason with stakeholders' preferences over optional traits, then select system components according to those preferences.* An ideal preference reasoning technique for component-based system development would evaluate the possible design options (i.e., sets of components) with respect to the given preferences in a way that is not unnecessarily dependent on one specific preference aggregation algorithm.

Finally, even when the requirement and preferences of the stakeholders are correctly modeled and a system design that appears to fulfill them is identified, it is still necessary to implement the system by composing the selected components and then verify that the completed system actually does satisfy the given requirement and preferences. As an example, a set of components may be selected so that every part of the overall requirement is provided by some selected component(s), but unexpected interactions between components may prevent the completed system from fulfilling some necessary sub-requirement. Many component-based system development frameworks, techniques, and tools that provide support for verification of the final system against formal requirements have already been presented in the computer science and systems engineering literatures, as we will see in Chapter 2. Few of these approaches, however, provide robust and flexible support for formal specification and verification of system requirements using multiple specification formalisms; instead, they generally restrict themselves to a single formalism. Furthermore, because of the problems inherent in commonly used quantitative preference reasoning methods, existing component-based development approaches that use such methods cannot provide strong formal guarantees that the design choices made during the development process are the best possible with respect to the stakeholders' expressed preferences. To move beyond previous frameworks, the challenge that must be addressed is to *present a unified framework for component-based system development that composes a set of selected components into a unified system, formally verifies that the system fully satisfies its requirement, and ensures that the system provides the optional traits it is committed to provide.*

1.3 A New Framework for Component-Based System Development

In this work, we propose a new framework for component-based system development, which is essentially *a component-based system for developing component-based systems.* Our new framework addresses each of the principal challenges discussed in the previous section: (1) help stakeholders model the given requirement for the system in greater detail by coordinating the formal specification of sub-requirements using multiple formal methods, (2) help system designers accurately represent and reason with stakeholders' preferences regarding optional traits that the system may have, and (3) once a preferred option is identified, construct (realize) a system from the selected components and verify that the resulting system fulfills its commitments regarding the requirement that it claims to satisfy and the optional traits that it claims to possess.

The first challenge is to help stakeholders better express their needs for the system by decomposing the overall system requirement into a number of sub-requirements, which can each be specified using different formalisms. This decomposition into sub-requirements is a natural part of system development [89], but too many component-based system design frameworks make the generous assumption that stakeholders understand and can easily express their specific needs in detail at the start of system development. Additionally, if sub-requirements will be formally verified, most existing frameworks assume that one formal language will be used to specify all sub-requirements. Our framework addresses this challenge by allowing stakeholders to first state several high-level sub-requirements and then decompose each of these into multiple levels of related simpler traits. The overall requirement is eventually encoded as a Boolean combination of low-level traits, which can each be specified using a different specification method. After the selected components are composed into a unified system, the system can then be verified against each low-level trait using an appropriate verification technique for each specification method used.

Once the stakeholders and system designers have identified the problem they wish to solve, the second challenge is to help system designers accurately model and reason with stakeholders' preferences about optional traits of the system in order to focus on the most preferred choices within the space of possible system designs (i.e., sets of components) that satisfy the given requirement. We assume in this work that one or more repositories of components are available and that appropriate search methods exist for computing the set of possible systems that can be created from these components. Based on the information each component provides about the properties it satisfies, it is possible to compute which sets of components are expected to satisfy the overall requirement when composed to form a system. We propose an automated method for computing this set of *correct* system designs, which will save time compared to doing this analysis manually and may also help stakeholders visualize the consequences of their expressed requirement. We also present automated methods for deciding whether a group of set-based qualitative preferences expressed by a stakeholder (or group of stakeholders) is *consistent*. Intuitively, a set of preference statements is consistent if no subset of the stated preferences can be interpreted to mean that a certain set of optional traits is preferred to itself; consistency of preferences is defined more formally in Section 5.3. If the set of preferences is not consistent (i.e., if one preference conflicts with other preferences), our methods automatically identify which preferences are contributing to the inconsistency, allowing the stakeholders to decide which preference(s) to relax or modify in order to achieve consistency.

After a consistent set of preferences is agreed upon, automated qualitative preference reasoning can be applied to determine which system design option(s) will be most preferred based on the given preferences. Note that more than one option may be presented to the stakeholders if multiple options are equally preferred, instead of forcing a decision about the single "best" system based on quantitative preference valuations that may not accurately represent the stakeholders' true preferences. Simply narrowing the space of possible options to this extent may allow the stakeholders to see the option that they most prefer; they may alternatively discover that they dislike all of the "preferred" options, in which case they can reconsider and possibly modify their specified preferences.

When the stakeholders have selected a promising set of components to be composed into a system, the third challenge is to compose the selected components and then verify that the completed system satisfies its commitments regarding the overall requirement that it must satisfy and the optional traits it claims to provide. Our proposed system development framework serves as a "conductor" for this realization and verification process. It guides composition actions to realize the system piece-by-piece, and it coordinates calls to the specified verification methods for each required trait (sub-requirement) and each optional trait that the system is expected to provide based on the descriptions of its components. If the "first-choice" system design fails the composition or verification process, our framework can repeat the process with an alternative that is similarly preferred or slightly less preferred, continuing until a completed and verified system can be returned or until all possible options are exhausted.

Note that this proposed framework does not make system design decisions for the stakeholders. While this may seem like an attractive option for stakeholders and system designers alike, we believe that it is not the role of software engineers or system designers to make major system requirements or design decisions without meaningful consultation with stakeholders. Instead, the proposed framework aims to provide easily understood justifications, similar to an "audit trail", for every decision made throughout the system development process. The overarching objective of our framework is to support, explain, and (when needed) clarify requirements or design decisions that the stakeholders must make in order for the stakeholders and system designers to create a system that fulfills the stakeholders' needs as fully as possible.

1.4 Contributions of This Work

The contributions of this work follow the threefold motivation given in the previous section. They are summarized as follows:

1. We present a general methodology for automated formal analysis of stakeholders' preferences. This methodology incorporates an algorithm for automated identification of inconsistent preferences and generation of feedback to help stakeholders locate and resolve any inconsistencies in their combined set of preferences.

2. We define a full formal framework for modeling, decomposition, and analysis of any given component-based system development problem. This formal framework facilitates modeling of requirements, stakeholder preferences, and their interplay in forming the space of possible solutions to the given system development problem.

3. We automate the coordination of all necessary tasks for realizing a given component-based system, constructing the system's *verification obligation* (i.e., the set of traits that it is expected or committed to provide), and verifying that the completed system truly fulfills its verification obligation.

4. We provide formal correctness (soundness and completeness) results for our preference reasoning and system development techniques to the greatest extent possible.

5. We discuss the implementation details and experimental results for those parts of our overall system development framework that have already been implemented, and we present a "roadmap" for completing and evaluating the implementation of the entire framework.

1.5 Organization

The remainder of this dissertation is organized as follows:

- **Chapter 2**: An overview of previous work in the areas of component-based systems, requirements engineering, and preference reasoning is given, along with commentary on how it has informed our work.

- **Chapter 3**: The basic concepts used in our work, such as traits, requirements, and preferences, are introduced and formally defined. All of these concepts are used throughout the remaining chapters.

- **Chapter 4**: Our existing techniques for realization of a component-based system and formal verification of its properties using multiple formal methods are presented.
- **Chapter 5**: Our work on representing and reasoning with qualitative preferences over traits of a component-based system is explained, and a method for applying these preference reasoning techniques to select preferred system designs is presented.
- **Chapter 6**: A new framework for realizing an optimal (correct and highly preferred) component-based system and verifying that the system satisfies its verification obligation is described in detail. This framework ties together elements presented in Chapters 3, 4, and 5. The usefulness of the framework is demonstrated in the context of developing a component-based system to coordinate the operations of a roadside assistance business.
- **Chapter 7**: The contributions of our work thus far are summarized, along with our plans for long-term future research along these lines.

CHAPTER 2. RELATED WORK

This chapter provides an overview of previous work that has informed our research in this area. Section 2.1 describes previous approaches to the modeling and creation of component-based systems, including significant work toward improving development of two well-known types of component-based systems: Web service compositions and software product lines. In Section 2.2, we provide an overview of related work in the area of software requirements engineering, with a focus on the goal-oriented requirements engineering methodology. Section 2.3 examines the relative merits of various qualitative and quantitative preference modeling and reasoning methods in order to explain our focus on set-based qualitative preferences in this work. The remainder of this dissertation will weave threads from these areas together into our framework for developing optimal component-based systems.

2.1 Component-Based Software Systems

A *component-based software system* is, simply put, a software system that is constructed from a set of pre-existing software components instead of being built with code that is entirely new or unique to that particular system. For decades, both academic and industrial researchers have been trying to find new and improved ways to develop new software systems using previously developed components. One of the challenges inherent in creating a framework for developing systems in a component-based manner is that the word "component" can mean many things. Szyperski [83] provisionally defines a component as having three characteristic properties, namely "that it is a unit of independent deployment; a unit of third-party composition; and it has no persistent state." The Workshop on Component-Oriented Programming at ECOOP 1996 [84] defined a component more precisely as follows:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Numerous frameworks and approaches for component-based system development have been presented in the research literature. A large number of existing component-based system development frameworks are summarized and categorized in [39, 40], while additional formal models for aspects of component-based software development are discussed at length in [38, 47]. Two areas where many of the concepts of component-based systems have been widely and successfully put into practice are Web service composition and software product lines. We introduce work that has been done in these areas in Sections 2.1.1 and 2.1.2 respectively.

2.1.1 Web Service Composition

Web services [2] are self-contained and self-describing programs that are made available for use over a network and are designed for interoperability with other Web services across different platforms. Web service composition [69] is the process of identifying and composing individual Web services into a single *composite service* (or *composition*) in order to provide certain functionality required by the user. However, to truly fulfill the needs of its users, a composite service should also provide a set of *non-functional properties* or *NFPs* (e.g., cost, availability, and throughput; sometimes called *quality of service (QoS)* properties in this setting) that is optimal with respect to the users' preferences. Therefore, the objective of the Web service composition problem is to identify the most preferred possible composite service(s) among those that satisfy the given requirements.

Many Web service composition frameworks have been proposed in the past decade as the potential for using Web service composition for rapid development of new Web-based applications has been recognized. All service composition frameworks are designed to produce composite services that satisfy their stakeholders' functional requirements. Some also account for low-level behavioral constraints or non-functional properties, but very few integrate all three of these aspects in a unified way. We are especially interested in service composition frameworks that apply formal methods to guarantee satisfaction of functional requirements or provision of non-functional properties (or both). These methods, many of which are surveyed in [86], can formally guarantee the correctness of the composite services in terms of their required functionalities. Many of these composition frameworks share a similar basic structure for characterizing and solving the Web service composition problem:

1. Transform each service's specification into a formal model.
2. Translate each functional requirement into a formally specified property.
3. Provide an algorithm for choosing and composing services such that the formal model of the resulting composition satisfies the required properties.

Differences between service composition frameworks are based on the differences in the semantics assigned by each framework to the given descriptions of Web services and properties. In the Roman model used by Berardi et al. [16] and the MoSCoE framework developed by Pathak et al. [68], both the available component services and the desired functionality of the composition are represented semantically as labeled transition systems (LTS). Hamadi and Benatallah [33] model services as Petri nets. Pistore et al. [70], in contrast, model Web service composition as a constraint satisfaction problem, allowing them to take advantage of existing work toward efficient solutions to such problems. Lin et al. [46] treat Web service composition as a planning problem, using description logic [7] to express the desired functionality.

However, each of these approaches to the service composition problem specifies one type of semantics that must be used to interpret the descriptions of the available component services and the desired functionality of the composition. In [59], we introduced a new "meta-framework" for service composition that supports the use of multiple formal semantics, i.e., more than one specification language, to specify and verify functional requirements for a composite service. This meta-framework, which forms part of the theoretical foundation of the component-based system development framework described in this dissertation, is presented in greater detail in Chapter 4.

Several existing approaches to the Web service composition problem are similar to our meta-framework in some respects. Both [46] and [90] handle Web service composition by decomposing a task into individual subtasks and then choosing services that can perform each subtask, as we do in our approach. However, [46] and [90] assume the existence of ontologies that define the semantics of the problem to solve and the properties to satisfy, and they further assume that properties are specified using some form of description logic.

In [53], the “type-aware” Web service composition problem, where the functionality specified by a goal service must be provided and a set of type-correctness constraints must be satisfied, is reduced by Nam et al. to an instance of the Boolean satisfiability problem (commonly known as the SAT problem for short) by taking advantage of a type hierarchy that is included in the problem definition. To the best of our knowledge, this is the only existing approach other than our “meta-framework” in [59] that directly reduces a type of Web service composition to Boolean satisfiability. Unfortunately, the framework in [53] considers only properties that can be expressed as inputs or outputs of the goal service, and it forms compositions based only on compatibility between services. In contrast, our meta-framework relies on the decomposition of the given property specification to generate a solution. It requires neither a type hierarchy nor ontologies to produce a solution, and it supports composition and verification using different formalisms, including description logics and goal services.

In [87], ter Beek et al. focus on verification of functional requirements in a scenario similar to our example in Chapter 4. The service-oriented architecture in [87] is modeled as a set of state machines, which are illustrated as UML-like diagrams. Temporal constraints representing functional requirements and behavioral constraints are specified in the temporal logic UCTL (an extension of CTL) and verified over the diagrams using an on-the-fly model checking tool. Though [87] does not consider non-functional properties, it shows that model checking for requirement verification is feasible in an industrial-scale application of service composition.

However, the service composition frameworks discussed so far in this section do not allow for consideration of stakeholders’ preferences over non-functional (or other optional) properties during the process of selecting components for the composition. Other researchers in the service-oriented computing community have developed methods for identifying compositions whose non-functional properties are optimal with respect to a set of user preferences. Some of these methods are described in [3, 72, 74, 92]. The composition method presented in [92] is representative of many techniques that consider both functional and non-functional properties. In [92], the entire problem is modeled as an integer linear programming problem, where simple syntactic matching of inputs and outputs is applied to form the composite service and quantitative valuations are used to model preferences over NFPs.

Two other methods for service composition that provide robust support for both functional and non-functional properties are the LOEM method [72] and the method of Alrifai and Risse [3]. The LOEM method works by identifying services for individual subtasks, applying local selection to identify several services for each subtask that are likely to produce a preferred composition, enumerating all possible compositions that can be created from the services identified for each subtask, and then choosing the best composition using mixed integer programming (MIP). The method presented by Alrifai and Risse in [3] decomposes global non-functional constraints into local constraints and then applies MIP to determine the optimal composition to solve the given problem. Unfortunately, both [3] and [72] deal specifically with quality-of-service (QoS) properties, leaving other types of optional traits unsupported. In addition, although [72] discretizes services' QoS property valuations into a finite number of ranges, the methods given in [3] and [72] are restricted to using only quantitative valuations for non-functional properties, since compositions are selected by maximizing a utility function. However, this problem is known to be NP-hard for non-functional properties with quantitative valuations [3]. Composition methods that consider non-functional properties also tend to lack the flexibility that our meta-framework from [59] provides in allowing the use of different formalisms to specify requirements and appropriate composition techniques.

In contrast, Santhanam et al. [74] use qualitative valuations to select services to compose based on a set of preferences among NFPs expressed by the stakeholders in a different language for qualitative preferences, namely tradeoff-enhanced conditional preference networks (TCP-nets) [14]. This approach informed our extension of our previous "meta-framework" [59] to handle global qualitative preferences over non-functional properties [62]. After decomposing the functional requirements for a composition as in [59], our method in [62] then applies NFP preferences to perform local selection of preferred component services. The result is a service composition framework that will always identify a composition which satisfies functional requirements and is at least as preferred as any other composition that satisfies the same functional requirements, as long as all component services being considered are compatible with each other. The strategy presented in Chapter 5 of this work for handling NFP preferences is inspired in part by [62] and [74].

A service composition method that specifically focuses on verifying that a composition provides a given set of NFPs is the method of Sun et al. in [82]. Unlike our meta-framework in [62], the method in [82] does not attempt to satisfy preferences over various NFPs that the system might provide, nor does it select services to participate in candidate compositions according to the NFPs that they do or do not provide. This method applies an existing service search and composition technique to identify the space of possible service compositions that fully satisfy the given functional requirement for the system. Although the example in [82] uses a composition algorithm from [66], any technique that produces a finite-state automaton for each composition could potentially be used in this context, as in our meta-framework. Each required NFP is then modeled as a finite-state automaton, after which the synchronous product of the automata representing all NFPs to be verified is computed. Sun et al. show that a given service composition provides a certain non-functional property (or composition of properties) if and only if the automaton for the composite service simulates the automaton for the property [82]. This verification method for NFPs can be freely applied both within the context of our meta-framework as described in [59, 62] and as part of the component-based system development framework presented here.

The vast majority of service composition frameworks that handle non-functional properties suffer from one or more serious drawbacks that limit their applicability, such as the following:

- They frequently treat all functional requirements as mandatory, choosing between several versions of the same low-level functionality instead of considering diverse low-level implementations of the same high-level functionality.
- They typically do not focus on verifying low-level behavioral constraints.
- They often consider only NFPs that affect the “quality of service” of the composition but ignore other important NFPs, especially those that are not easily quantified.

- They typically require that the names and/or structures of a service's accepted inputs and available outputs must exactly match those of other services.

One existing service composition framework that successfully integrates verification of stakeholders' functional requirements, low-level behavioral constraints, and non-functional properties is the TQoS algorithm [31]. TQoS considers both transactional and QoS properties when composing services, selecting services that have locally optimal QoS for each part of the desired functionality. Additionally, TQoS guarantees by construction that any composite services it produces satisfy a standard set of transactional constraints. The VeriComp service composition framework, introduced in [58] and used as a running example in Chapter 6 of this work, is another framework that was developed specifically to unify verification of these three types of composite service traits.

2.1.2 Software Product Lines

Software product lines [91] are families of software systems that are designed to take advantage of their common aspects and predicted variabilities. Systems that are part of a software product line share common assumptions and often share common code in their implementation but differ from each other in significant ways. For example, different software products in a product line may provide different sets of features, varying levels of service, or compatibility with different hardware systems. Developing software systems as a single product line can produce systems that are more reliable and better customized than systems that are developed from the ground up, while saving effort that might otherwise go into reimplementing components or features that were developed for a closely related software system.

The concept of a software product line has a relatively long history, which can be traced back to David Parnas's concept of "program families" [64]. Parnas defined a program family as a set of programs that have sufficiently many properties in common that it is best to first examine the shared properties of all programs in the set before studying the specific properties of individual programs. Programs in a program family have a common "ancestor", which represents the programs' shared properties. It follows that a new member of the program family can be developed by starting from the design decisions (and perhaps the code) contained in the ancestor program, then completing the new program with additional decisions for the specific environment where it will be applied.

Software product lines can be developed using a process called *scope, commonality, and variability (SCV) analysis* [23]. This process determines the product line's *commonalities*, which are the characteristics (e.g., basic features, components, or design decisions) that are shared among the entire set of systems in the product line, along with its *variabilities*, which are assumptions that are not true of all systems in the product line or other aspects that vary among different members of the product line. Each variability can take a certain range of values; a software product can be generated, sometimes automatically or semi-automatically, by specifying a particular set of values for the variabilities and taking advantage of previously implemented commonalities. The Family-Oriented Abstraction, Specification, and Translation (FAST) approach to developing software product lines [30], which makes use of SCV analysis, is one approach that has been successfully adopted for product line development in industry [6].

Having a well-defined specification for the scope, commonalities, and variabilities of a software product line makes it possible to create and deliver versions of standard systems that are customized for specific stakeholders' needs. This could be accomplished in some cases by selecting additional components to couple with the base system, where the selection of components is guided by stakeholders' preferences over the variabilities of the system. Many researchers in the software product line engineering community have proposed techniques for such preference-based customization within a software product line. There has also been an effort in the software product line research community to automate verification of requirements among the many software systems that belong to a given product line.

As in the Web service composition setting, a number of formal specification and verification languages have been applied to address these problems. Mannion [49] uses a hierarchical lattice of requirements that are specified in first-order propositional logic to verify a product line model or particular software systems created from that model. Batory [8] employs a formal grammar to translate details of product-line features that are specified within a feature diagram [36] into propositional-logic statements, then applies a Boolean satisfiability (SAT) solver to determine whether the selected features are compatible with one another. To assist in this process, Schobbens et al. [79] provide a unified formal semantics for seven types of feature diagrams. The relationship between various types of feature diagrams and propositional-logic statements is examined in greater detail by Czarnecki and Wasowski in [25], and the feasibility of this approach is explored by Mendonça et al. in [51] and more recently by Apel et al. in [5].

However, existing approaches for product-line verification tend to share many of the same drawbacks as approaches for verification of Web service compositions. In particular, most current applications of model checking for product line verification are restricted to verifying the satisfiability of propositional logic statements, although Cordy et al. [24] have proposed an approach that can handle numeric-valued features as well as features that appear multiple times in the same software product. There is therefore a need for a product line verification framework that can verify requirements specified in diverse formal languages, similar to our service composition meta-framework in [59, 62].

2.2 Goal-Oriented Requirements Engineering

Although the discipline of requirements engineering (RE) is wide-ranging and difficult to summarize in a single sentence, van Lamsweerde [89] broadly defines requirements engineering as "a coordinated set of activities for exploring, evaluating, documenting, consolidating, revising and adapting the objectives, capabilities, qualities, constraints and assumptions that the system-to-be should meet based on problems raised by the system-as-is and opportunities provided by new technologies." The practice of requirements engineering tries to avoid the problem of developing "the wrong system", i.e., a system that does not fulfill the needs of those who are involved with building, maintaining, owning, or using it. A *stakeholder* in the system is defined by van Lamsweerde [89] as "a group or individual affected by the system-to-be, who may influence the way this system is shaped and has some responsibility in its acceptance." If a system does not satisfy the needs and desires of its stakeholders, large amounts of time and money may be wasted in either reconfiguring the delivered system or, in the worst case, discarding the new system completely and returning to the previous state of affairs.

Goal-oriented requirements engineering (GORE) [26, 88, 93] is a requirements engineering methodology in which requirements for a system are defined in terms of goals or objectives specified by the system's stakeholders. The basic infrastructure of the GORE methodology as applied in our research is a *goal model*, which is a formal model based on an AND-OR graph that encodes the relationships that exist among a set of *(required) goals* G^R and a set of *tasks* G^T; a goal model may also incorporate a set of *optional goals* G^O. A (required) goal describes a condition, outcome, or state of the world that must be achieved [93], while a task indicates a certain activity that an actor performs to fulfill or realize a required goal in full or in part [44]; in other words, each task realizes or operationalizes a goal.

Within the AND-OR graph at the core of the goal model, the goal located at the root node of the graph (called the *root goal*) corresponds to the overall required functionality of the system, while goals at intermediate levels of the graph (internal nodes) express portions of the system's functionality or alternatives for providing that functionality.[1] Each goal may be refined into one or more subgoals through either AND-decomposition, where a goal is divided into subgoals that must all be satisfied in order to fulfill the original goal, or OR-decomposition, where each subgoal indicates an alternative way to fully satisfy the original goal. Because tasks specify (in full or in part) how particular goals will be fulfilled, they form the leaf nodes of the goal model, where each task's parent is the goal that it contributes to realizing. A *correct design* in this context is a set of tasks that, when taken together, are sufficient to fully satisfy the root goal.

Optional goals represent conditions that are desirable but are not vital to the system's correctness [44], such as non-functional properties. Optional goals need not be organized in a unifying structure (though this is possible), but they are connected to the goal tree by *contribution links*. Each link is labeled with MAKE (++) if the required goal supports (contributes positively to) the optional goal or BREAK (--) if the required goal denies (contributes negatively to) the optional goal. An optional goal in our model is satisfied if and only if it has (a) no incoming BREAK links from any satisfied required goal and (b) at least one incoming MAKE link from any satisfied required goal.

[1]For clarity of explanation, all example goal models used in this chapter and throughout the dissertation form trees. Our techniques can be easily extended to work with any goal model that forms a directed acyclic graph (DAG). In a goal model that is not a single tree, multiple "root" nodes may exist; each root node represents the overall requirement for a different system, so only one root node is ever considered at a time.

A GORE goal model can be represented graphically for easier comprehension by both requirements engineers and system stakeholders. Figure 2.1 shows a graphical representation of a goal model, which is slightly modified from [80], that describes the hierarchy of required goals, tasks, and optional goals for the public transportation system in the Trentino region of Italy. Unshaded ovals indicate (required) goals, hexagons indicate tasks, and shaded ovals indicate optional goals. Additionally, each goal node is annotated with AND or OR, which denotes the way in which the goal is decomposed into subgoals. For instance, the root goal *Supply Public Transport Service*, which represents the required overall functionality of the system, is AND-decomposed into four subgoals. Each of the subgoals must be completely fulfilled in order to realize the root goal. In turn, this implies that the root goal is satisfied if and only if a set of tasks (leaf nodes, which are drawn as hexagons in Figure 2.1) are selected from the goal model and implemented so that the combination of tasks is sufficient to satisfy the AND-OR decomposition of the root goal. We have not considered a similar AND-OR structure for the optional goals in order to simplify our explanation; however, there is no constraint that prevents such decomposition of optional goals. Any contributions of required goals or tasks to optional goals are denoted by dotted edges. For instance, the task *Evaluate Alternative Resources* positively contributes to the optional goal *Minimize Cost Per User*, while the goal *Invest in the Service* positively contributes to the optional goal *Satisfy Users* but negatively contributes to the optional goals *Minimize Cost Per User* and *Increase Profits*.

It should be noted that other interesting and useful concepts can be incorporated into goal models, e.g., softgoals (which cannot be simply "satisfied" or "unsatisfied" in the usual Boolean sense [93]), temporal ordering of tasks, and labeled contribution links indicating the degree by which a softgoal or optional goal is supported or denied by a goal or task (partial satisfaction semantics). Although we have not considered these additional concepts at this point, our techniques for preference specification and analysis are complementary to these concepts and can be directly combined with them in the future. Our plans to add support for these concepts to our system development framework will be discussed in Section 7.2.

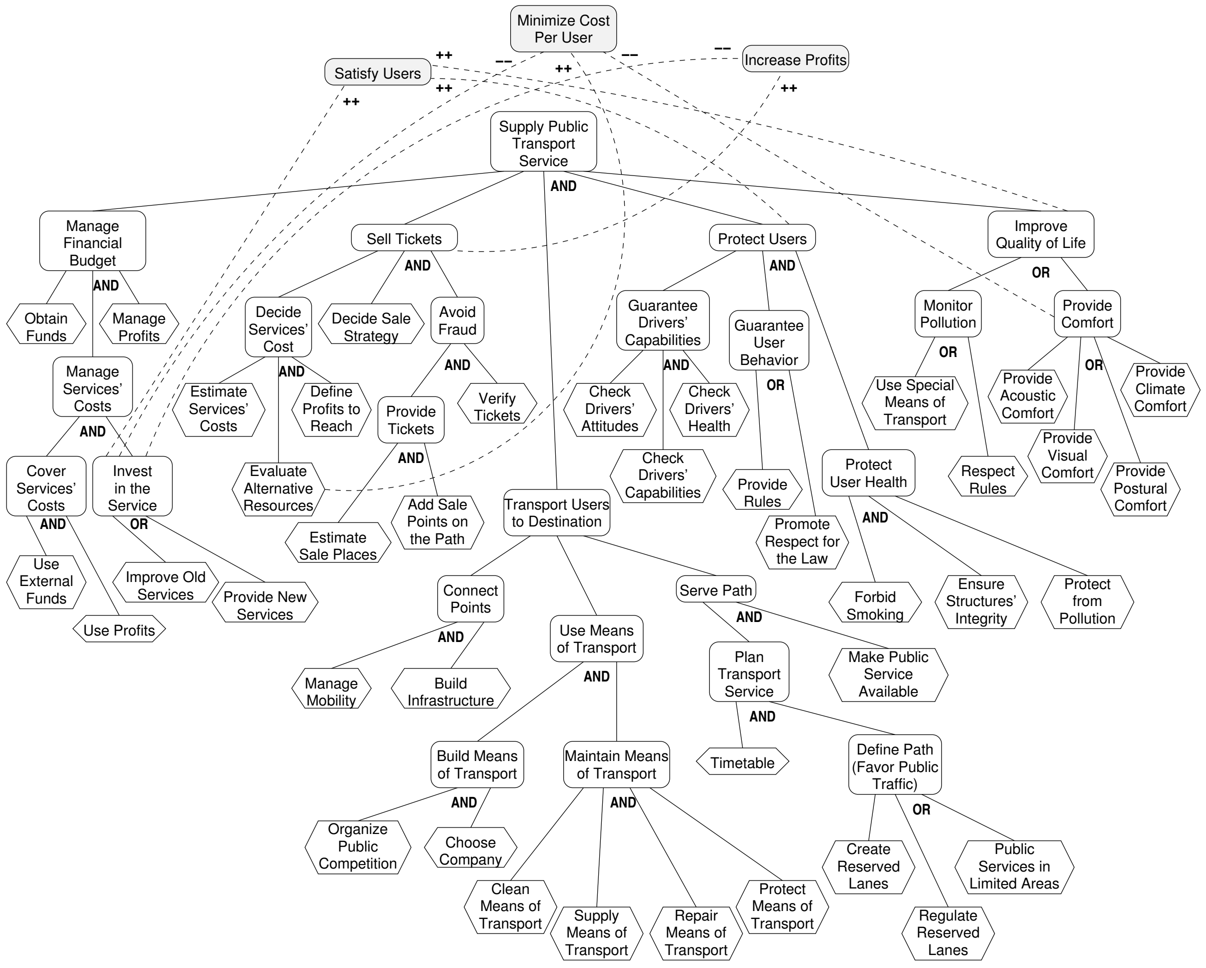


Figure 2.1 Trentino Transport goal model, taken from [80]

2.3 Preference Reasoning

2.3.1 Quantitative Methods

While any preference reasoning or multi-criteria decision making (MCDM) method can be used to optimize the selection of traits to be satisfied by a component-based system, some types of methods are better suited for this purpose than others. The vast majority of component-based system development frameworks use quantitative methods, such as utility theory [37] or the Analytic Hierarchy Process (AHP) [73], to represent and reason with preferences. These methods rely on stakeholders providing exact valuations to indicate the relative preference of the possible values for each trait and/or the relative importance of the traits under consideration. The preference valuations are then used to formulate a numerical optimization problem, which is then solved using integer linear programming or some other well-understood algorithm. Ideally, this process should establish a correct total order over possible system designs.

Quantitative preference reasoning methods have proven useful in a number of decision-making settings in industry, such as the applications described in [40, 73]. These methods are particularly useful in situations where easily quantifiable traits (e.g., cost, capacity, utilization) are being considered and where it is natural to specify preferences between pairs of individual traits (as opposed to preferences among larger sets of traits). However, when designing a component-based system, one cannot assume that the system's traits are easily quantifiable nor that all preferences are pairwise between individual traits. As a result, quantitative methods have several notable drawbacks in this setting:

1. It is easy for a stakeholder to know with certainty that one trait A is more important than another trait B. It is much more difficult for a stakeholder to be certain about *how much* more important A is than B, and in fact stakeholders often do not consciously know this information [27]. The use of quantitative preference valuations requires stakeholders to make the latter type of assertion, directly or indirectly specifying numeric weights that reflect the relative importance of the various traits. Given that the meaning of "importance" is neither uniform across all preference reasoning methods nor well-defined in most methods [18, 78], it is entirely possible that the given preference weights will fail

to reflect the stakeholders' true views on the relative importance of the criteria being considered.

2. Many multi-criteria decision making (MCDM) frameworks, including the popular Analytic Hierarchy Process (AHP) [73], require stakeholders to specify pairwise preferences for all pairs of traits under consideration. This work can be time-consuming if many traits are being considered, and it may seem to be entirely unnecessary drudgery from the stakeholders' perspective [27]. To save time, important traits may be omitted, arbitrary preference valuations may be assigned for pairs that are considered less important, or other shortcuts may be taken, compromising the reliability of the decision-making method's results [13]. Some recently developed quantitative methods, such as Conditional S-AHP [57], do address this particular shortcoming.

3. Stakeholders may not have meaningful preferences between some pairs or sets of traits. Instead of representing *indifference* (i.e., lack of preference) between traits, quantitatively-valued MCDM methods may consider these traits to be equally preferred. Because these decision-making methods may consider such weak or nonexistent preferences to be just as important as stronger preferences, the preference model may not accurately reflect the stakeholders' true preferences (or lack thereof); this may distort the results in unexpected ways [18].

4. Because almost all MCDM methods define a *total order* over the set of options, they may report that a given decision is clearly the most preferred when, in reality, several other possible decisions may be "approximately" equally preferred (or even more preferred) to the chosen decision [13].

Such sources of hidden error can give stakeholders a *false sense of precision* about the results that these methods return. The single "most preferred" system design identified by one of these methods can obscure alternative system designs that might provide a more acceptable solution to the system design problem. On the other hand, the use of arbitrary or poorly-quantified preference valuations and the omission of important traits may significantly reduce stakeholders' confidence in the eventual results. Consequently, although quantitative methods are useful for naturally quantifiable preferences (e.g., costs), using such methods for other types of preferences may produce inaccurate or misleading results if proper care is not taken [13, 18, 27].

2.3.2 Qualitative Methods

To avoid the drawbacks associated with using quantitative preference reasoning methods for component-based system development, our research focuses on reasoning with *qualitative preference valuations* (or simply *qualitative preferences*) to optimize the selection of traits to be satisfied by a component-based system. Qualitative preference reasoning techniques identify different levels of preferences between traits or sets of traits, rather than absolute preference values as used by many quantitative techniques. Many qualitative techniques can also model a lack of preference (i.e., indifference) between traits. The primary benefit of using qualitative instead of quantitative preference valuations for reasoning with system stakeholders' preferences is that the uncertainty inherent in these statements can be incorporated into the semantics of the preference model. Qualitative MCDM techniques generally define a *partial order* over the set of available design options, indicating when there is no clear preference between two or more possible system designs.

One qualitative preference language that is useful in many applications is *conditional preference networks (CP-nets)* [11]. CP-nets compactly represent preferences among several options by using the fact that a stakeholder's preference regarding one attribute is often entirely or conditionally independent from the stakeholder's preference regarding other attributes. A CP-net consists of a directed graph over a set of (qualitative) preference variables, where each variable is represented by a node in the graph and each edge from a node representing a variable A to a node representing a variable B indicates that preferences regarding variable B are dependent (i.e., conditioned) on the value of variable A. Every node in the CP-net graph is annotated with a *conditional preference table*, which specifies preferences over the values of that node's variable given each possible combination of values for the parent nodes' variables.

The full set of preferences implied by the CP-net are expressed in an *induced preference graph*, whose nodes are all possible combinations of valuations of all preference variables being considered. The preference graph induced by a CP-net contains an edge from node C to node

D if and only if the combination of preference valuations C is preferred to the combination D according to the CP-net. Tradeoff-enhanced conditional preference networks (TCP-nets) [14] extend the CP-net formalism by allowing stakeholders to specify the relative importance of the various attributes in the preference model. The resulting tradeoffs between preference variables are then incorporated into the preference reasoning algorithm in a consistent way.

Both CP-nets and TCP-nets rely on a *ceteris paribus* (all else being equal) semantics, which assumes that any given preference statement holds true all else being equal. In other words, any preference over the possible values of one trait is considered to be independent of all preferences over any other trait's possible values unless explicitly specified otherwise. This *ceteris paribus* semantics greatly reduces the size of the preference space to be considered while it limits the amount of information that stakeholders must provide about their preferences, helping combat some of the drawbacks of quantitative preference valuations given previously.

Although CP-nets and TCP-nets offer a great deal of flexibility in terms of the preferences they can model, both languages are restricted to modeling preferences between single properties. For example, the preference "given low cost but not high security, I prefer high availability to the combination of strong access control and rapid updates" cannot be expressed in a CP-net nor a TCP-net. *Conditional importance networks (CI-nets)* [12] allow such *set-based* preferences to be specified in an intuitive way. A CI-net consists of preference statements of the form, "if the propositions in set S^+ are true and those in set S^- are false, then it is preferred to have the set of propositions S_A be true instead of those in set S_B, all else being equal [*ceteris paribus*]." By default, having more true propositions is preferred to having fewer true propositions, all else being equal (though this preference may be reversed if appropriate for the scenario); however, specific preference statements may override this default preference. As in [T]CP-nets, the preference statements in a CI-net together with the default preference form an induced preference graph that indicates preferences between sets of propositions. The syntax and semantics of CI-nets are explained in greater detail in Section 5.2.

CI-nets have played an important role in our research. In [60, 61], we used CI-nets to represent preferences over optional goals in goal models for software systems, while our VeriComp Web service composition framework [58] modeled preferences over quality-of-service traits and other non-functional properties of composite services. CI-nets are not only useful for developing compositional systems, though. In [63], we modeled a client's preferences regarding the relative sensitivity of various sets of authentication credentials using a CI-net as part of an algorithm for minimizing the sensitivity of credentials to be disclosed in order to access an online service. All of these methods relied heavily on techniques for using model checking to verify the consistency of the given preferences and construct a partial order over the set of possible outcomes (sets of trait valuations), which were developed by Santhanam et al. in [75, 76, 77].

CHAPTER 3. COMPONENT-BASED SYSTEMS: DEFINITIONS AND CORE CONCEPTS

In our work, a *component-based system* is defined as a collection of individual *entities* that are linked or composed together in order to satisfy a *requirement* specified by the system's stakeholders. Each entity has a set of *traits* or properties that it possesses or satisfies. Some of these traits are classified as *required traits*, which form part of the requirement for the system; in fact, the overall system requirement is expressed in our work as a Boolean combination of these required traits. Other traits, classified as *optional traits*, may not contribute toward the system requirement but may make the resulting system more desirable to the stakeholders.

When developing a component-based system, entities may be chosen to be components of the system if they contribute toward satisfying part or all of the system requirement. If there exist many available entities that can make similar contributions to satisfying the requirement, *preferences* between various sets of traits that may be satisfied by the entities are used to determine which entity to include as a component of the system. Our overall objective in this work is to identify, create, and verify a component-based system that provides both (1) a sufficient set of required traits so that the given requirement is fully satisfied and (2) a set of optional traits that is at least as preferred by the stakeholders as any other system that satisfies the same requirement. In this chapter, we formally define and informally describe each of these core concepts of our component-based system development framework.

3.1 Entities, Components, and Systems

An *entity* is a possible component of the system. Let E denote the set of all entities under consideration while developing a given component-based system. A *component* of the system is simply an entity in E that has been selected for inclusion in the system. The set of entities considered for participation in the system depends on the problem domain being addressed. We assume that either (1) the entities in E are known before beginning the system development process or (2) there exists a way to quickly (perhaps automatically) discover the set of available entities during the development process.

A *system* (more specifically, a *component-based system*) S is a composition of components that have been selected from the available set of entities E. Let f be a problem-specific composition function, which defines the method(s) by which entities may be composed to become components of a system. Given such a function f and a set of components C chosen from the set of available entities ($C \subseteq E$), we formally define a system as $S = f(C)$.

3.2 Traits

We define a *trait* or *property* to be a triple $\psi = \langle \varphi, F, M \rangle$, where φ is the specification of the trait (property), F is the formal language or other method used to specify φ, and M is an automated or manual *verification method* that decides in a correct (i.e., sound and complete) manner whether a given system S satisfies the property φ as specified using F (in notation, $S \models_F \varphi$). In other words, F defines the semantics used to determine whether φ is satisfied by a given entity or system, while M is a decision procedure that correctly verifies whether a given system (i.e., composition of one or more components) satisfies the trait as specified using the semantics given by F. Given a correct M, we say that a system S *provides* or *satisfies* a trait $\psi = \langle \varphi, F, M \rangle$ if and only if M decides that $S \models_F \varphi$; for simplicity, we write $S \models \psi$ to mean $S \models_F \varphi$.

Each trait under consideration is a concrete expression of an abstract property that a system may possess or provide. This broad definition allows any possible property of a system or component, whether optional or mandatory, to be modeled as a trait within our framework. A task that must be performed to help accomplish the system objective (as in goal-oriented requirements engineering [88, 93]) can be modeled as a trait, and a required low-level behavior such as an expected input/output sequence can also be modeled as a trait. Similarly, a systemwide high-level requirement to ensure that a certain minimum number of users can use a system simultaneously is considered a trait, and a requirement that a system's access control component must encrypt all login names and passwords is also considered a trait. If various encryption algorithms are being considered, each algorithm can be treated as a separate trait; stakeholders can then specify preferences over the available encryption algorithm traits (as described in Section 3.4).

While each abstract trait always remains the same, the formalism F used to concretely specify each trait may change in order to more accurately represent the intent of the stakeholders. For example, the system designers may determine in consulting with stakeholders that a low-level temporal property that was previously specified in Linear Temporal Logic (LTL) [71] would better express the stakeholders' true meaning if it were specified as a Computation Tree Logic (CTL) [21] statement instead, because of the differences between the semantics of LTL and CTL. Further, the method M used to verify a given property may be freely replaced with any other method that correctly verifies the same property.

Let Ψ denote the set of all traits under consideration as part of a given system development problem. This set of traits will vary for each problem domain where this solution framework is applied. As we will see in the next section, it is generally not necessary for a system to provide all traits in Ψ in order to satisfy the overall system requirement and be acceptable to its stakeholders. Any system can be expected to provide a (possibly empty) subset of all traits under consideration in a given problem. We define the set of traits provided by a given system S as $\Psi_S = \{\psi \in \Psi : S \models \psi\}$.

Note that the composition of any two components or systems S_1 and S_2 must not be assumed to provide the same set of traits as the union of the individual components' or systems' traits: in notation, for any property ψ, $S_1 \models \psi \wedge S_2 \models \psi \not\Rightarrow f(S_1, S_2) \models \psi$. This is because interactions between the components of a system may generate unanticipated side effects that prevent the system from satisfying ψ; conversely, such interactions may give rise to so-called "emergent" properties of the system that are not provided by any of its single components [10]. In our view, it is reasonable to anticipate that a system will provide any properties that its components provide, and in fact our "meta-framework" for identifying system designs that are likely to satisfy a given requirement (presented in [59] and in Chapter 4) relies on this approach.

However, in practice it is necessary to verify that the final system still provides the same traits that its individual components satisfy. This requires verifying a system against its "verification obligation" as defined in the next section.

3.3 System Requirement

Every system design problem involves an overall requirement that must be satisfied by the system in order for it to be accepted by its stakeholders. The overall requirement for the system is generally decomposed into sub-requirements, which each specify a part of the functionality or a non-functional property (such as maximum cost or minimum quality of service) that the system is required to provide. Some of these are *high-level* sub-requirements, which describe functions or use cases that the system must provide, standards for non-functional properties that the system must meet, or similar system properties. High-level sub-requirements must be verified in most cases by reference to high-level descriptions of the system and its components or through empirical observation of the completed system. Others are *low-level* sub-requirements, which formally express temporal or other logical properties that the system or its components will be required to possess. Low-level sub-requirements must be verified by applying an appropriate formal verification tool or technique to prove whether a low-level behavioral specification of the component(s) or system satisfies the given sub-requirement.

In our model, the overall requirement for any component-based system can be decomposed into a Boolean combination of sub-requirements along an AND-OR graph [54], where the overall requirement and each sub-requirement form the nodes of the graph. Each higher-level "parent" sub-requirement in the AND-OR graph may be further decomposed into lower-level "child" sub-requirements using AND-decomposition or OR-decomposition; these decompositions define the edges of the AND-OR graph. In AND-decomposition, the parent sub-requirement is satisfied if and only if all of its child sub-requirements are satisfied, while in OR-decomposition, the parent sub-requirement is satisfied if and only if one or more of its child sub-requirements are satisfied. Sub-requirements that cannot be decomposed any further, whether high-level or low-level, are represented as traits as described in the previous subsection; these form the leaf nodes of the AND-OR graph. AND-OR decomposition of high-level system requirements is a standard part of the goal-oriented requirements engineering (GORE) methodology [88, 93]. However, one of our contributions in this work is a novel technique for AND-OR decomposition and verification of low-level requirements that must be verified using different verification techniques, which was initially presented in [59] and is described further in Chapter 4.

Following the structure of the AND-OR graph, the overall system requirement R for any system can be represented as a set of one or more sets of traits r_i, where each set of traits denotes an alternative sufficient set of sub-requirements that will fully satisfy the overall system requirement. This overall requirement is satisfied if and only if every trait in any one of the sub-requirement sets r_i is satisfied by the proposed system. Observe that if each trait is considered to be a Boolean (true/false) proposition, then the requirement R is defined as a disjunction of conjunctions of propositions. Therefore, the requirement R can be viewed as an instance of the Boolean satisfiability (SAT) problem.

Formally, the *requirement* for a system is a non-empty set of non-empty sets of traits $R \subseteq 2^{\Psi}$, such that a system S will be accepted by its stakeholders if and only if every trait in any set of traits in R is satisfied by that system. (Note that if requirements R where $R = \emptyset$ or $\emptyset \in R$ were allowed, such requirements would always be trivially satisfied.) In notation:

$$S \models R \Leftrightarrow \exists r_i \in R : \forall \psi \in r_i : S \models \psi \tag{3.1}$$

Observe that it is sufficient for a system to satisfy only one set of traits $r_i \in R$, so different systems may fulfill the requirement R using different sets of traits. The *verification obligation* $vo(S, R)$ indicates the sets of traits that the system S can be expected to satisfy in order to fulfill the requirement R. Formally, $vo(S, R) = \{r_i \in R : r_i \subseteq \Psi_S\}$. Every trait in any one of the sets of traits in $vo(S, R)$ must be verified to ensure that the system S satisfies the requirement R as defined in Equation 3.1. This verification obligation is useful because it indicates the minimum level of verification required to ensure that the system S is acceptable to the stakeholders.

A *correct system* is a system that satisfies its overall requirement. Formally, a system S is correct for the problem specified by requirement R if and only if $S \models R$ as defined in Equation 3.1. This requirement R must be expressible as a Boolean combination of traits (as defined in Section 3.2), which may be obtained using AND-OR decomposition as described previously. We will describe in Chapter 6 how our component-based system design framework identifies the verification obligation for every set of components that is likely to satisfy the overall requirement (Section 6.3) using a technique based on satisfiability (SAT) solving. Once an optimal (most preferred) set of components is selected, the components are composed and the resulting system is verified against its verification obligation using the methods specified for each trait in the obligation for that system (Section 6.5).

3.4 Preferences

The system stakeholders may express *preferences* over various traits of the system by defining a *dominance relation* $\succ : 2^{\Psi} \rightarrow 2^{\Psi}$ as follows. For all non-empty sets of traits $\Psi_1, \Psi_2 \in 2^{\Psi}$ such that $\Psi_1 \cap \Psi_2 = \emptyset$, $\Psi_1 \succ \Psi_2$ if and only if any system with traits Ψ_1 is preferred to (dominates) any system with traits Ψ_2, all else being equal (*ceteris paribus*). This formulation is modeled on the example of [75]. The semantics of the dominance relation are problem-specific, so any appropriate preference modeling language may be selected for the domain where this system-development framework is being applied: for example, in Chapters 5 and 6, CI-nets [12] are used to model qualitative preferences. However, we require that the relation $\succ$ must define a partial order over the set of all possible sets of traits, i.e., the powerset of all traits 2^{Ψ}.

Note that no distinction is made between "required traits" and "optional traits" in this definition: preferences may be specified over traits that contribute toward satisfying the system requirement as well as traits that are completely optional. This allows stakeholders to indicate that certain ways of satisfying the overall system requirement may be more preferred than others. It may also avoid problems if a previously required trait must be reclassified as optional or vice versa based on input from stakeholders.

The dominance relation over sets of traits $\succ$ can be extended in an intuitive way to also define a dominance relation $\succ : 2^{E} \rightarrow 2^{E}$ over possible systems (i.e., sets of entities) in the following way. Given any two sets of entities $C_1, C_2 \subseteq E$ and two sets of traits $\Psi_1, \Psi_2 \subseteq \Psi$, where $C_1 \neq C_2$, $\Psi_1 \neq \Psi_2$, $C_1 \models \Psi_1$, and $C_2 \models \Psi_2$, we define $C_1 \succ C_2$ if and only if $\Psi_1 \succ \Psi_2$ as defined previously. Note that the same $\succ$ notation is used for both of these dominance relations to indicate the tight coupling between them.

A system S is *non-dominated* if and only if there exists no other system S' such that $S' \succ S$, i.e., S' dominates (is preferred to) S. A *most preferred correct system*, or equivalently an *optimal system*, is a correct system that is also non-dominated with respect to the set of all other correct systems. Note that there may exist multiple most preferred correct systems for a given requirement and preference set, since this definition does not require that such a system must be strictly preferred to (i.e., dominate) all other correct systems.

Although the definition of preferences given in this section is compatible with the use of quantitative preference reasoning techniques, our overall objective in this work is to use qualitative preference reasoning techniques to optimize the selection of correct systems in order to identify the most preferred correct system(s) for solving a given problem. In Chapter 5, we will apply qualitative preference reasoning based on a dominance relation as described here to accomplish this objective.

CHAPTER 4. SYSTEM MODELING, REALIZATION, AND VERIFICATION USING MULTIPLE FORMAL METHODS

As explained in Section 2.1, most component-based system development frameworks expect that the entities they will be composing will be relatively homogeneous, with all verifiable requirements specified in the same formal language (if one is used). This is generally done with a view toward a unified verification process, as all requirements can then be verified using the same technique. If all functional requirements for the system can be correctly expressed using the chosen formal language, then this is not a problem. For example, if we are only concerned with properties that specify the outputs to be returned when certain inputs are received, these properties can be expressed effectively using a description logic. Similarly, if we are only concerned with specifying the system's desired temporal behavior, these properties can be expressed clearly with state machines or temporal logic.

On the other hand, if the desired functionality includes both types of properties, then using only one formalism (description logic or temporal logic) will not be sufficient to fully express the functionality. One way to avoid this problem might be to obtain a more general, more powerful formalism, but this is not always possible. Furthermore, even when such a formalism can be identified, we claim that verifying the required trait using that formalism may be computationally expensive or intractable. For instance, while it is possible to combine temporal logics with description logics, satisfiability in such "temporal description logics" may be undecidable unless the expressiveness of the description logic or temporal logic is restricted (see [48] for details).

In this chapter, we discuss the approach to realizing and verifying component-based systems that we use in our framework for developing optimal component-based systems. This approach was initially developed as our "meta-framework" for Web service composition in [59] and later refined and generalized to any system that is developed using goal-oriented requirements engineering in [60] (and the related technical report [61]). The core assumption of this approach is that given a set of specification formalisms (where "formalism" is broadly defined) and their corresponding verification techniques, the overall functional requirement for the system is *decomposable* into individual sub-requirements (traits) that can each be expressed in at least one of these existing formalisms. The process is divided into the following steps:

1. Decompose the overall requirement for the system R into a Boolean combination of traits $\psi_1 \odot \psi_2 \odot \cdots \odot \psi_l$, where $\odot \in \{\wedge, \vee\}$ and each property ψ_i can be fully expressed in an existing formalism.

2. For each trait ψ_i, use existing search techniques to identify a collection of components $\{w_{i1}, w_{i2}, \ldots, w_{ik}\}$ that claim to satisfy ψ_i. Some of these components may be compositions of two or more smaller components.

3. Using these sets and the Boolean relationship between traits ψ_i, choose at least one set of components such that their composition is likely to satisfy the overall requirement R. At the same time, identify the specific traits against which this potential system must be verified in order to guarantee that it satisfies R (its *verification obligation*).

4. Attempt to compose the selected components to realize the proposed system and verify it against its verification obligation. Our work is *independent* of any particular composition operator, so any type of composition (e.g., parallel or sequential) may be used. If successful, the problem has been solved.

In essence, we reduce the problem of finding component-based systems that are likely to provide the desired functionality to identifying satisfiable assignments of an appropriate propositional logic formula (the SAT problem), which is generated by using the results obtained from applying existing composition algorithms to the individual traits (ψ_is) that constitute R. As an example, if R is decomposed into $\psi_1 \wedge \psi_2$, if $\{w_1, w_2\}$ and $\{w_3, w_4\}$ are sets of components that satisfy ψ_1 and ψ_2 respectively, and if any component that satisfies ψ_1 can be combined with any component that satisfies ψ_2, then the solution involves computing a satisfiable assignment to the propositional logic formula $(w_1 \vee w_2) \wedge (w_3 \vee w_4)$. The reduction is realized by using a simple AND-OR tree [54] that captures both the dependencies between traits and the set of services that satisfy each trait while preserving the separate semantics of different traits. Using the AND-OR tree, our meta-framework indicates exactly which traits must be satisfied by a given composition of components (step 3).

However, the reduction alone is not sufficient to guarantee the solution's correctness. A possible system (composition of components) must still be realized and verified against its verification obligation to ensure that it does satisfy the requirement R (step 4). Suppose that components w_1 and w_3 are incompatible with each other, and suppose further that w_4 does not satisfy ψ_2 when composed with either w_1 or w_2, even though w_4 satisfies ψ_2 in isolation. In this case, the only possible system that satisfies both ψ_1 and ψ_2 will be $w_2 \otimes w_3$.

Because we are interested in *which* components are being composed (and whether their composition is feasible) but not *how* the components are being composed, we model the composition process using a generic composition operator $\otimes$ that can represent any type of composition. For specific compositions, this generic operator may be freely replaced by the specific composition operator(s) under consideration (e.g., parallel or sequential). Our technique does not depend on and is not restricted to any specific types of formalisms used to express traits; in fact, new formalisms can be incorporated directly into our method.

Our approach was inspired by solution techniques such as those described in [4, 52, 94], which also involve decomposing a given functional requirement into a set of sub-requirements. However, these techniques are targeted toward component-based systems created from a library of components which satisfy certain assumptions. For more general component-based system development problems, particularly for Web service composition, two main factors work against these older techniques. First, there is in general no *a priori* guarantee that any two components can be composed unless they are specifically designed to work together, such as components developed as part of a generic library. Second, even when unrelated components such as Web services can be composed, the resulting composition may not have the same properties as its individual components. For instance, if a high-security component is composed with a low-security component, the additional security features of the high-security component may be compromised. An approach to component-based system development that relies on decomposing the problem and then combining partial solutions must address both of these factors.

Note. We will use a scenario centered on creating and verifying a Web service composition, taken from our previous work in [59], as a running example throughout this chapter. Much of the explanation given in this chapter is also based on [59]. Consequently, our meta-framework for requirement modeling and verification will be explained here in terms of Web service composition. Keep in mind that our meta-framework can easily be generalized to any type of component-based system, as we will see in Chapters 5 and 6.

4.1 Example: Medical Records Management

Consider a proposed medical record management system to be implemented using Web services. The primary purpose of the system is to provide remote access to a centralized store of medical records. In addition, a variety of security, privacy, and other regulatory requirements form an integral part of the functional requirement for the system. We will focus in this chapter on one specific high-level privacy requirement for this system:

> The system shall provide an appropriate level of access to a patient's medical record only to that patient or to persons who are directly involved in that patient's care.

This requirement is too general to immediately formalize in a meaningful way, so it is decomposed into the following two sub-requirements:

1. The system shall ensure that medical information is electronically exchanged between trusted hosts via secure communication.

2. The system shall provide an appropriate level of access to each patient's medical record to each authorized person based on that person's role in that patient's care.

Suppose each sub-requirement is further decomposed into a set of *atomic* properties (i.e., properties that cannot be decomposed further), such as the seven properties listed in Table 4.1. A (non-empty) subset of these atomic properties must be satisfied in order for each sub-requirement to be satisfied. Specifically, sub-requirement 1 is satisfied only if properties ψ_1, ψ_2,

Table 4.1 List of properties that contribute to satisfying privacy requirement

Property	*Meaning*
ψ_1	When a message is received from IP address i, a reply message shall be sent to i eventually.
ψ_2	The system shall periodically request each user's IP address to verify that it has not changed from its previous value, and terminate if it has.
ψ_3	All communications between system components shall be encrypted.
ψ_4	A patient may view, but not edit, his or her own medical information.
ψ_5	A doctor may view and edit medical information of his or her own patients.
ψ_6	A patient or doctor can access medical information from outside the hospital via automated telephone service.
ψ_7	A patient or doctor can access medical information from outside the hospital via online Web access.

and ψ_3 are all satisfied; on the other hand, sub-requirement 2 is fulfilled only if properties ψ_4 and ψ_5 are satisfied, plus at least one of ψ_6 or ψ_7 is satisfied. The overall functional requirement of the medical record management service can be viewed as:

$$\overbrace{(\psi_1 \wedge \psi_2 \wedge \psi_3)}^{\text{sub-requirement 1}} \quad \wedge \quad \overbrace{(\psi_4 \wedge \psi_5 \wedge (\psi_6 \vee \psi_7))}^{\text{sub-requirement 2}}$$

The traits ψ_i in this example can be naturally expressed in specific formalisms. For instance, ψ_1 and ψ_2 can be specified using a temporal logic such as CTL [21], ψ_3 can be specified using WS-SecurityPolicy [56] assertions, and ψ_4, ψ_5, ψ_6, and ψ_7 can be specified using a description logic [7]. Furthermore, it is generally difficult, if not impossible, to unify all property expressions in one formalism without loss of information or without making each property statement prohibitively complex.

We address this problem by providing a *meta-framework* that can encode the decomposition of the top-level requirement into a Boolean combination of sub-requirements or traits expressed in different formalisms (e.g., $\psi_1, \psi_2, \ldots, \psi_7$ as in Table 4.1). We use this meta-framework to identify sets of components that are likely to satisfy the overall requirement when composed by effectively combining the results provided by existing composition methods for each of the sub-requirements.

4.2 Problem Decomposition and Modeling

The service composition problem considers a user-specified functional requirement R that must be satisfied by composing one or more existing services available in a given service repository E. It is clear that this problem is an instance of the more general component-based system development problem. We claim that R can be *decomposed* into a set of individual properties or traits $\{\psi_1, \ldots, \psi_n\}$, such that (1) the Boolean combination of ψ_is is equivalent to R and (2) no single trait ψ_i can be further decomposed into multiple sub-traits. This forms the central theme of our meta-framework.

We begin by defining the notion of an atomic property in our context and then define the notion of decomposing a property into one or more atomic properties.

Definition 1 (Atomic Property) *A specification of an* atomic property *is a tuple $\psi \doteq \langle \varphi, F \rangle$, where φ is an expression that is interpreted according to a formalism F.*

For instance,

$$\begin{aligned} \psi_1 &= \langle \varphi_1, F_1 \rangle = \langle AF\ \texttt{sendAck}, CTL \rangle \\ \psi_2 &= \langle \varphi_2, F_2 \rangle = \langle \forall \texttt{caredBy(patient, X)}.\texttt{accessDataOf(X, patient)}, \mathcal{ALC} \rangle \end{aligned} \tag{4.1}$$

are examples of atomic property tuples. AF `sendAck` (*always, eventually* perform the `sendAck` operation) is a property expressed in the temporal logic formalism CTL (Computation Tree Logic [21]). Similarly, the second atomic property states that all patient caregivers have access to the medical records of the patients assigned to them; this property is expressed in the description logic $\mathcal{ALC}$ [7].

Observe that Definition 1 is nearly identical to the definition of a trait as given in Section 3.2, but without the verification method M. For simplicity, we omit any mention of a selected verification method M_i for a given atomic property ψ_i when we write atomic property tuples in this chapter.

Definition 2 (Composite Property) *A composite property θ is a Boolean composition of atomic properties, i.e., $\theta \doteq \psi_1 \odot \psi_2 \odot \cdots \odot \psi_n$ where $\forall i \in [1, n] : \psi_i$ is an atomic property and $\odot \in \{\wedge, \vee\}$.*

For instance, using Equation 4.1, $\langle\varphi_1, F_1\rangle \wedge \langle\varphi_2, F_2\rangle$ is a Boolean decomposition of a composite property that expresses requirements in two different formalisms: temporal logic and description logic.

The decomposition of a composite property can be represented using a structure known as an *AND-OR tree* [54], where the root node of the tree represents the complete functional property specification under consideration and each leaf node represents an atomic property. AND-OR trees are similar to *goal graphs* or *goal models*, which are used in the goal-oriented requirements engineering (GORE) methodology described in Section 2.2 to model, decompose, and reason about the satisfiability of goals for software systems; this connection will be explored further in the running example in Chapter 5. AND-OR trees are also similar to *feature diagrams*, which are used in software product line engineering to illustrate commonalities and variabilities between multiple products in a product line [79].

We now formally define the AND-OR tree in our context.

Definition 3 (AND-OR Tree) *An AND-OR tree of a composite property θ is denoted by $\mathcal{T}^\theta \doteq (Q, q_0, Q_\ell, \Delta, L)$, where Q is the set of nodes, $q_0 \in Q$ is the root node, $Q_\ell \subseteq Q$ is the set of leaf nodes, $\Delta : Q \to (\{\wedge, \vee\} \times \mathcal{P}(Q))$ is a transition function such that*

$$\Delta(q) = \begin{cases} \emptyset & \textit{if } q \in Q_\ell \\ \{op, next(q)\} & \textit{otherwise, where } next(q) \in \mathcal{P}(Q) \textit{ and } |next(q)| = 2 \end{cases}$$

and $L : Q \to sf(\theta)$ is a labeling function where

$$L(q) = \begin{cases} \langle\varphi, F\rangle & \textit{if } q \in Q_\ell \\ L(q_1)\ op\ L(q_2) & \textit{otherwise, where } next(q) = \{q_1, q_2\} \textit{ and } \Delta(q) = \{op, next(q)\} \end{cases}$$

In the above, $op \in \{\wedge, \vee\}$ and $sf(\theta)$, which denotes the set of subformulas of θ, is defined as:

$$sf(\theta) = \begin{cases} \{\theta\} & \textit{if } \theta \doteq \langle\varphi, F\rangle \\ sf(\theta_1)\ \cup\ sf(\theta_2)\ \cup\ \{\theta_1, \theta_2, \theta_1\ op\ \theta_2\} & \textit{if } \theta \doteq \theta_1\ op\ \theta_2 \end{cases}$$

Example. Figure 4.1 illustrates one possible realization of the AND-OR tree $\mathcal{T}^R$, where R denotes the overall property of the medical record service discussed in Section 4.1. The decomposition of R is performed according to Table 4.1. Note that there are multiple configurations

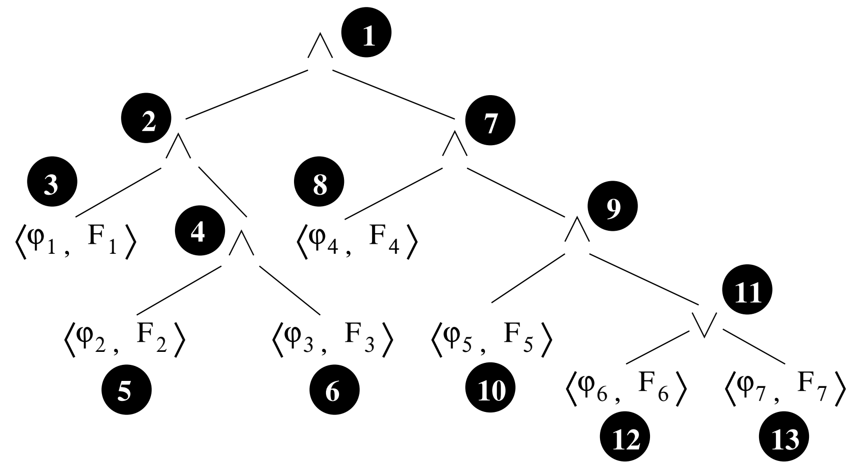


Figure 4.1 AND-OR tree showing decomposition of requirements in Table 4.1

of $\mathcal{T}^R$ that can be realized from the Boolean decomposition based on the order in which the atomic properties are considered.

4.3 Identification of Promising Component Sets

Given an AND-OR tree representation of the desired overall requirement R, we now show that much (but not all) of the Web service composition problem can be reduced to the AND-OR tree satisfiability problem, which is related to the Boolean satisfiability (SAT) problem. This involves the following steps:

1. *Reuse of existing composition techniques*: For each leaf-level property (atomic property) $\langle \varphi_i, F_i \rangle$ in the AND-OR tree $\mathcal{T}^R$, identify the set of services that satisfy $\langle \varphi_i, F_i \rangle$.

2. *Feasibility of candidate composite services*: Using the above set, identify and realize *feasible* service compositions, i.e., those that *may* satisfy the overall requirement R.

3. *Verifiability of feasible composite services*: From the set of feasible service compositions, identify those whose behavior *does* satisfy the requirement R.

Note that the final step goes beyond the reduction to AND-OR tree satisfiability, since this reduction alone is not sufficient to ensure that interactions between component services do not prevent the requirement R from being completely satisfied. Our meta-framework computes the verification obligation for each feasible composition that is identified. This verification obligation is essentially the set of leaf-level properties against which that composition must be verified to provide this assurance.

Because our meta-framework is *independent* of any particular type of composition, we continue to use a generic composition operator $\otimes$ to represent composition throughout this chapter. The choice of composition paradigm, e.g., parallel or sequential, makes no difference in our meta-framework until a composition is actually created; at that time, the generic composition operator may be freely replaced with the actual composition operator(s) to be used.

Identify Services that Satisfy Leaf-Level Sub-Requirements. Recall that the leaf-level sub-requirements are atomic properties of the form $\psi_i = \langle \varphi_i, F_i \rangle$, where φ_i is represented in the formalism F_i. Assume that there exists a method M_i and a corresponding algorithm that can handle properties expressed in F_i and can (semi-)automatically identify services (atomic and/or composite) that satisfy φ_i. In other words, M_i can identify one or more services (components) w such that w satisfies φ_i by the semantics of formalism F_i; we denote this by $w \models \langle \varphi_i, F_i \rangle$ or equivalently by $w \models_{F_i} \varphi_i$. We use these existing methods to identify the set of services that satisfy each atomic property at the leaf level of the AND-OR tree $\mathcal{T}^R$. Note that with the addition of a verification method M_i for each atomic property ψ_i, an atomic property becomes equivalent to a trait as defined in Section 3.2.

Example. Table 4.2 illustrates sets of services that satisfy some leaf-level atomic properties in the AND-OR tree $\mathcal{T}^R$ presented in Figure 4.1. Each set can be identified by using existing compositional methods that are capable of addressing the composition problem if the functional requirements are provided in one specific formalism. (Recall that $w_1 \otimes w_2$ denotes a composite service composed of services w_1 and w_2.)

Identify Feasible Compositions. Each node in the AND-OR tree $\mathcal{T}^R$ is the root of a subtree that corresponds to a subformula of R. For example, in Figure 4.1, node 11 is the root of $\mathcal{T}^{\theta_{11}}$, where $\theta_{11} \doteq \langle \varphi_6, F_6 \rangle \vee \langle \varphi_7, F_7 \rangle$. (For $\mathcal{T}^R$ in Figure 4.1, we will use θ_i to denote

Table 4.2 Sample sets of services that satisfy leaf-level properties in Figure 4.1

Services	*Atomic Property*	*Node #*
...	...	...
$\{w_4, w_5\}$	$\psi_5 \doteq \langle \varphi_5, F_5 \rangle$	10
$\{w_1 \otimes w_2, w_3\}$	$\psi_6 \doteq \langle \varphi_6, F_6 \rangle$	12
$\{w_3, w_4\}$	$\psi_7 \doteq \langle \varphi_7, F_7 \rangle$	13

the subformula of R that corresponds to the AND-OR subtree rooted at node number i, with θ_1 being R itself.) We identify the set of possible service compositions which, if they can be realized, are *likely* to satisfy each composite property θ_i. This objective is realized by using the semantics of satisfiability in propositional logic. For instance, θ_{11} can be satisfied by any service that satisfies either $\langle \varphi_6, F_6 \rangle$ or $\langle \varphi_7, F_7 \rangle$, i.e., by any one of the services $w_1 \otimes w_2$, w_3, or w_4 as shown in Table 4.2. Furthermore, the choice of service(s) provides information regarding the subformula(s) that must be satisfied, which we refer to as the *verification obligation*. In the current example, if the service $w_1 \otimes w_2$ is selected, then the atomic property $\langle \varphi_6, F_6 \rangle$ needs to be satisfied.

For each node i in $\mathcal{T}^R$, we identify a set of pairs (W_i, Ψ_i), where W_i denotes a set of services that are likely to satisfy θ_i when composed and Ψ_i is the set of formulas capturing the verification obligation on W_i. We now define a relation *mark* that is used to identify such a set of pairs for each node in the AND-OR tree.

Definition 4 *Given $\mathcal{T}^\theta$ and a repository of components E, we define a relation $\forall q \in Q$: $mark(q) \subseteq E \times sf(\theta)$ such that*

$$\begin{aligned}
&1.\quad [q \in Q_\ell \ \wedge\ \exists i : \textstyle\bigotimes_i w_i \models L(q)] \\
&\qquad\qquad \Rightarrow\ (\cup_i \{w_i\}, \{L(q)\}) \in mark(q) \\
&2.\quad [\Delta(q) = \{\wedge, \{q_1, q_2\}\} \wedge\ \exists (W_1, \Psi_1) \in mark(q_1) \wedge\ \exists (W_2, \Psi_2) \in mark(q_2)] \\
&\qquad\qquad \Rightarrow\ (W_1 \cup W_2, \Psi_1 \cup \Psi_2) \in mark(q) \\
&3.\quad [\Delta(q) = \{\vee, \{q_1, q_2\}\} \wedge\ \exists q' \in \{q_1, q_2\} : \exists (W, \Psi) \in mark(q')] \\
&\qquad\qquad \Rightarrow\ (W, \Psi) \in mark(q)
\end{aligned} \tag{4.2}$$

Example. Consider both Figure 4.1 and Table 4.2. From Rule 1 in Definition 4, we have

$$\{(\{w_4\},\{\psi_5\}),\quad (\{w_5\},\{\psi_5\})\} = \mathit{mark}(q_{10})$$

$$\{(\{w_1,w_2\},\{\psi_6\}),\quad (\{w_3\},\{\psi_6\})\} = \mathit{mark}(q_{12})$$

$$\{(\{w_3\},\{\psi_7\}),\quad (\{w_4\},\{\psi_7\})\} = \mathit{mark}(q_{13})$$

Therefore, using Rule 3 in Definition 4, we have

$$\left\{\begin{array}{ll} (\{w_1,w_2\},\{\psi_6\}), & (\{w_3\},\{\psi_6\}), \\ (\{w_3\},\{\psi_7\}), & (\{w_4\},\{\psi_7\}) \end{array}\right\} = \mathit{mark}(q_{11})$$

This means that property θ_{11} at node q_{11} is satisfiable if $w_1 \otimes w_2$ satisfies ψ_6, if w_3 satisfies either ψ_6 or ψ_7, or if w_4 satisfies ψ_7. Proceeding using Rule 2 in Definition 4 for node q_9, we have $\mathit{mark}(q_9)$ equal to the following set:

$$\left\{\begin{array}{l} (\{w_1,w_2,w_4\},\{\psi_6,\psi_5\}), \\ (\{w_1,w_2,w_5\},\{\psi_6,\psi_5\}), \\ (\{w_3,w_4\},\{\psi_6,\psi_5\}),\quad (\{w_3,w_5\},\{\psi_6,\psi_5\}), \\ (\{w_3,w_4\},\{\psi_7,\psi_5\}),\quad (\{w_3,w_5\},\{\psi_7,\psi_5\}), \\ (\{w_4\},\{\psi_7,\psi_5\}),\quad (\{w_4,w_5\},\{\psi_7,\psi_5\}) \end{array}\right\} = \mathit{mark}(q_9)$$

Theorem 1 (Feasibility) *For any AND-OR tree $\mathcal{T}^\theta$, if $w_1 \otimes \cdots \otimes w_k$ satisfies θ, then there exists a subset Ψ of the set of atomic formulas representing θ such that $(\{w_1,\ldots,w_k\},\Psi) \in \mathit{mark}(q_0)$, where q_0 is the root of $\mathcal{T}^\theta$.*

Proof. The proof proceeds inductively from the child nodes to the parent node of the AND-OR tree $\mathcal{T}^\theta$ using the definition of the *mark* relation (Definition 4).

For the leaf nodes, the theorem is trivially true by Rule 1 in Definition 4.

Any intermediate node (including the root node) of $\mathcal{T}^\theta$ has the form $\wedge$ or $\vee$. Assume the theorem holds for any node at depth $d{+}1$. We prove that the theorem holds for any intermediate node q_i^d at depth d. Suppose q_i^d is an $\wedge$-node; then its "marking" follows Rule 2 in Definition 4. It takes one element from the marking of each child node (at depth $d+1$) and generates the set of services W and the set of verification obligations Ψ. The set W contains all services required to satisfy each conjunct of θ_i^d (i.e., each θ_j^{d+1}). Further, all possible combinations of elements from the marking of each child node are considered. Therefore, the theorem holds for any $\wedge$-node at depth d.

Suppose q_i^d is an $\vee$-node. Then the "marking" follows Rule 3 of Definition 4, i.e., the union of the markings of the child nodes is equal to the marking of q_i^d. In other words, the marking of q_i^d includes any set of services that is sufficient to satisfy any one of the disjuncts in θ_i^d (i.e., any θ_j^{d+1}). $\square$

Corollary 1 *For any node q in any AND-OR tree $\mathcal{T}^\theta$, if $w_1 \otimes w_2 \otimes \ldots \otimes w_k$ satisfies $L(q)$, then there exists a subset Ψ_q of the set of atomic formulas used to represent $L(q)$ such that $(\{w_1, w_2, \ldots, w_k\}, \Psi_q) \in mark(q)$.*

4.4 System Realization and Verification

Theorem 1 states a necessary condition for generating a composite service from a set of component services: if there exists a set of services in a tuple belonging to $mark(q_0)$, the composition of those services (if it can be formed) is *likely* to satisfy θ. We refer to the set of services in each tuple of $mark(q_0)$ as the *feasible* composition set. However, this condition is not sufficient, as the converse of Theorem 1 does not hold. As a counterexample, consider the case where $mark(q_0) = \{(\{w_1, w_2\}, \{\psi_1, \psi_2\})\}$ and any one of the following is true:

- w_1 is not compatible with w_2
- $w_1 \otimes w_2 \not\models \psi_1$
- $w_1 \otimes w_2 \not\models \psi_2$

In these cases, the composition $w_1 \otimes w_2$ either does not exist or does not satisfy θ, as it does not satisfy either ψ_1 or ψ_2. The first condition can be detected by attempting to construct the composition, while the other conditions can be detected by verification against the resulting composition.

Therefore, we attempt to realize (i.e., compose or construct) each feasible composite service by using an appropriate existing composition algorithm. Any composition that cannot be realized using an available composition algorithm is removed from the set of feasible compositions. The remaining feasible compositions proceed to the verifiability-checking step.

Recall that for any tuple in *mark*$(\cdot)$ for any node in $\mathcal{T}^R$, a set of atomic properties is identified as the *verification obligation* for that tuple. In essence, the verification obligation must be satisfied by the composite service to ensure satisfaction of the property at the node under consideration. This is formally stated in the following theorem.

Theorem 2 (Verifiability) *For any* $\mathcal{T}^\theta$*, the composition* $w_1 \otimes w_2 \otimes \ldots \otimes w_k$ *satisfies* θ *if and only if there exists a subset* Ψ *of the set of atomic formulas used to represent* θ *such that* $(\{w_1, w_2, \ldots, w_k\}, \Psi) \in mark(q_0)$ *and, for all* $\psi \in \Psi$*,* $w_1 \otimes w_2 \otimes \ldots \otimes w_k \models \psi$*.*

Proof. The proof proceeds inductively, as for Theorem 1. □

Note that properties at the leaf level are never considered together at any step, so verifiability of each leaf-level property ψ_j can be checked using the existing methods M_j that deal with verifying properties specified using the formalism F_j.

Corollary 2 *For any node* q *of any AND-OR tree* $\mathcal{T}^\theta$*,* $w_1 \otimes w_2 \otimes \ldots \otimes w_k$ *satisfies* $L(q)$ *if and only if there exists a subset* Ψ *of the set of atomic formulas used to represent* $L(q)$ *such that* $(\{w_1, w_2, \ldots, w_k\}, \Psi) \in mark(q)$ *and, for all* $\psi \in \Psi$*,* $w_1 \otimes w_2 \otimes \ldots \otimes w_k \models \psi$*.*

On-the-fly Checking for Verifiability. Based on Theorem 2, given a property R, we can obtain a composite service that satisfies R by (a) constructing $\mathcal{T}^R$, (b) identifying the set of services that satisfy the atomic properties in R, (c) "marking" each node in $\mathcal{T}^R$, and finally (d) checking whether there exists a tuple $(\{w_1, w_2, \ldots, w_k\}, \Psi)$ in *mark*(q_0) where $w_1 \otimes w_2 \otimes \cdots \otimes w_k$ is realizable and $\forall \psi \in \Psi : w_1 \otimes w_2 \otimes \ldots \otimes w_k \models \psi$. In the worst case, this process will require

$$\sum_{(W,\Psi)\in mark(q_0)} |mark(q_0)| \times |\Psi|$$

verifications, where $|\cdot|$ denotes the size of a relation or set.

We now propose an alternative approach, which is likely to be beneficial if all of the following are true:

- a majority of the feasible composite services are not verifiable
- a majority of the feasible composite services that are not verifiable can be detected near the leaf level of the AND-OR tree
- the user wants to develop multiple composite services with significant overlap in functional requirements

The essence of our alternative approach is to combine the feasibility checking process (i.e., *mark*$(\cdot)$ computation) with the verification process. This is realized by computing the *verif* relation, defined as follows, for each node in the AND-OR tree.

Definition 5 *Given an AND-OR tree $\mathcal{T}^\theta$ and a repository of components E, we define a relation $\forall q \in Q : \mathit{verif}(q) \subseteq E \times \mathit{sf}(\theta)$ such that*

$$
\begin{aligned}
&1.\quad [q \in Q_\ell \ \wedge\ \exists i : \textstyle\bigotimes_i w_i \models L(q)] \\
&\qquad\qquad \Rightarrow (\cup_i\{w_i\}, \{L(q)\}) \in \mathit{verif}(q) \\
&2.\quad \left[\begin{array}{l} \Delta(q) = \{\wedge, \{q_1, q_2\}\} \wedge\ \exists (W_1, \Psi_1) \in \mathit{verif}(q_1) \wedge\ \exists (W_2, \Psi_2) \in \mathit{verif}(q_2) \\ \wedge\ \boxed{\forall \psi \in \Psi_1 \cup \Psi_2 : \textstyle\bigotimes_i w_i \models \psi\ \wedge\ \cup_i\{w_i\} = W_1 \cup W_2} \end{array}\right] \\
&\qquad\qquad \Rightarrow (W_1 \cup W_2, \Psi_1 \cup \Psi_2) \in \mathit{verif}(q) \\
&3.\quad [\Delta(q) = \{\vee, \{q_1, q_2\}\} \wedge\ \exists q' \in \{q_1, q_2\} : \exists (W, \Psi) \in \mathit{verif}(q')] \\
&\qquad\qquad \Rightarrow (W, \Psi) \in \mathit{verif}(q)
\end{aligned}
\tag{4.3}
$$

The critical difference between the *verif* and *mark* relations is emphasized in a box in Definition 5. Note that in computing the *verif* relation for the intermediate nodes, we are incrementally creating the specified composition(s) at the same time; in the previous approach, the compositions are realized only after the *mark* relation is computed.

Example. Going back to our AND-OR tree $\mathcal{T}^R$ in Figure 4.1 and the sets of services in Table 4.2, suppose $w_3 \otimes w_4 \not\models \psi_5$, $w_4 \otimes w_5 \not\models \psi_7$, and w_1 is incompatible with w_4. If we had first computed the *mark* relation and then checked the verification obligations at the root node, we would have followed the previous example and obtained *mark*(q_9), which contains eight elements. Those eight elements would have been combined with elements of the marking from q_9's sibling nodes to compute the *mark* relation for the parent. Instead, if the *verif* relation is computed according to Definition 5, *verif*(q_9) will be

$$\left\{ \begin{array}{c} (\{w_1, w_2, w_5\}, \{\psi_6, \psi_5\}),\ (\{w_3, w_5\}, \{\psi_6, \psi_5\}), \\ (\{w_3, w_5\}, \{\psi_7, \psi_5\}),\ (\{w_4\}, \{\psi_7, \psi_5\}) \end{array} \right\}$$

In short, using on-the-fly verification, it is possible to remove four intermediate compositions that will not satisfy the overall functional property R due to incompatibility between participating services or unsatisfiability of atomic properties. This reduction can result in a considerable gain in computation efficiency if the requirement under consideration has many atomic properties (i.e., if the size of the AND-OR tree is large) and if there are multiple services (with possible incompatibilities) that satisfy each of the atomic properties.

Additionally, if the stakeholders want to develop multiple composite services with common requirements, identifying the set of verifiable services for the common requirements will lead to effective reuse of intermediate results. Consider Figure 4.2, which illustrates an AND-OR graph unifying AND-OR trees $\mathcal{T}^R$ and $\mathcal{T}^{R'}$ that share multiple atomic properties. Suppose that a composition satisfying R has been identified using the *verif* relation for $\mathcal{T}^R$ and further suppose that, for each node $q_i \in \mathcal{T}^R$, the value of *verif*(q_i) was saved for later use. When the user wants to find a composition that satisfies R' using on-the-fly verification, the previously computed values of *verif* for the shared nodes can be reused; the *verif* relation needs to be computed for only the three nodes in $\mathcal{T}^{R'}$ that are not shared with $\mathcal{T}^R$.

The following theorem formally states the correctness of our method.

Theorem 3 (Soundness and Completeness) *For any $\mathcal{T}^R$, $w_1 \otimes w_2 \otimes \ldots \otimes w_k$ satisfies R if and only if $(\{w_1, w_2, \ldots, w_k\}, \Psi) \in$ verif(q_0), where q_0 is the root of $\mathcal{T}^R$ and Ψ is a subset of the set of* atomic *formulas used to represent R.*

Proof. The proof is realized via induction over the depth of the AND-OR tree $\mathcal{T}^R$. □

The complexity of our methods (based on Theorems 2 and 3) depends directly on the size of $\mathcal{T}^R$, i.e., the number of choices of services that satisfy each atomic property. Let the number of atomic properties representing R be N and the maximum number of choices of

$$\begin{aligned} R: \quad & (\langle \varphi_1, F_1 \rangle \wedge \langle \varphi_2, F_2 \rangle \wedge \langle \varphi_3, F_3 \rangle) \\ & \wedge (\langle \varphi_4, F_4 \rangle \wedge \langle \varphi_5, F_5 \rangle \wedge (\langle \varphi_6, F_6 \rangle \vee \langle \varphi_7, F_7 \rangle)) \\ R': \quad & (\langle \varphi_3, F_3 \rangle \wedge \langle \varphi_8, F_8 \rangle) \\ & \wedge (\langle \varphi_4, F_4 \rangle \wedge \langle \varphi_5, F_5 \rangle \wedge (\langle \varphi_6, F_6 \rangle \vee \langle \varphi_7, F_7 \rangle)) \end{aligned}$$

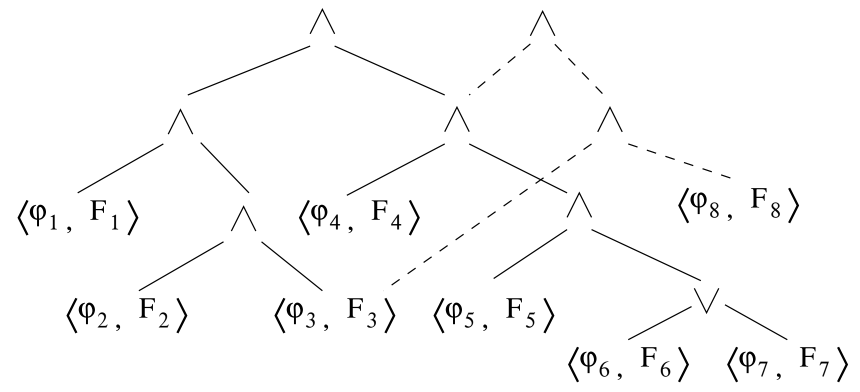


Figure 4.2 AND-OR graph showing decomposition of requirements for two related systems, rooted at nodes labeled $\mathcal{T}^R$ and $\mathcal{T}^{R'}$ respectively. Dotted lines indicate parts of the graph that belong solely to $\mathcal{T}^{R'}$.

services satisfying an atomic property be K. The *mark* relation is essentially identifying a satisfiable assignment set (in terms of atomic properties) for the Boolean representation of R and associating services that satisfy each element in the satisfiable assignment set. The size of the satisfiable assignment set for any Boolean formula with N propositions (atomic properties) is 2^N. Therefore, there are at most $KN2^N$ elements in $mark(q_0)$ or $verif(q_0)$ at the root of $\mathcal{T}^R$. Note that this is not a tight upper bound, particularly for the *verif* relation. The cost for computing the *verif* relation depends on both (a) the reduction in results due to unsatisfiability of verification obligations or incompatibility of selected services at intermediate nodes and (b) the increased computation needed to perform verification at each intermediate node.

CHAPTER 5. MODELING, REFINING, AND APPLYING QUALITATIVE PREFERENCES OF STAKEHOLDERS

In the previous chapter, we showed how a correct solution to a component-based system development problem, i.e., a system that satisfies the overall requirement given by its stakeholders, can be obtained. However, for any given requirement, there may exist many possible system designs (i.e., sets of components) that satisfy the requirement. The goal of the system designers should be to construct and deliver the best such possible system, but to do so requires a well-defined notion of what is meant by the "best" or "optimal" system. This must be defined uniquely for each system by the preferences and priorities of the system's stakeholders with respect to various combinations of optional traits that a given system may provide beyond the basic requirement.

Our strategy for achieving this goal within our component-based system development framework is to order all correct component-based system designs, as previously identified using the meta-framework from Chapter 4, from most to least preferred according to the stakeholders' preferences. This ordering can be accomplished in two different ways:

1. We could determine which sets of components can actually be composed into correct systems that solve the given problem, then order these correct systems with respect to the stakeholders' preferences. To do so, we must be able to decide whether one system is preferred to (*dominates*) another.

2. Alternatively, we could first place all system designs (sets of components) that are likely to solve the problem correctly in order from most to least preferred. Given this ordering, we can attempt to realize and verify the most preferred system design. If verification of this design fails, we can try again with the second-most preferred design, the third-

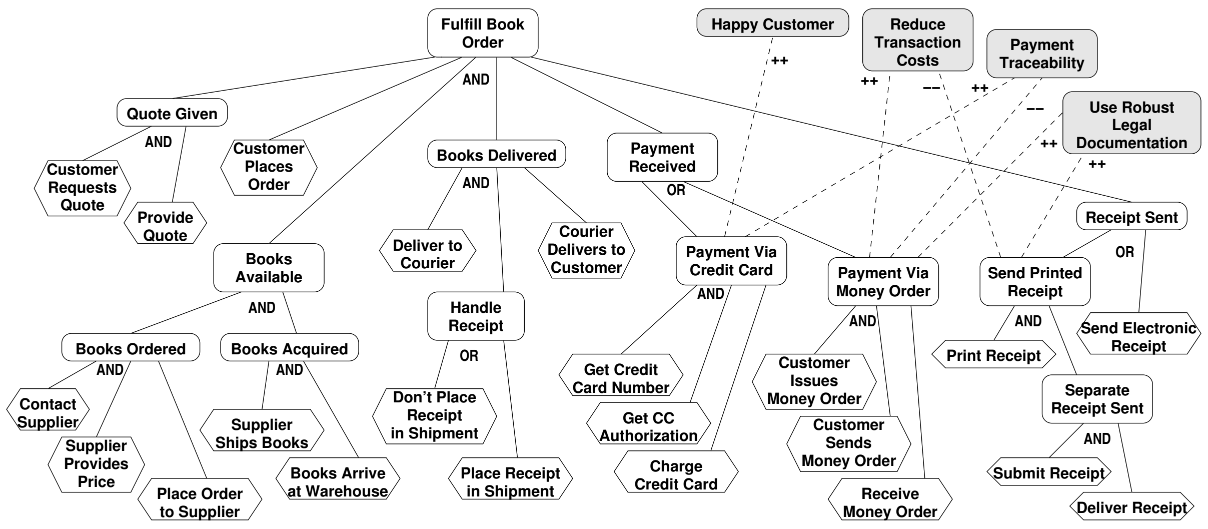


Figure 5.1 Online book sales goal model, taken from [44] (via [60])

most preferred design, and so on until an optimal correct system is realized or until all possibilities are exhausted. To do so, we need to have a correct method for ordering all possible solutions in descending order of preference.

This chapter describes the methods that we use to model, refine, and apply the qualitative preferences of a proposed system's stakeholders within the component-based system design process. The methods that we present in this chapter were introduced in our previous work on preference reasoning with goal-oriented requirements engineering (GORE) in [60, 61] and explained in more detail in the context of the credential disclosure minimization problem in [63]. We present these preference handling methods using an example set of preferences over optional goals within a goal model as used in GORE (see Section 2.2 for more details). However, the methods presented in this chapter are independent of any particular requirements engineering methodology and are broadly applicable in many settings.

5.1 Example: Preferences between Optional Goals of a System

We will use a modified version of the online book sales goal model introduced by Liaskos et al. in [44] and adapted for our work in [60, 61] as a running example throughout this chapter. This goal model is shown in Figure 5.1, which is taken from [60]. An online book selling system developed according to this goal model must fulfill several core goals, such as ensuring that the ordered books are available, providing a price quote for each order, and receiving payments. Each of these high-level goals can be decomposed into one or more smaller, well-defined tasks, which may each be satisfied by a single component or combination of components. For instance, in order for the book selling system to obtain books to sell, it must contact suppliers, receive price quotes from those suppliers, and then choose which books to order from each supplier as well as how many copies of each book to order. This is clearly an instance of AND-decomposition as described in the previous chapter. Furthermore, some core goals or subgoals can be satisfied in multiple ways, such as the different payment options commonly offered to customers. Such a selection of one or more sufficient alternatives for satisfying a goal is an example of OR-decomposition as defined previously.

Observe that the main part of the goal model in Figure 5.1 (all except for the shaded ovals and dashed lines) is an instance of an AND-OR tree as described in Section 4.2. The root node (*Fulfill Book Order*) describes the overall requirement for the system, which is AND-decomposed into four subgoals (internal AND or OR nodes) and one task (leaf node or trait). Tasks always appear as leaf nodes in the tree formed by the goal model, because they represent basic operations that the system may perform in partial fulfillment of the overall requirement. The mapping to our meta-framework for problem decomposition in Chapter 4 is clear: each task can be specified as a trait of the system as defined in Section 3.2, with different semantics and verification techniques used to verify that the system performs each task, while the goal model specifies the Boolean combinations of tasks that are sufficient to satisfy the overall requirement. The primary difference from the meta-framework in Chapter 4 is that every node (goal or task) in the goal model is given a name. In contrast, only the leaf nodes (traits) in the AND-OR tree in Section 4.2 are given high-level descriptions beyond a logical operator.

In addition to the main goal model, there are also certain optional goals (non-functional properties) that the bookseller considers relevant to the success of the system. These include reduced transaction costs, customer satisfaction, payment traceability, and use of robust documentation for legal purposes. Unfortunately, it may not be possible to satisfy all of these optional goals or traits while still providing all of the core functions required by the system.

For example, requiring payment by money order instead of credit card would reduce transaction costs to the business, but it would also reduce customer satisfaction by making payment less convenient. Realizing this, the bookseller has identified several acceptable tradeoffs between optional goals:

P1. If robust documentation is used, payment traceability is more important than reducing transaction costs.

P2. If transaction costs are reduced at the expense of customer satisfaction, then using robust documentation takes precedence over ensuring payment traceability.

P3. If robust documentation is not provided, then it should be possible to trace payments in order to satisfy regulatory auditing requirements, even at the expense of reduced customer satisfaction and increased transaction cost.

Although the goal model in Figure 5.1 can be analyzed by itself to determine the set of all likely correct designs for the system, information about the stakeholders' preferences over the optional goals or traits is also needed to identify the *best* or *most preferred* design, i.e., the set of components that can be combined in a way that provides the required functionality of the system while fulfilling the most preferred subset of the optional traits. Although it would be ideal from the stakeholders' perspective if a single system could fulfill the entire set of optional goals, this may not be feasible because of the possible existence of optional goals that receive both positive and negative contributions from two or more goals or tasks that are part of the same system design.

In Figure 5.1, if a design includes the goals *Payment via Money Order* and *Send Printed Receipt*, then that design cannot fulfill the optional goal *Reduce Transaction Costs*, since the first goal positively contributes to this optional goal while the second goal contributes negatively to it. Also note that this design will not be able to fulfill the optional goals *Reduce Transaction Costs* and *Payment Traceability* because of the satisfaction of the goal *Payment via Money Order*, which positively contributes to one optional goal and negatively contributes to the other.

In light of such conflicts, a user will typically ascertain preferences over the set of optional goals with the objective of obtaining a functionally correct design that is most preferred. A system design can be considered *correct* if the goal at the root of the goal tree is satisfied. The preference of the design, on the other hand, depends directly on the set of optional goals fulfilled by the design. Therefore, a *most preferred correct* system design is one that fulfills a set of optional goals γ such that there exists no other correct design that fulfills another set of optional goals γ', where γ' is preferred to γ.

5.2 Formalizing Preferences as a CI-Net

To capture and reason with preferences over the set of optional goals, we use *conditional importance networks* (CI-nets) [12], which allow stakeholders to specify relative importance among sets of items. A CI-net $\mathcal{C}$ is a set of conditional importance statements of the form:

$$S^+, S^- : S_1 \succ S_2$$

where S^+, S^-, S_1 and S_2 are pairwise disjoint subsets of the set of all optional goals G^O. Informally, such a statement is read as: “Given two designs, if both fulfill the optional goals in S^+ and do not fulfill the optional goals in S^-, then the design that fulfills all optional goals in S_1 is preferred to the design that fulfills all optional goals in S_2.” For instance, preference P2 described in Section 5.1 states, “If transaction costs are reduced at the expense of customer satisfaction, then using robust legal documentation takes precedence over ensuring payment traceability.” This preference is expressed in the CI-net language as follows:

$$\{\textit{Reduced Transaction Cost}\}, \{\textit{Happy Customer}\} :$$
$$\{\textit{Use Robust Legal Documentation}\} \succ \{\textit{Payment Traceability}\}$$

Formally, a CI-net $\mathcal{C}$ over a set of optional goals G^O is *satisfiable* if and only if there exists a strict partial order (irreflexive and transitive) relation $\succ$ over the powerset of G^O such that:

1. For each CI-net statement $S^+, S^- : S_1 \succ S_2$, if $\gamma \subseteq G^O \setminus (S^+ \cup S^- \cup S_1 \cup S_2)$ then $\gamma \cup S^+ \cup S_1 \succ \gamma \cup S^+ \cup S_2$.

2. $\succ$ is monotonic, i.e., $\gamma \supset \gamma' \Rightarrow \gamma \succ \gamma'$.

Based on this definition, the CI-net preference language satisfies the criteria for a dominance relation for our component-based system development framework as defined in Section 3.4.

Revisiting the preference statement above, by the rule in item 1, the set of optional goals {*Reduced Transaction Cost, Use Robust Legal Documentation*} is preferred to ($\succ$) the set of optional goals {*Reduced Transaction Cost, Payment Traceability*}. By the rule in item 2, a set of optional goals is preferred to all of its proper subsets.

CI-nets are a natural choice for modeling user preferences over sets of optional goals in goal models and sets of optional traits that component-based systems may provide for the following reasons:

1. Preferences in CI-nets are *monotonic.* According to the semantics of CI-nets, a set of optional goals is preferred to all its proper subsets. This property is necessary to model preferences over optional goals because the stakeholders would typically like to see as many optional goals or traits fulfilled as possible; ideally, all of them would be fulfilled.

2. The CI-net semantics induces a *strict partial order* among the subsets of the optional goal set with respect to the given conditional importance statements. Thus, it is possible to order the subsets of optional goals in a way that is consistent with the semantics of a CI-net. Such an ordering can be used to search for designs that fulfill more preferred optional goals ahead of those that fulfill less preferred optional goals.

3. When it is not possible to find a design that satisfies all of the optional goals, stakeholders are often willing to trade certain sets of optional goals for others, rather than only choosing between individual optional goals. Unlike many other preference modeling languages, these *tradeoffs* can be directly represented in CI-nets using conditional importance statements over sets of optional goals.

5.3 Consistency Checking and Dominance Testing in CI-Nets

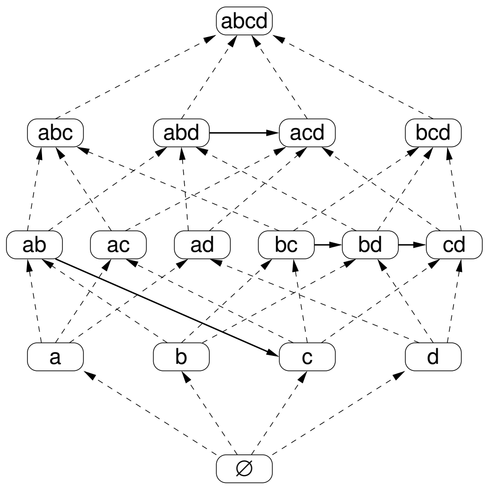
Given two choices (of sets of optional goals), deciding the preference of one choice over the other is referred to as *dominance testing.* Although dominance testing is known to be

PSPACE-complete [12, 29], Santhanam et al. demonstrated in [75] an effective model checking-based approach to dominance testing for certain families of preferences, such as TCP-nets [14]. In this section, we follow a similar approach for dominance testing between choices (of sets of optional goals) where preferences are represented using CI-nets. This approach relies on an alternative semantics of CI-nets given in terms of an *improving flipping sequence*, analogous to the worsening flipping sequence defined in [12].

Definition 6 (Improving Flipping Sequence [12]) *A sequence of sets of optional goals* $\gamma_1, \gamma_2, \cdots \gamma_{n-1}, \gamma_n$ *is an* improving flipping sequence *with respect to a set of CI-net statements if and only if, for* $1 \leq i < n$*, either*

1. (*Monotonicity Flip*) $\gamma_{i+1} \supset \gamma_i$*; or*
2. (*Importance Flip*) *there exists a conditional importance statement* $S^+, S^- : S_1 \succ S_2$ *in the CI-net, such that all three of the following conditions are satisfied:*
   (a) $\gamma_{i+1} \supseteq S^+$*,* $\gamma_i \supseteq S^+$*, and* $\gamma_{i+1} \cap S^- = \gamma_i \cap S^- = \emptyset$*;*
   (b) $\gamma_{i+1} \supseteq S_1$*,* $\gamma_i \supseteq S_2$*, and* $\gamma_{i+1} \cap S_2 = \gamma_i \cap S_1 = \emptyset$*;*
   (c) *if* $\gamma = G^O \setminus (S^+ \cup S^- \cup S_1 \cup S_2)$*, then* $\gamma \cap S_1 = \gamma \cap S_2$*.*

In this definition, condition 1 states that fulfilling additional optional goals is always preferred to fulfilling fewer optional goals. Condition 2 states that if the optional goals in set S^+ are fulfilled and the optional goals in set S^- are not fulfilled, then fulfilling the set S_1 of optional goals is more important than fulfilling the set S_2 of optional goals, all others being equal (the *ceteris paribus* condition discussed in Section 2.3, which is ensured by condition 2(c)). Given a CI-net $\mathcal{C}$ and two sets γ and γ' of optional goals, we say that γ is preferred to γ' under $\mathcal{C}$, denoted by $\mathcal{C} \models \gamma \succ \gamma'$, if and only if there is an improving flipping sequence with respect to $\mathcal{C}$ from γ' to γ (Proposition 1 in [12]). In our example CI-net (see Section 5.1), we can thus say that the set {*Reduced Transaction Cost*, *Use Robust Legal Documentation*} is preferred to the set {*Payment Traceability*}. This is because the set {*Payment Traceability*} has an improving (importance) flip to the set {*Reduced Transaction Cost*, *Payment Traceability*},



a = *Happy Customer*
b = *Reduce Transaction Costs*
c = *Payment Traceability*
d = *Use Robust Legal Documentation*

CI-net statements:

P1. $\{d\},\{\}\ :\ \{c\} \succ \{b\}$
P2. $\{b\},\{a\}\ :\ \{d\} \succ \{c\}$
P3. $\{\},\{d\}\ :\ \{c\} \succ \{a, b\}$

Figure 5.2 Induced preference graph for CI-net in online book sales example

which in turn has an improving (monotonicity) flip to {*Reduced Transaction Cost*, *Use Robust Legal Documentation*}.

From this definition, one can construct a graph where each node corresponds to a set of optional goals and each directed edge from one node to another denotes an "improving flip", capturing the fact that the set of optional goals at the destination node is preferred to the set of optional goals at the source node. This graph is referred to as the *induced preference graph.*

Definition 7 (Induced Preference Graph) *Given a CI-net $\mathcal{C}$ over a set of optional goals G^O, the* induced preference graph $\delta(\mathcal{C}) = (N, E)$ *is constructed as follows. The nodes N correspond to the powerset of G^O, and each directed edge $(\gamma, \gamma') \in E$ corresponds to an improving (monotonicity or importance) flip from γ to γ' as per the CI-net semantics (Definition 6) such that $\gamma' \succ \gamma$.*

Figure 5.2 presents the CI-net statements corresponding to the preferences over optional goals specified in Section 5.1, along with the induced preference graph constructed from those statements. The dashed edges between sets of optional goals in this graph correspond to monotonicity flips and the solid edges correspond to importance flips. Every path in the graph represents an improving flipping sequence induced by the CI-net.

A set γ' of optional goals *dominates* (i.e., is preferred to) another set γ with respect to CI-net $\mathcal{C}$ if and only if the node corresponding to γ' is reachable from the node corresponding to γ in the induced preference graph $\delta(\mathcal{C})$. For example, the set {*Reduced Transaction Cost*, *Use Robust Legal Documentation*} is preferred to the set {*Payment Traceability*} due to the existence of the path $c \rightarrow bc \rightarrow bd$ in Figure 5.2.

The induced preference graph of a CI-net is *consistent*, meaning that no set of traits is preferred to itself, if and only if it is cycle-free. Before using the preferences contained in a CI-net to make decisions about possible system designs, it is necessary to first ensure that the given set of preferences is consistent.

5.3.1 Kripke Structure Modeling of CI-Net Semantics

We use the NuSMV [20] or Cadence SMV [50] symbolic model checker to verify reachability (and therefore dominance) from one node to another in the induced preference graph. There are three primary advantages in using one of these model checking tools for testing dominance. First, they are equipped with symbolic or binary decision diagram-based algorithms that allow for efficient state-space exploration of large graphs. Second, they can verify properties beyond simple reachability in expressive temporal logic (e.g., CTL and LTL), a capability that we will use in Section 5.3.3 to obtain a preference ordering over sets of optional goals. Finally, the SMV input language, which is accepted by both NuSMV and Cadence SMV, allows us to directly encode the CI-net preference statements. The induced preference graph is then automatically constructed by the model checker to answer dominance (verification) queries.

The model checker takes as input a Kripke structure $\langle S, S_0, T, L\rangle$, where S is the set of states, $S_0 \subseteq S$ is the set of start states, $T \subseteq S \times S$ is the set of transition relations, and L is a labeling function mapping each state in S to a set of propositions that hold at that state. In our encoding, we represent each optional goal as a proposition x_i, where the value of the proposition is true when the optional goal is fulfilled and false when the optional goal is not fulfilled. The propositions are uninitialized, which allows the model checker to consider all possible valuations of the propositions as initial states of the Kripke structure. Given a set of CI-net statements $\mathcal{C}$, the Kripke structure $K_{\mathcal{C}}$ representing the induced preference graph $\delta(\mathcal{C})$ contains states that are labeled with the truth values of the set of optional-goal propositions x_i along with two types of helper Boolean variables: a set of variables h_i and a single variable g.

SMV Input Language: Role of Helper Variables. A Kripke structure is encoded in SMV using a set of variables, each variable's possible initial valuations, and a set of transition relations. Each transition relation describes the valuation of one variable based on certain conditions of the current state-variable valuations. For instance, consider the following partial SMV encoding of a Kripke structure with two Boolean variables a and b. (Note that in all SMV code used in this chapter, the value 0 represents "false" and the value 1 represents "true".)

```
                              next(a) := case
                                             a = b : !a;
init(a) := 0;
                                             1     : a;
                                         esac;
```

This SMV specification states that the initial valuation of a is 0 (false), while the initial valuation of b can be either 0 or 1 since it is not explicitly given. The corresponding Kripke structure has two different start states: one where a and b are equal to 0 (false) and another where a is equal to 0 (false) and b is equal to 1 (true). Furthermore, the transition relation (described by the `next` operation) states that the value of a is toggled only when the valuations of a and b are equal in the current state. The absence of `next` definitions for b indicates that the valuation of b can change non-deterministically whenever a change in state occurs in the Kripke structure.

In the encoding of $\delta(\mathcal{C})$ as a Kripke structure $K_{\mathcal{C}}$, attributes over which the CI-net statements are specified are encoded as Boolean variables in $K_{\mathcal{C}}$. Each state in $K_{\mathcal{C}}$ corresponds to a node in $\delta(\mathcal{C})$: if $x_3 \wedge x_4$ holds (evaluates to true) in a state in $K_{\mathcal{C}}$, that state corresponds to the node annotated with x_3 and x_4 in $\delta(\mathcal{C})$. Next, note that the existence of a given edge in $\delta(\mathcal{C})$ depends on the contents of the source and destination nodes (improving flip, see Definition 6). Direct encoding of such edges in SMV requires encoding of transitions in $K_{\mathcal{C}}$ where the `next` operation on each variable, which describes the enabling condition of the transitions, includes conditions that depend on the variables' values in the next states. Encoding such conditions in SMV may lead to circular dependencies between `next` operations for two or more variables. As an example, consider the following SMV code:

```
next(a) :=  case                          next(b) :=  case
              next(b) : !a;                             next(a) : !b;
              1       : a;                              1       : b;
            esac;                                     esac;
```

From the above encoding, it is not clear what valuation a and b should have in the next state when the current state valuations of the variables are equal to 1.

Role of h_i. To correctly encode the edges of $\delta(\mathcal{C})$ as transitions in $K_{\mathcal{C}}$, we use one auxiliary Boolean variable h_i for each proposition x_i. Each h_i is encoded such that if h_i is 0 (false) in the current state, then in the next state the valuation of x_i cannot change; otherwise, the valuation of x_i may change in the next state if a condition matching a CI-net statement is satisfied. All h_i variables are initialized to 0, and the model checker updates the h_is non-deterministically. For instance, the semantics of the CI-net statement $\{d\}, \{\} : \{c\} \succ \{b\}$ (preference P1 from Section 5.1, resulting in edges $bd \rightarrow cd$ and $abd \rightarrow acd$ in $\delta(\mathcal{C})$) can be encoded in SMV as:

```
next(b) := case
              h_a = 0           -- a does not change in next state
           &  b = 1 & h_b = 1   -- b can change in the next state
           &  c = 0 & h_c = 1   -- c can change in the next state
           &  d = 1 & h_d = 0   -- d does not change in next state
                  : 0
           ...
           esac;
next(c) := case
              h_a = 0           -- a does not change in next state
           &  b = 1 & h_b = 1   -- b can change in the next state
           &  c = 0 & h_c = 1   -- c can change in the next state
           &  d = 1 & h_d = 0   -- d does not change in next state
                  : 1
```

```
    ...
    esac;
```

The enabling conditions are identical in both cases to ensure that the valuations of b and c are updated under identical conditions as specified by the CI-net, namely when $d = 1$ in both the current and next states (ensured by $h_d = 0$ in the current state) and when the valuation of a is unaltered in both the current and next states (ensured by $h_a = 0$ in the current state). Further, $c = 0$ and $h_c = 1$ in the current state, which allows the value of c to change in the next state; similarly, $b = 1$ and $h_b = 1$ in the current state, which allows for the toggling of b in the next state.

In this way, the semantics of CI-nets as given in Definition 6 can be directly encoded as SMV specifications. This encoding eliminates the need to manually construct the induced preference graph $\delta(\mathcal{C})$. Instead, the model checker automatically constructs and explores the Kripke-structure model representing $\delta(\mathcal{C})$.

Role of g. Within this encoding, the different valuations of each h_i for the same valuation of each x_i correspond to states in $K_\mathcal{C}$ that allow different ways in which the valuation of that x_i can be changed. Consequently, $K_\mathcal{C}$ contains multiple states where an identical set of x_is hold true; all of these states correspond to one node in $\delta(\mathcal{C})$. Transitions between these states do not change the valuation of any x_i and, therefore, do not correspond to any edge in $\delta(\mathcal{C})$.

The variable g is set to 1 (true) whenever a transition traversed in $K_\mathcal{C}$ results in a change in the valuation of at least one of the x_is (i.e., when a transition in $K_\mathcal{C}$ corresponds to an edge in $\delta(\mathcal{C})$). Conversely, if a transition in $K_\mathcal{C}$ does not indicate a change in any of the x_i variables, the variable g is set to 0 (false). Consider the following SMV code, which updates g based on the CI-net statements that encode the preferences expressed in Section 5.1:

```
next(g) := case
        -- Guards corresponding to P1, where g will be set to 1 :
                      h_a = 0 -- a does not change in next state
            & b = 1 & h_b = 1 -- b can change in the next state
            & c = 0 & h_c = 1 -- c can change in the next state
            & d = 1 & h_d = 0 -- d does not change in next state
                : 1 -- g is set to 1 indicating that this transition
                    -- corresponds to a change in "b" or "c"
        ...
        -- Guards corresponding to P2, where g will be set to 1 :
        ...
        -- Guards corresponding to P3, where g will be set to 1 :
        ...


        1: 0 -- default case : if no variables change, then g is 0
    esac;
```

Note that these are precisely the same conditions under which b changes to 0 (false) and c changes to 1 (true), as defined in the previous SMV code excerpt. The code in this excerpt sets g to 1 whenever the conditions for changing the value of b and c are satisfied. The full `next(g)` block contains conditions for setting g to 1 when any monotonicity or importance flip causes one or more variables to change; we have omitted the remaining conditions for clarity of explanation. The `1` condition at the end of the block sets g to 0 if no other condition is met, i.e., if no variables change during the specified transition. In Section 5.3.3, we show how the variable g can be used directly to compute the ordering of preferred solutions.

Figure 5.3 shows how the data variables x_i, the helper variables h_i, and the change variable g interact within the Kripke structure $K_{\mathcal{C}}$ for a node in the induced preference graph $\delta(\mathcal{C})$ containing variables a and b. The most preferred node in $\delta(\mathcal{C})$ is the set of all elements, while the least preferred node is the empty set; nodes containing only a or only b are intermediate nodes. Each node in $\delta(\mathcal{C})$ is modeled by a set of interconnected states in $K_{\mathcal{C}}$.

We have expanded one node of $\delta(\mathcal{C})$, where $a = 1$ and $b = 0$, in Figure 5.3 to fully show the corresponding set of states in $K_{\mathcal{C}}$. The expanded node is divided into two subsets of states: the left subset a_g represents the set of states where $g = 1$, while the right subset $a_{\neg g}$ represents the set of states where $g = 0$. There are four states in both subsets, one for each possible valuation of the two Boolean variables h_a and h_b. Any state in a_g can be reached immediately from some state in $K_{\mathcal{C}}$ that represents the node where $a = 0$ and $b = 0$ in $\delta(\mathcal{C})$. States where $h_a = 0$ and $h_b = 1$ move to states in $K_{\mathcal{C}}$ where $a = 1$ and $b = 1$, regardless of g's value. All other states in a_g can move to some state in $a_{\neg g}$ by a transition in $K_{\mathcal{C}}$; however, since $g = 0$ in all states in $a_{\neg g}$, any transition to or between the states in $a_{\neg g}$ does not correspond to any edge in $\delta(\mathcal{C})$. Note that, as in a_g, the state in $a_{\neg g}$ where $h_a = 0$ and $h_b = 1$ has transitions to states where $a = 1$ and $b = 1$. The rest of the Kripke structure $K_{\mathcal{C}}$ is constructed similarly: each node in $\delta(\mathcal{C})$ corresponds to a set of states in $K_{\mathcal{C}}$, where the number of states in the set is exponential in the number of variables (optional goals) in $\delta(\mathcal{C})$.

Theorem 4 *Given a CI-net $\mathcal{C}$, a Kripke structure $K_{\mathcal{C}}$ constructed as described in this subsection preserves the semantics of the induced preference graph $\delta(\mathcal{C})$ of the CI-net.*

Proof. Consider the induced preference graph $\delta(\mathcal{C})$ for CI-net $\mathcal{C}$ as defined in Definition 7. Each state in $K_{\mathcal{C}}$ maps onto exactly one node in $\delta(\mathcal{C})$. Furthermore, given two nodes $\gamma, \gamma' \in \delta(\mathcal{C})$ and two states $s, s' \in K_{\mathcal{C}}$ where s maps to γ and s' maps to γ', there exists a directed edge $(\gamma, \gamma') \in \delta(\mathcal{C})$ if and only if both (1) there exists a transition $s \to s' \in K_{\mathcal{C}}$ and (2) $g = 1$ in state s'. This transition $s \to s'$ models the improving flip (γ, γ') in the induced preference graph. $\square$

5.3.2 Model Checking for Verifying Consistency and Dominance

Given a CI-net $\mathcal{C}$, we use the method in Section 5.3.1 to specify the corresponding Kripke model $K_{\mathcal{C}}$ for input to the Cadence SMV or NuSMV model checker. We begin by verifying that the induced preference graph $\delta(\mathcal{C})$ modeled by $K_{\mathcal{C}}$ is consistent (i.e., cycle-free). This is done by checking $K_{\mathcal{C}}$ against the LTL formula $F\ G(g = 0)$, which is satisfied if and only if every path from the initial state in $K_{\mathcal{C}}$ eventually reaches a point where no x_i variable ever changes (i.e., g is always 0) in any future state.[1] If a cycle exists in the induced preference graph, then every state in the cycle always has at least one outgoing transition from that state where $g = 1$, indicating that a variable is changing; this violates the consistency property.

[1]Details of LTL syntax and semantics can be obtained in [71].

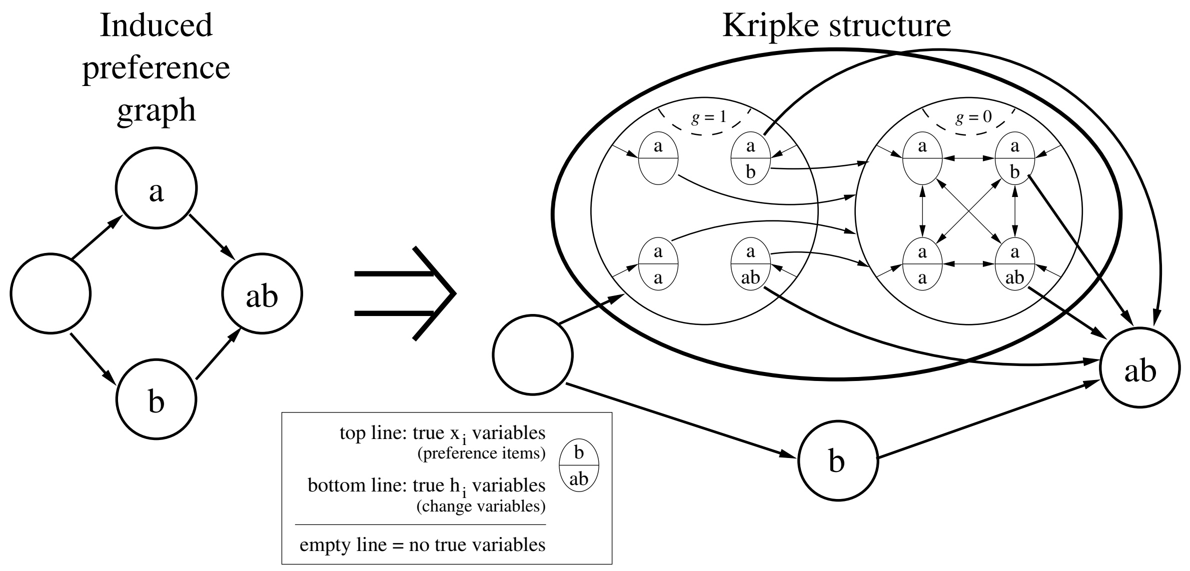

Induced preference graph
a
ab
b
Kripke structure
g = 1
g = 0
a
a b
a a
a ab
ab
b
top line: true x_i variables (preference items)
bottom line: true h_i variables (change variables)
empty line = no true variables
b
ab

Figure 5.3 Kripke structure encoding of part of an induced preference graph

After the model $K_{\mathcal{C}}$ is verified to be consistent, it can be used for preference reasoning. For any sets of optional goals γ and γ', we use the CTL formula $X \Rightarrow EF(X')$, where X (resp. X') is the propositional formula indicating the presence or absence of optional goals in γ (resp. γ'), to check whether γ' is preferred to γ. This property is satisfied by any state in $K_{\mathcal{C}}$ where X holds true and where there is a path leading to a state where X' holds true.[2] If the property is satisfied, we conclude that γ' is preferred to γ. An improving flipping sequence from γ to γ' can be obtained by querying the model checker with the negation of the formula $X \Rightarrow EF(X')$; the counterexample returned by the model checker is a path in the Kripke structure that proves dominance, which can be used to construct the improving flipping sequence. On the other hand, if the property $X \Rightarrow EF(X')$ is not satisfied, then there is no improving flipping sequence from γ to γ', i.e., γ' is *not* preferred to γ. In the CI-net used in our example (see Section 5.1), the model checker returns true when queried with the formula $(bd \Rightarrow EF(acd))$, which verifies that acd is preferred to or dominates bd. When we query the model checker with the CTL formula $\neg(bd \Rightarrow EF(acd))$, it yields a counterexample corresponding to either the path $bd \rightarrow cd \rightarrow acd$ or the path $bd \rightarrow abd \rightarrow acd$. Either path proves the dominance of acd over bd.

[2]Details of CTL syntax and semantics can be obtained in [22].

We find the most preferred set of optional goals by verifying the CTL property $EF(g = 1)$ for all states in $K_{\mathcal{C}}$. This property is satisfied at a state s in $K_{\mathcal{C}}$ if and only if s can reach any state (including itself) where g evaluates to 1 (true). The property is not satisfied at states in $K_{\mathcal{C}}$ that correspond to the top-most node (containing the set of all optional goals) of the induced preference graph $\delta(\mathcal{C})$. This is because the top-most node in $\delta(\mathcal{C})$ does not contain any outgoing edges. Any one of the states in $K_{\mathcal{C}}$ that corresponds to the top-most node in $\delta(\mathcal{C})$ is identified by NuSMV or Cadence SMV as a counterexample, proving the unsatisfiability of the property $EF(g = 1)$. In our running example, this query returns the state where variables a, b, c, and d are true, which corresponds to the set of all optional goals. This reflects the fact that fulfilling all given optional goals is the most preferred alternative.

5.3.3 Preference Ordering over Sets of Optional Traits

Once we have modeled the induced preference graph $\delta(\mathcal{C})$ as a Kripke-structure model $K_{\mathcal{C}}$ and confirmed using the NuSMV or Cadence SMV model checker that the modeled preferences are consistent, our next objective is to obtain an ordering of sets of optional goals from most preferred to least preferred. Note that $\delta(\mathcal{C})$ represents a strict partial order between sets of optional goals. The ordering we obtain using this technique is a total order consistent with this strict partial order. We achieve this by performing model checking on the model $K_{\mathcal{C}}$ and its modifications against CTL properties. The steps in our approach are as follows.

1. We verify all states in $K_{\mathcal{C}}$ against the CTL property $EF(g = 1)$. As noted previously, this returns the most preferred set of optional goals from the top of $\delta(\mathcal{C})$. In general, as $\delta(\mathcal{C})$ is a strict partial order, it may have multiple elements at the top. Any state that corresponds to any one of the top elements will be returned as the counterexample, which proves the unsatisfiability of the CTL property.

2. Let $\gamma_1, \gamma_2, \ldots, \gamma_n$ be the sequence of sets of optional goals that has been obtained so far (as the total order consistent with the partial order presented in $\delta(\mathcal{C})$). We define the following formula:

$$I = \bigvee_{i=1}^{n} \bigwedge_{j} (x_{ij}) \tag{5.1}$$

where X_{ij} is the proposition representing the presence or absence of the jth optional goal in the set γ_i. We then query the model checker with the modified CTL property $EF(g = 1) \ \vee \ I$. The property is satisfied by state s in $K_{\mathcal{C}}$ if and only if (a) all states s' reachable from s can, in turn, reach some state where $g = 1$ is true or (b) s corresponds to nodes $\gamma_1, \gamma_2, \ldots, \gamma_n$ in $\delta(\mathcal{C})$. On the other hand, if the property is not satisfied by s, then s cannot reach a state where g is set to true and s does not correspond to nodes $\gamma_1, \gamma_2, \ldots, \gamma_n$. In other words, if the property is satisfied, there exists no state in $K_{\mathcal{C}}$ corresponding to a set of optional goals that is at least as preferred as at least one element in $\gamma_1, \gamma_2, \ldots, \gamma_n$.

3. If the model checker returns false, then it identifies (as a counterexample) a state corresponding to a set of optional goals γ_{n+1}, which is at least as preferred as one of the previously identified sets of optional goals $\gamma_1, \gamma_2, \ldots, \gamma_n$. In this case, we iterate Step 2 using the new sequence $\gamma_1, \gamma_2, \ldots, \gamma_n, \gamma_{n+1}$. Otherwise, the property is satisfied by all states in $K_{\mathcal{C}}$, meaning there exists no set of optional goals that is at least as preferred as one of the elements in $\gamma_1, \gamma_2, \ldots, \gamma_n$. If this occurs, we remove from the Kripke structure $K_{\mathcal{C}}$ all states corresponding to the optional-goal sets $\gamma_1, \gamma_2, \ldots, \gamma_n$ (obtained by iterating Step 2 so far) by adding $\neg I$ (see Equation 5.1) to the Kripke structure as an invariant, since the model checker only considers the states where the invariant holds. Thus, the reduced model corresponds to the induced preference graph where the nodes corresponding to $\gamma_1, \gamma_2, \ldots, \gamma_n$ are not considered. We then iterate starting from Step 1 until the invariant produces a model where no states are considered by the model checker.

Note that in Step 2, the states in the model corresponding to $\gamma_1 \ldots \gamma_n$ are *ignored* by the model checker (although they remain present in the model), as our query is modified to consider all states except these. This enables us to obtain the top-most nodes one by one in sequence without altering the model. However, when all of the top-most nodes are obtained, we *remove* the states corresponding to $\gamma_1 \ldots \gamma_n$ from the model in Step 3 by adding $\neg I$ as an invariant to the current model. This makes it possible to obtain the next set of top-most nodes in the subsequent iteration. We explain this process using the example $\delta(\mathcal{C})$ presented in Figure 5.2.

Iteration 1. Initially, the Kripke structure $K_{\mathcal{C}}$ encoding of $\delta(\mathcal{C})$ is verified against the property $EF(g = 1)$ following Step 1 above. The result (counterexample) obtained is the top-most element $\gamma_{11} = (abcd)$. In Step 2, model checking is performed again with the property $EF(g = 1) \ \vee \ I$, where $I = (a \wedge b \wedge c \wedge d)$ represents the fulfillment of all four optional goals. The property is satisfied because all states except the ones corresponding to $(abcd)$ can reach a state where $g = 1$ (true). Therefore, as per Step 3, we remove from $K_{\mathcal{C}}$ the states corresponding to the node $\gamma_{11} = (abcd)$ by adding $\neg I = (\neg a \vee \neg b \vee \neg c \vee \neg d)$ as an invariant to $K_{\mathcal{C}}$. As a result, we have forced the model checker to consider only the states where the invariant holds; at any state corresponding to the node $(abcd)$, the invariant does not hold. This can be viewed as an updated $K_{\mathcal{C}}$, which encodes a $\delta(\mathcal{C})$ where the nodes $\{(abc), (acd), (bcd)\}$ are at the top (as in Figure 5.2, but with the $(abcd)$ node and its incoming edges removed).

Iteration 2. Step 1 is performed again and the model checker returns as a counterexample one of the states that corresponds to either (abc), (acd), or (bcd). Note that one of these states is identified non-deterministically by the model checking algorithm. Suppose that the state corresponding to (abc) is obtained as a counterexample. So far, we have $\gamma_{11} = (abcd)$ (obtained in the previous iteration) followed by $\gamma_{21} = (abc)$ in our total ordering of sets of optional goals. Proceeding to Step 2, we have a new $I = (a \wedge b \wedge c \wedge d) \vee (a \wedge b \wedge c)$. Note that the states corresponding to $(abcd)$ have already been removed by adding the invariant $\neg a \vee \neg b \vee \neg c \vee \neg d$ before the start of Step 1 in the current iteration. As a result, we do not need to consider the first disjunct in the new I. When model checking is performed again, one of the states corresponding to either (acd) or (bcd) is obtained as a counterexample. Suppose that a state corresponding to $\gamma_{22} = (acd)$ is returned as a counterexample.

We proceed to perform Step 2 again with $I = (a \wedge b \wedge c) \ \vee \ (a \wedge c \wedge d)$. The model checker returns a counterexample state corresponding to the node $\gamma_{23} = (bcd)$. Proceeding further, Step 2 is again performed using $I = (a \wedge b \wedge c) \ \vee \ (a \wedge c \wedge d) \ \vee \ (b \wedge c \wedge d)$. At this point, the model checker fails to find any counterexamples for the property $EF(g = 1) \vee \ I$. In Step 3, we remove all of the states corresponding to the nodes $(abc), (acd)$ and (bcd) by adding the invariant $\neg I = (\neg a \vee \neg b \vee \neg c) \ \wedge \ (\neg a \vee \neg c \vee \neg d) \ \wedge \ (\neg b \vee \neg c \vee \neg d)$ to the model, and we start a new iteration from Step 1. So far, we have obtained an ordering of sets of optional goals $\gamma_{11} = (abcd), \gamma_{21} = (abc), \gamma_{22} = (acd), \gamma_{23} = (bcd)$.

Remainder of the Process. The iterative process, starting from Step 1, is illustrated in Table 5.1. The iteration is continued until $\neg I$ results in a $K_{\mathcal{C}}$ where no states are considered by the model checker. The total number of iterations is equal to the height of the partial order in $\delta(\mathcal{C})$. In this example, using $\delta(\mathcal{C})$ as shown in Figure 5.2, it is equal to 10. Each such iteration obtains a sequence of sets of optional goals that are equally preferred (or indistinguishable as per the given preferences). For instance, in iteration 2, we obtained (abc), (acd), and (bcd), which are equally preferred. Such elements are obtained by iterating Step 2 multiple times, with a new value of I each time. The maximum number of iterations starting at Step 2 is equal to the width of the partial order in $\delta(\mathcal{C})$. In this example (Figure 5.2), it is equal to 3.

The main advantage of using the method presented in this section is that a total ordering of sets of optional goals is obtained without performing all possible pairwise comparisons. Instead, systematic updates to the model corresponding to the induced preference graph and repeated model checking using a CTL property are used to automatically find the total order of sets of optional goals.

5.4 Determining the Most Preferred Alternatives

We now have effective methods for finding the most preferred set of optional goals, as well as for computing a sequence of optional goals that forms a total ordering over the powerset of optional goals and is consistent with the stated CI-net preferences. At this point, there are two options for identifying the most preferred system design alternatives (i.e., sets of tasks) from the set of all feasible system designs in the goal model (i.e., those that appear to be sufficient to satisfy the root goal):

1. Verify all feasible system designs to determine which of them can form actual systems that fulfill the root goal of the goal model, then determine which correct systems are most preferred based on the set of optional goals satisfied by each correct system.

Table 5.1 Steps to find ordering of optional goal sets for online book sales example

#	*Iteration*	*Query*	*Result*	*Action*
1.	Iteration 1	$\texttt{EF}(g = 1)$	$[abcd]$	$I = (abcd)$
2.		$\texttt{EF}(g = 1) \vee I$	–	Revise model by adding $\neg I$ as invariant
3.	Iteration 2	$\texttt{EF}(g = 1)$	$[abc]$	$I = (abc\bar{d})$
4.		$\texttt{EF}(g = 1) \vee I$	$[acd]$	$I = (abc\bar{d}) \ \vee \ (a\bar{b}cd)$
5.		$\texttt{EF}(g = 1) \vee I$	$[bcd]$	$I = (abc\bar{d}) \ \vee \ (a\bar{b}cd) \ \vee \ (\bar{a}bcd)$
6.		$\texttt{EF}(g = 1) \vee I$	–	Revise model by adding $\neg I$ as invariant
7.	Iteration 3	$\texttt{EF}(g = 1)$	$[abd]$	$I = (ab\bar{c}d)$
8.		$\texttt{EF}(g = 1) \vee I$	–	Revise model by adding $\neg I$ as invariant
9.	Iteration 4	$\texttt{EF}(g = 1)$	$[ac]$	$I = (a\bar{b}c\bar{d})$
10.		$\texttt{EF}(g = 1) \vee I$	$[ad]$	$I = (a\bar{b}c\bar{d}) \ \vee \ (a\bar{b}\bar{c}d)$
11.		$\texttt{EF}(g = 1) \vee I$	$[cd]$	$I = (a\bar{b}c\bar{d}) \ \vee \ (a\bar{b}\bar{c}d) \ \vee \ (\bar{a}\bar{b}cd)$
12.		$\texttt{EF}(g = 1) \vee I$	–	Revise model by adding $\neg I$ as invariant
13.	Iteration 5	$\texttt{EF}(g = 1)$	$[bd]$	$I = (\bar{a}b\bar{c}d)$
14.		$\texttt{EF}(g = 1) \vee I$	–	Revise model by adding $\neg I$ as invariant
15.	Iteration 6	$\texttt{EF}(g = 1)$	$[bc]$	$I = (\bar{a}bc\bar{d})$
16.		$\texttt{EF}(g = 1) \vee I$	–	Revise model by adding $\neg I$ as invariant
17.	Iteration 7	$\texttt{EF}(g = 1)$	$[c]$	$I = (\bar{a}\bar{b}c\bar{d})$
18.		$\texttt{EF}(g = 1) \vee I$	–	Revise model by adding $\neg I$ as invariant
19.	Iteration 8	$\texttt{EF}(g = 1)$	$[ab]$	$I = (ab\bar{c}\bar{d})$
20.		$\texttt{EF}(g = 1) \vee I$	–	Revise model by adding $\neg I$ as invariant
21.	Iteration 9	$\texttt{EF}(g = 1)$	$[a]$	$I = (a\bar{b}\bar{c}\bar{d})$
22.		$\texttt{EF}(g = 1) \vee I$	$[b]$	$I = (a\bar{b}\bar{c}\bar{d}) \ \vee \ (\bar{a}b\bar{c}\bar{d})$
23.		$\texttt{EF}(g = 1) \vee I$	$[d]$	$I = (a\bar{b}\bar{c}\bar{d}) \ \vee \ (\bar{a}b\bar{c}\bar{d}) \ \vee \ (\bar{a}\bar{b}\bar{c}d)$
24.		$\texttt{EF}(g = 1) \vee I$	–	Revise model by adding $\neg I$ as invariant
25.	Iteration 10	$\texttt{EF}(g = 1)$	[ ]	$I = (\bar{a}\bar{b}\bar{c}\bar{d})$
26.		$\texttt{EF}(g = 1) \vee I$	–	Revise model by adding $\neg I$ as invariant
27.		$\texttt{EF}(g = 1)$	–	No more states to explore. Terminate.

2. Order all feasible system designs in descending order of preference based on the optional goals they claim to satisfy, then attempt to realize the most preferred system design and verify that the realized system satisfies its verification obligation regarding both the root goal and any optional goals it is committed to fulfill. If the most preferred design cannot be verified, repeat with the second-most preferred design, then the third-most preferred design, and so on until a verified correct system is found or all designs are exhausted.

Both options produce the same results in the end, but one option may be more efficient than the other depending on the number of feasible system designs and the complexity of the goal model. If there are few feasible system designs to be realized and verified, or if the goal model is simple enough that verification can be completed relatively efficiently, then it may be better to determine which of the designs can be realized and verified as correct systems before considering preferences; perhaps only one of the systems will be correct, in which case preferences will not matter. On the other hand, if there are many feasible system designs or if the goal model is fairly complex, then it may be desirable to place the feasible system designs in descending order of preference, then realize and verify only the most preferred system designs needed to obtain an optimal correct system.

Before we proceed to find the most preferred satisfiable set of goals (i.e., the most preferred system design), we define some additional concepts. Let 2^S denote the powerset of set S.

1. Let R be a formula representing the Boolean combination of the goals in the goal model, where each proposition R_i in the formula indicates the corresponding goal $g_i \in G^R$ (recall that G^R is the set of required goals in the goal model). Then a *satisfiable goal assignment* is any set of goals $\hat{G}^R \subseteq G^R$ such that setting the corresponding propositions to true satisfies R. Let $\mathit{Sat}(R) = \mathit{Sat}(G^R)$ denote the set of all satisfiable goal assignments in 2^{G^R}. For instance, *Sat*(*Books Delivered*) is a set containing two different solutions: one solution contains the goal *Don't Place Receipt in Shipment* and the other contains the goal *Place Receipt in Shipment.* This is because either one of these goals must be satisfied in order to satisfy the goal *Handle Receipt*, satisfaction of which is necessary to satisfy the root goal *Books Delivered.*

2. Let $\gamma \subseteq G^O$ be a set of optional goals. A *contributing goal set*, denoted by *Contrib*(γ), is a set of goals (equivalently, a set of truth assignments to goal propositions in R) that, taken together, support (contribute positively to) every optional goal in the set γ. In other words, for every optional goal $s_j \in \gamma$, both of the following hold:

   (a) At least one goal in the set supports (has a ++ link to) s_j.

   (b) No goal in the set denies (has a -- link to) s_j.

   For instance, in Figure 2.1, *Contrib*(*Use Robust Legal Documentation*) = {*Payment Via Money Order*, *Send Printed Receipt*}.

With the help of these two concepts and the method described in Section 5.3.3 for computing a total preference ordering over the powerset of optional goals using CI-nets, we present our algorithm to find the most preferred functionally satisfactory assignment of goals in the goal tree. The aim is to identify a satisfiable goal assignment for the root goal of the goal model such that the assignment also contributes to the most preferred subset of optional goals G^O. In order to achieve this, we iterate through the possible sets of optional goals in 2^{G^O} in descending order of preference with respect to the given CI-net specification, using the method described in Section 5.3.3. The algorithm proceeds as follows:

1. Let R be the Boolean formula described above which, when satisfied, results in the satisfaction of the root goal in the goal model. Let $\gamma_1, \gamma_2, \ldots, \gamma_n$ be the total order sequence of sets of optional goals from most preferred to least preferred that was obtained by the method in Section 5.3.3.

2. For each i from 1 to n, for each $x \in \mathit{Sat}(R)$, and for each $y \in \mathit{Contrib}(\gamma_i)$: If $y \subseteq x$, then return x.

3. If the loop terminates, then no satisfiable assignment was identified for any set of optional goals (including the empty set); in this case, inform the user and return nothing.

In the above procedure, Step (2) is iterated until either (a) a satisfiable goal assignment that contributes to all of the optional goals in some γ_i (considered in descending order of preference) is found or (b) no one satisfiable goal assignment contributes to all of the optional goals in any γ_i. Note that in the latter case, indeed there is no satisfiable goal assignment because the procedure considers *all* possible subsets of optional goals one by one, including the set $\gamma_n = \emptyset$ where none of the optional goals are fulfilled (the least preferred option). In other words, the above procedure finds a satisfiable goal assignment if one exists. Further, the procedure considers sets of optional goals in descending order of preference, so it will find a satisfiable goal assignment (if one exists) that fulfills a more preferred set γ_i of optional goals ahead of other assignments that fulfill the set γ_j $(j > i)$, where γ_i is preferred to γ_j.

This establishes the correctness of our procedure and its optimality in finding the most preferred satisfiable goal assignment — in other words, the most preferred design.

Theorem 5 (Soundness and Completeness) *If our algorithm returns a satisfiable assignment to R, then there is no satisfiable goal assignment that contributes to a more preferred set of optional goals than the assignment returned by our algorithm. Furthermore, if our algorithm does not return a satisfiable assignment to R, then no satisfiable assignment to R exists.*

The proof of Theorem 5 follows directly from the steps of our method described above.

5.5 Implementation and Preliminary Results

We have developed a tool in Java to implement our approach for finding the most preferred goal assignments for a given goal model. The tool's architecture comprises two main components: a *Goal Model Analyzer* component that serves as the tool's front-end and a *Preference Reasoner* component that provides the tool's back-end functionality.

The Preference Reasoner uses either the NuSMV or Cadence SMV model checker to compute the next-most-preferred optional goal set at each step of the goal model analysis. This component reads a text file containing a CI-net specification and automatically generates the Kripke structure in the SMV language. The functionality of the Preference Reasoner is provided by two modules, which each have two sub-modules:

1. A *pre-processor* module that uses two sub-modules to produce input to the model checker, namely:
   (a) *Parser:* Reads CI-net statements specified in a text input file.
   (b) *Translator:* Automatically translates CI-net statements to generate the SMV input model.
2. A *reasoning driver* module that coordinates preference reasoning. Its two sub-modules invoke the model checker to do different tasks:
   (a) *Consistency Checker:* Checks the consistency of CI-nets, returning true if and only if the CI-net is consistent.
   (b) *Rank Order Generator:* Takes the model generated by the pre-processor, generates appropriate temporal logic (CTL or LTL) properties, and invokes the model checker.

After the first run of the model checker, it reads the output of the model checker, appropriately updates the property or refines the model by including invariants, and repeatedly invokes the model checker until all ordered results are obtained.

The Goal Model Analyzer constructs the goal model, including the AND-OR tree of goals and tasks, the optional goals, and the contribution links, from a text input file. Once finished with this task, the Goal Model Analyzer executes the algorithm given in Section 5.4, obtaining preference data from the Preference Reasoner. One important aspect of the Goal Model Analyzer and its interaction with the Preference Reasoner is that the latter communicates the best (most preferred) set of optional goals to the former and then waits until it is instructed to compute the next best set of optional goals. In other words, the Preference Reasoner does not compute the entire total order sequence of sets of optional goals; instead, it computes one and sends it to the Goal Model Analyzer. If the Goal Model Analyzer cannot find a satisfiable assignment that contributes to the set of optional goals, it communicates with the Preference Reasoner to obtain the next set of optional goals in the sequence of the total order. The Goal Model Analyzer eventually returns either the most preferred satisfiable goal assignment(s) or a message stating that no satisfiable goal assignment could be found.

The Preference Reasoner and Goal Model Analyzer are loosely coupled, allowing different approaches for preference reasoning or goal model analysis to be substituted into the existing tool with minimal effort. This approach fits naturally into the context of our overall component-based system development framework, as we will show in the next chapter. In addition, the data structures for the goal model are designed for extensibility, which will simplify the process of adding support for additional concepts to the Goal Model Analyzer in the future.

We have tested our implementation of our goal-model analysis framework against modified versions of three goal models from the existing literature on goal-oriented requirements engineering. These goal models describe requirements for an online book selling service [44] (Figure 2.1), a generalized online shopping system [43], and a public transport system [80]. We have also used a CI-net specifying new sets of preferences for each goal model that, in our opinion, are reasonable for each goal model's application domain.

Table 5.2 Results of running preference analysis tool on three case studies

Goal Model	*Required Goals*	*Tasks*	*Optional Goals*	*CI-net Rules*	*Mean Total Run Time (s)*	*Calls to Preference Reasoner*	*Mean Pref. Reasoning Time (s)*
Bookseller [44]	13	22	4	3	0.52	3	0.47
Trentino Transport [80]	24	40	3	3	0.47	2	0.34
Online Shop [43]	7	16	3	2	0.22	1	0.17

Table 5.2 summarizes the results obtained by running our analysis tool on each goal model. The tests were executed on a Gateway laptop with 4 GB of RAM and an Intel Core 2 Duo T5550 dual-core CPU (1.83 GHz), running 64-bit Windows Vista Service Pack 2. The times shown in Table 5.2 represent the mean of the running times reported for 20 runs of our tool over each model under the same configuration. It is clear that the preference reasoner's CI-net analysis accounts for the bulk of the running time. The time required for preference reasoning appears to depend primarily on the number of calls to the preference reasoner, although differences in the number of optional goals and CI-net preference rules also have an effect. However, the goal-model analyzer uses very little additional running time: about 0.05 seconds for the two smaller models and about 0.14 seconds for the much larger transport system model. While we plan to perform additional experiments with larger goal models and more complex preferences to further quantify the effects of goal model and CI-net size on running time, these preliminary results show that our goal model and preference analysis framework is as efficient as other comparable goal model analysis techniques such as [28] and [80], even though it supports more expressive preference specifications than those techniques.

CHAPTER 6. A NEW FRAMEWORK FOR DEVELOPING OPTIMAL COMPONENT-BASED SYSTEMS

In this chapter, we introduce our *modular*, *generic*, *end-to-end*, and *semi-automatic* framework for the development of optimal (i.e., correct and strongly preferred) component-based systems. Let us clarify what is meant by these four properties of our framework:

- *Modular:* The framework provides a common infrastructure to connect a number of *modules*, each of which provides a portion of the functionality necessary to identify, compose, and verify a component-based system in the application domain under consideration. Each large-scale step of the workflow can be customized for a given application domain by incorporating appropriate modules into the system to provide support for the formal specification, verification, and preference reasoning techniques used in that domain. Our framework is designed to incorporate a standard module specification that will allow future users to easily customize the framework for their needs.

- *Generic:* The framework can handle any type of problem for which a component-based system can provide a solution. It can incorporate different methods for modeling and decomposition of requirements, specification of and reasoning over preferences, discovery of available components, composition of selected components, and verification that a system satisfies a given set of traits. All that is needed to customize the framework for a given type of problem is to supply the appropriate modules and data for performing each task in the system development workflow.

- *End-to-end:* The framework provides support for component-based system development from the initial requirements-gathering and problem-modeling stages all the way through to delivery of the completed system.

- *Semi-automatic:* The framework automates many time-consuming low-level tasks in the component-based system development process, such as identifying the space of possible systems, determining which possible systems will be most preferred, and verifying a system's correctness with respect to a given requirement. This allows the system's developers and stakeholders to focus on higher-level tasks such as identifying and refining the requirement and preferences for the system. If the framework identifies more than one "most preferred" system, the stakeholders have the option to manually examine the set of results and choose a system or to allow the framework to arbitrarily choose a solution. The intent is not to replace humans in the component-based system development process, but rather to allow them to work more effectively.

As mentioned in Chapter 1, this framework essentially amounts to *a component-based system for developing component-based systems.* It can identify and construct component-based systems for solving problems in a wide variety of application domains, given a correct set of modules for handling problems in that domain and appropriate sets of entities to use in creating component-based solutions to those problems. Furthermore, modules within this framework can be easily replaced with other modules to support alternative approaches for identifying and creating optimal component-based systems. This property allows our framework to potentially be used as a testbed for comparing different tools or techniques for each part of the system development process.

Our framework comprises five stages, which are shown in more detail in Figure 6.1:

1. Defining the problem space.

2. Identifying the specific problem to solve.

3. Defining the solution space for the problem to be solved.

4. Identifying one or more solutions (system designs) that are optimal with respect to stakeholders' preferences.

5. Constructing an optimal component-based system to solve the given problem and verifying the correctness of the completed system.

The framework can be customized by inserting relevant modules into the framework where they are needed. In the medical records management example from Chapter 4, system designers would use modules that provide techniques for specifying requirements for Web service compositions (defining the problem space), apply the meta-framework from Section 4.2 to decompose the requirement (identifying the problem), access service repositories and read service descriptions (defining the solution space), decide which combination(s) of services will provide the needed functionality and the most preferred non-functional properties (identifying optimal solutions), and execute an on-the-fly composition and verification technique like that described in Section 4.4 to construct a system and verify it against each trait in its verification obligation (verifying solution correctness). Likewise, to create an optimal online book selling system based on the example in Chapter 5, system designers would combine a goal-oriented requirements engineering module with a CI-net preference modeling module to define the problem space and the specific problem, include modules for the search technique(s) of their choice to define the solution space, call a module that combines satisfiability (SAT) solving with CI-net preference reasoning to order possible system designs from most to least preferred as described in Section 5.4, and then verify the correctness of possible system designs from most to least preferred using verification modules as specified by each design's verification obligation.

In Sections 6.1 through 6.5, we will discuss our vision for the operations that should occur during each stage within this framework. We will also give a running example throughout these sections, originally created by Ali et al. in [1] and modified for use in our work in [58], which demonstrates how each stage of our framework can be applied to create a component-based system called *HelpMeOut* that supports the operations of a roadside assistance business. Section 6.6 discusses current implementations of our framework, modules that we have previously produced for use with this framework, and planned long-term work toward evaluating our framework as well as making our framework more fully unified and easier to use.

6.1 Defining the Problem Space

The first task to be completed is to define the space of problems to be solved using the framework. This represents the initial customization of the system design framework, as ap-

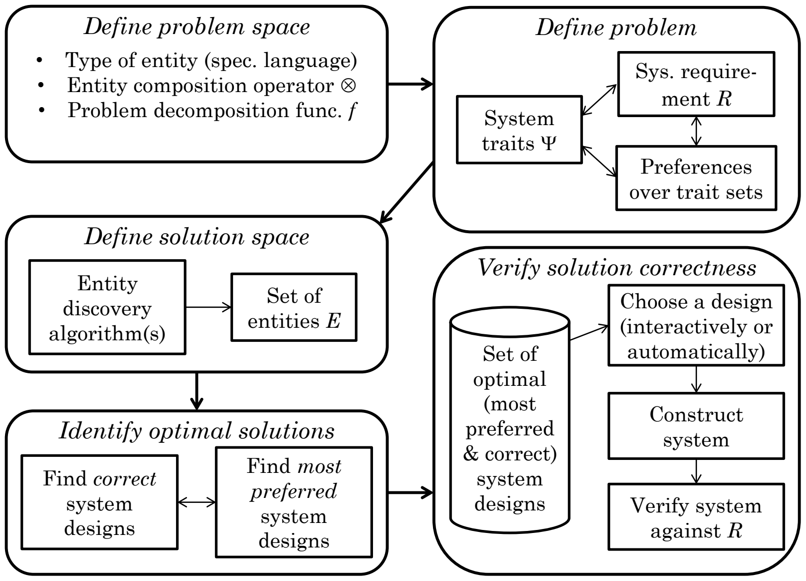

Define problem space
Type of entity (spec. language)
Entity composition operator ⊗
Problem decomposition func. f
Define problem
System traits Ψ
Sys. requirement R
Preferences over trait sets
Define solution space
Entity discovery algorithm(s)
Set of entities E
Identify optimal solutions
Find correct system designs
Find most preferred system designs
Verify solution correctness
Set of optimal (most preferred & correct) system designs
Choose a design (interactively or automatically)
Construct system
Verify system against R

Figure 6.1 Structure of our framework for developing optimal component-based systems

propriate modules need to be chosen or developed for each of the following aspects of the application domain where the framework is to be applied:

1. The *type(s) of entities* that are eligible to be selected as components of the system. This includes any common assumptions that can be made about the type(s) of entities being considered, any standard specification language(s) that may be used to describe such entities, or any other information that can be used to reduce the problem space and therefore the difficulty of finding solutions to such problems.

2. The *composition operator(s)* $\otimes$ that will be used to compose components into a unified system.

3. The *problem decomposition function* f that will be used to indicate the ways in which problems may be decomposed into subproblems and subproblems may be related to one another. This decomposition function must provide semantics for determining whether a component-based system satisfies a given problem.

Note that the composition operator(s) $\otimes$ and the problem decomposition function f work together to determine whether a given composition of components satisfies a given (sub)problem.

Example. At the start of the system design process, the system designers learn that the stakeholders in the system (in this case, primarily the owners of the roadside assistance business) would like to develop a system that makes it easier for a vehicle's driver to call them for assistance in case of a roadside emergency. Furthermore, they would like to develop the system using publicly available Web services that can tie into their existing dispatching and billing systems. The system designers suggest that the system be developed as a Web service composition, and the stakeholders agree.

As soon as the decision about the type of solution to be provided is made, the system designers know what type of entities they will use to develop the system, and they can also make several important assumptions about the components of the system and the ways in which they can be composed. Specifically, Web services are self-contained, self-describing, and frequently specified using one of several relatively standard specification languages (e.g., WSDL [17] or BPEL [55]). The system designers can use these assumptions to select or create a "module" of techniques that are targeted toward developing and verifying Web service compositions, as opposed to other types of component-based systems. They also select one or more service composition methods $\otimes$ and any appropriate problem decomposition function(s) f. In this case, the VeriComp service composition algorithm presented in [58] is selected as the entity composition operator $\otimes$ and our "meta-framework" for decomposing a composition design problem [59, 62] is selected as the problem decomposition function f. With this finished, the system designers and stakeholders can proceed to define the specific problem to be solved.

6.2 Identifying and Modeling the Specific Problem to Solve

The process of identifying the system design problem to be solved consists of three steps: identifying the set of traits Ψ to be considered in the system design, defining the system requirement R, and determining stakeholders' preferences over individual traits or sets of traits. As shown in Figure 6.1, these three steps reinforce each other. One or more of these steps may need to be repeated multiple times to iteratively refine the problem specification until it fully represents the stakeholders' view of the problem.

6.2.1 Identifying the Traits for the System

The first step in developing a component-based system is to identify the set of traits that should be considered important enough to include in the problem specification for the system. This is essentially a "brainstorming" process in which stakeholders identify high-level traits that the system may be expected to possess, such as tasks that the system may accomplish or non-functional properties that the system might provide, along with low-level traits such as behavioral specifications that will ensure the continued correct operation of the system. This step may also include initial discussion about which traits should form a part of the system requirement and which traits should be considered optional. However, the full requirement, which will be expressed as a Boolean combination of the traits identified in this step, will not be formalized until later.

Although traits do not need to be fully specified and do not need to have a verification technique chosen in this step of the development process, stakeholders and system designers should consider what metrics, formal specifications, or other verification methods would be appropriate for each trait being considered. It will be necessary to meaningfully specify these traits before attempting to verify a system design for correctness (see Section 6.5). If a trait cannot be easily formalized or verified, stakeholders should consider decomposing it into simpler "sub-traits" that are easier to verify, using the concept of AND-OR decomposition as presented in Section 3.3 to track relationships between traits; these relationships can be reused later to help define the system requirement. The objective of this step is to identify a set of traits that will play important roles in the desired system, that can be specified in a verifiable way, and that cannot be easily decomposed into simpler sub-traits.

6.2.2 Defining the System Requirement

After identifying the initial set of traits, the system stakeholders must define the requirement R for the system. As presented in Section 3.3, the requirement is a set containing one or more sets of traits, where all traits in any one of these sets must be provided by a system in order for the system to be acceptable to its stakeholders. The requirement can include both high-level traits, which indicate functions or non-functional properties to be provided, and low-level traits, which define required system behaviors and are often specified using statements in a temporal or other formal logic. Essentially, the requirement R is a Boolean combination of traits that were identified in the previous step.

Any new or existing requirements elicitation technique or combination of such techniques may be used to define the requirement. However, at a minimum some form of problem decomposition must be used. The problem decomposition defines a function f, which indicates both how the overall requirement is decomposed into alternative sets of individual traits (e.g., using an AND-OR graph) and how multiple components that each provide certain traits can be composed into a system that satisfies the overall requirement (or sub-requirement). It is important that the composition function f provide a semantics for combining or aggregating the verification obligations of the selected components (i.e., the traits that each component is expected to provide) to form the unified verification obligation for the system. The function f must also provide a method for keeping track of the various verification techniques that must be used to verify that the entire system satisfies the traits in its verification obligation.

Additional traits may be identified during the requirement elicitation and decomposition process, as stakeholders discover additional high-level and low-level properties that components may satisfy or that the system is required to provide. This is to be expected. The newly identified traits can be added to the existing set of traits and incorporated into the requirement if necessary, then formalized before verifying a system design.

Example. Figure 6.2 presents the functional requirements for the *HelpMeOut* system using a goal model (AND-OR graph) as described in Section 5.1. The required functions include collecting the vehicle's location and problem, searching for a nearby point of assistance, locating a mechanic that can visit the user, receiving payment, and reporting the event. Intermediate sub-requirements (goals) appear in round-edged boxes in Figure 6.2, while basic high-level traits that may be realized from available services (tasks) are shown in hexagons.

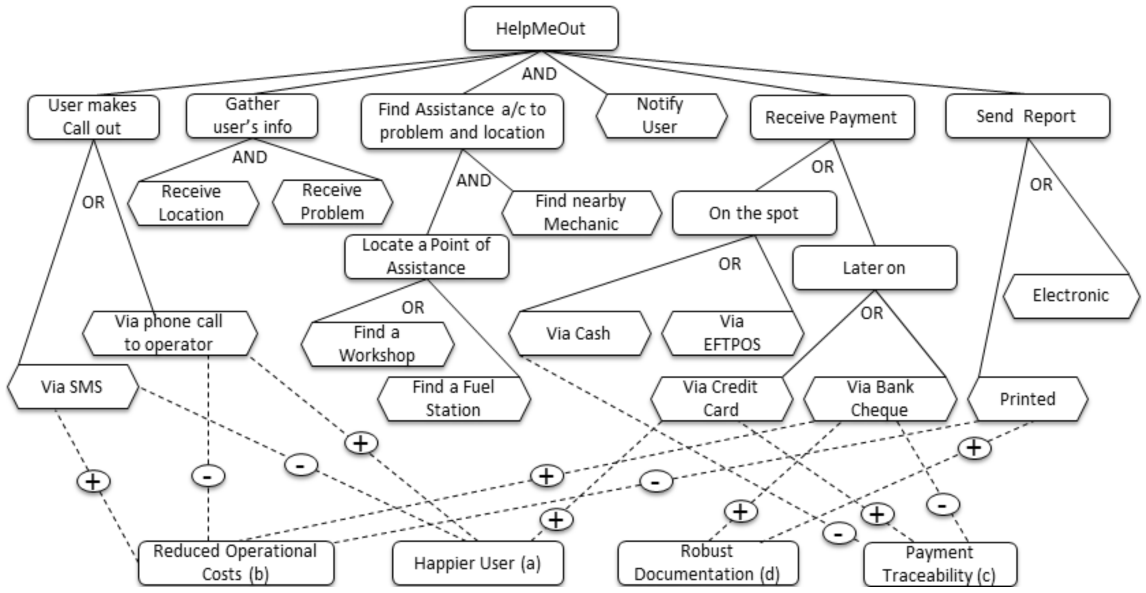


Figure 6.2 Goal model for *HelpMeOut* service

The graph illustrates dependencies between the requirements. For instance, the root node (level 0) describes the overall functionality to be provided. This functionality is fully realized if all of the sub-requirements at level 1 are satisfied (AND-decomposition). In contrast, the sub-requirement *Receive Payment* is satisfied if either one of the high-level traits *On the Spot* or *Later On* is satisfied (OR-decomposition).

While these functional requirements describe the necessary functions to be provided by the system, behavioral constraints encoded as low-level traits (not shown in Figure 6.2) ensure correct interaction or ordering of the services participating in the composition. These behavioral constraints are expressed as properties specified in a temporal logic, e.g., CTL [21] or LTL [71]. For example, *HelpMeOut* requires that if the *EFTPOS* service or the *Cash* service is used for payment on the spot, then a printed report should be sent instead of an electronic report. Consequently, the corresponding low-level trait becomes a part of the requirement for the system if either one of these services is chosen to form part of the system.

6.2.3 Determining Preferences over Traits

The stakeholders also need to determine their preferences among various traits of the system. Preferences may be used in a number of different ways within the development process:

they may be used to express preferences between different ways of satisfying the system requirement, to decide which of several optional features to include as part of the finished system, or to communicate other information as the stakeholders desire. Stakeholders express their preferences by defining a dominance relation between sets of traits as shown in Section 3.4, but they do not need to define this relation explicitly. Instead, system developers can use a preference modeling formalism, such as CI-nets [12] or TCP-nets [14], to create a dominance relation from informal preferences expressed by the stakeholders. Any preference formalism that satisfies the constraints in Section 3.4 is compatible with our component-based system development framework.

Example. Along with functional requirements, Figure 6.2 captures dependencies between non-functional properties (NFPs) and services. The NFPs, which are modeled here as optional traits of the system, are represented in boxes which are connected to functional requirements (goals) via edges annotated with “+” or “-”. The “+” annotation represents the satisfaction of the functional requirement having a positive impact on the NFP (a MAKE contribution link), while the “-” annotation represents a negative impact (a BREAK contribution link). For instance, satisfying the requirement that the user can contact an operator via a phone call may result in a happier user (positive impact) but will have a negative impact on reducing operational cost.

It may not be possible to consider a set of basic requirements such that (a) they have only positive impacts on the NFPs, (b) all NFPs are considered, and (c) the root-level requirement is satisfied according to the problem decomposition function. Therefore, preferences and trade-offs over NFPs are important for identifying a preferred set of basic requirements that result in satisfying the overall requirement for the system. Consider the following preference statements:

1. If robust documentation is used, payment traceability is more important than reducing operational costs.

2. If costs are reduced at the expense of customer satisfaction, then using robust documentation takes precedence over ensuring payment traceability.

These two preferences can be modeled naturally as CI-net statements. The first preference can be expressed as

$$\begin{aligned}&\{\textit{Robust Documentation}\};\{\}:\\&\{\textit{Payment Traceability}\}\succ\{\textit{Reduced Operational Costs}\}\end{aligned}\tag{6.1}$$

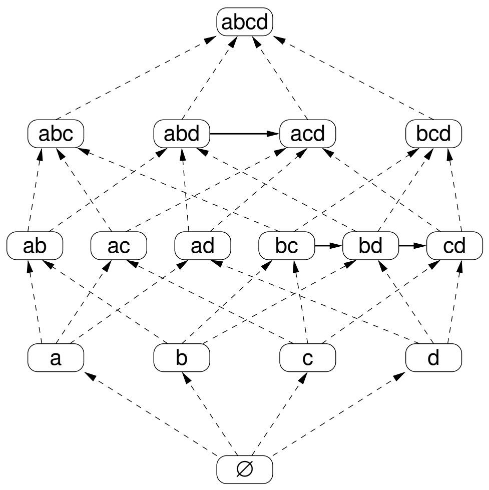
while the second preference can be expressed as

$$\begin{aligned}&\{\textit{Reduced Operational Costs}\};\{\textit{Happier User}\}:\\&\{\textit{Robust Documentation}\}\succ\{\textit{Payment Traceability}\}\end{aligned}\tag{6.2}$$

Figure 6.3 shows the induced preference graph corresponding to the preferences expressed by these CI-net statements. As in Section 5.3, each directed edge in the graph represents an improving flip from a less-preferred set of NFPs to a more-preferred set. Dashed edges (e.g., from $\{c\}$ to $\{bc\}$) indicate *monotonicity flips*; here, the NFP set {*Reduced Operational Costs, Payment Traceability*} is preferred because it is a proper superset of the NFP set {*Payment Traceability*}. Solid edges (e.g., from $\{bc\}$ to $\{bd\}$) indicate *importance flips*, which are induced by a specific CI-net statement (in this case, by statement 6.2 above). Figure 6.3 shows that the set of all NFPs is most preferred, while the empty set (no NFPs satisfied) is least preferred. This reflects the natural preference of the stakeholders for the system to provide as many NFPs as possible.

6.3 Defining the Solution Space

At this point in the system development process, the entire problem model has been constructed. The next step is to understand the set of possible solutions, i.e., the set of possible systems that can be created to solve the given problem. Because a system is represented in our framework as a composition of components, the set of possible systems (i.e., the *solution space*) $\mathbf{S}$ is a subset of the powerset of entities (possible components) E that are considered within the framework for a given problem; in notation, $\mathbf{S} \subseteq 2^E$. In general, $\mathbf{S} = 2^E$ in the absence of any constraints on the set of possible systems; however, this scenario is unlikely in practice, as there may be incompatibilities between entities (i.e., entities that cannot be composed with each other) or other constraints on the structure of the system which eliminate sets of entities from the set of possible systems $\mathbf{S}$.



a = *Happier User*
b = *Reduced Operational Costs*
c = *Payment Traceability*
d = *Robust Documentation*

CI-net statements:

6.1 $\{d\}, \{\} \ : \ \{c\} \succ \{b\}$
6.2 $\{b\}, \{a\} \ : \ \{d\} \succ \{c\}$

Figure 6.3 Induced preference graph for CI-net statements in *HelpMeOut* example

Defining the solution space for the problem is a two-step process. The first step involves identifying the set of entities E that are available to participate in a component-based system for this application domain. One or more modules for identifying the set of entities E must be provided by the system designers based on the type(s) of entities desired. Such a module may take the form of a list of available entities provided explicitly by the stakeholders, or it may be a program that computes the set of entities using a problem-specific search algorithm (or set of algorithms for different types of entities).

The second step is to determine which sets of available entities can be composed to form systems. A domain-specific module can be included in the framework at this point to efficiently eliminate sets of entities that are incompatible with each other from consideration. The module may use any technique to perform this reduction as long as it correctly returns $\mathbf{S}$, i.e., the set of all sets of entities that can be composed to form systems. By default, if no module is specified, the framework uses a "brute-force" algorithm to complete this process, listing every set of entities in 2^E and checking whether the entities in each set can be composed to form a system. Note that this process is intended only to remove sets of entities that cannot be composed together from consideration, not to consider whether the resulting systems are *correct* (i.e., whether they satisfy the given requirement R).

Example. Let us suppose that the developers of the *HelpMeOut* service have access to a repository of services (i.e., a set of entities or possible components E) that are available for use in the composition. Each service in the repository is specified in a standard service specification language such as WSDL [17] or BPEL [55], which describes the service's high-level functionality (semantics) as well as its inputs, outputs, and low-level behavior. A *labeled transition system* (LTS) for each service in E, which captures the dynamics of that service, is then extracted from this complete specification. The extraction of the LTS for each service can be done automatically using any existing method for transforming service descriptions into LTSs, such as those in [9] or [65]. These methods may be incorporated into our framework as separate modules. LTSs for the services in our repository are depicted in Figure 6.4. Because the *PhoneCall* and *SMS* services serve only as interfaces between a user and the system, their LTSs are not shown to reduce complexity.

The solution space for this problem is the set $\mathbf{S}$ of combinations of services (components) chosen from the repository E that are *compatible* with each other, meaning that the services in that combination can be composed to form a working service composition (system). Some of the possible service compositions in $\mathbf{S}$ can be obtained by comparing the inputs (resp. outputs) of each service to the outputs (resp. inputs) of every other service. If a given set of services contains at least one internal interface (i.e., between services in the set) that has an unavailable input, then the services in that set are not all compatible with each other. This input-output analysis should reduce the set of possible compositions to test in a relatively efficient manner, but it is susceptible to missing sets of components that are semantically compatible with each other even though the names of their matching inputs and outputs suggest otherwise. The technique for identifying and overcoming such "data mismatches" introduced by Ali et al. in [1] can be applied in this case to help identify possible compositions that would be missed otherwise. Other methods that make use of more sophisticated interface theories, such as interface grammars as used in [32] and [34], can also be used for more accurate and complete discovery of sets of compatible services. Techniques for adaptation of existing services, such as the one proposed in [67], can expand the space of possible solutions even further.

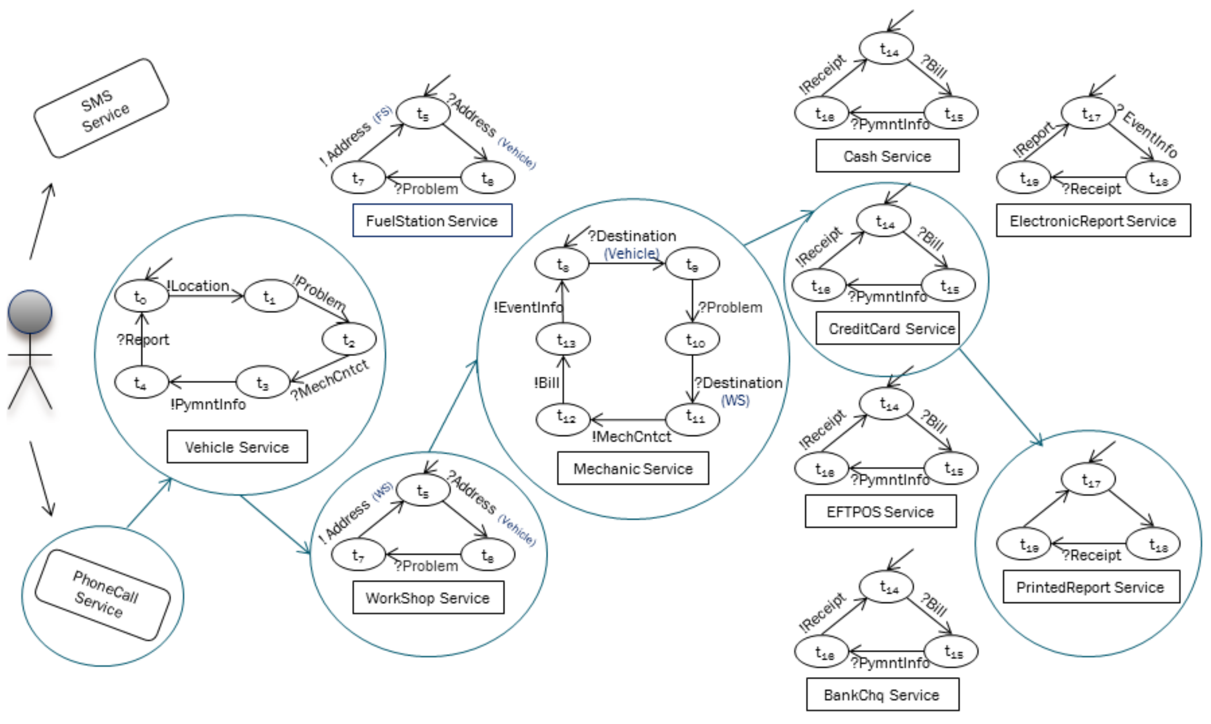


Figure 6.4 Repository of available component services for *HelpMeOut*. Circled services constitute the final composition.

6.4 Identifying Optimal Solutions for the Problem

After the solution space (i.e., the set of all sets of components that can be composed to form systems) has been determined, any appropriate technique(s) can be used to search the solution space in order to identify the set of correct system designs $D = \{C \subseteq E : vo(f(C), R) \neq \emptyset\}$. This search technique (or set of techniques) must also be supplied to our system development framework as a separate module. Different modules or combinations of modules may be used depending on the problem decomposition function f being used. In some cases, an iterative composition algorithm such as the one used in [60] or [62] may prove useful. In other cases, some other sort of search algorithm may be a better choice.

After possible system designs that do not satisfy the requirement R are eliminated from consideration, a qualitative preference reasoning technique is used to determine which of the correct system designs in D are most preferred according to the dominance relation $\succ$ specified by the stakeholders earlier in the process (see Section 6.2.3). Like the other customizable parts of this framework, the preference reasoning technique to be used is specified by providing a module, which generates calls to the tool that implements the reasoning technique and then processes the results from this tool. This preference reasoning process computes the set of *most preferred correct designs* for the system. Depending on the preferences that are specified, there may be more than one "most preferred" correct system design. Because the dominance relation $\succ$ defines a partial order over the set of designs D, if the most preferred system(s) cannot be verified to satisfy the system requirement, the next-most-preferred system(s) can be identified, composed, and verified; this process may be repeated until a system that satisfies the stakeholders' requirement is identified [60]. Alternatively, if there are few possible system designs to consider, then it may be advantageous to determine which designs can be composed and verified before applying preference reasoning; see Section 5.4 for a fuller discussion of the relative merits of each approach.

Example. The developers of the *HelpMeOut* service use the methods presented in [58] to compute the set of feasible service compositions (system designs) D. As part of this framework, a module named NextPref uses the non-functional properties (NFPs) and the CI-net statements describing the preferences and tradeoffs over them to compute an ordered sequence $\gamma_1, \gamma_2, \ldots, \gamma_n$. Each γ_i in the sequence represents a subset of the NFPs where $\gamma_{i+1} \not\succ \gamma_i$ with respect to the CI-net statements. In other words, the sequence of γ_is forms a total order consistent with the partial order of the induced preference graph. Based on the techniques described in Chapter 5, the NextPref module represents the induced preference graph (see Figure 6.3) as an input model of a standard model checker (specifically NuSMV [20]) and identifies sequence $\gamma_1, \gamma_2, \ldots, \gamma_n$ by verifying carefully selected temporal properties of the induced preference graph.

Let us examine how the induced preference graph shown in Figure 6.3 is used to generate the sequence of sets of NFPs from most to least preferred. To conserve space, let a = *Happier User*, b = *Reduced Operational Costs*, c = *Payment Traceability*, and d = *Robust Documentation*. Clearly $\gamma_1 = \{a, b, c, d\}$ is the most preferred set of NFPs. Next, consider all sets of NFPs with edges pointing to $\{a, b, c, d\}$ in the graph. Figure 6.3 contains no edges between three of these sets, which means that none of them are strictly preferred to each other; however, the graph does contain an edge from $\{a, b, d\}$ to $\{a, c, d\}$, which is induced by CI-net statement 6.1. Therefore, we assign $\gamma_2 = \{a, b, c\}$, $\gamma_3 = \{b, c, d\}$, and $\gamma_4 = \{a, c, d\}$ (although these sets could be in any order). We then assign $\gamma_5 = \{a, b, d\}$, as it is strictly less preferred than γ_4 according to Figure 6.3. This process continues until all sets of NFPs (including the empty set) have been placed into the sequence, following the process described in Section 5.3.3.

The other part of the framework from [58] that is applied at this stage is the ServSelect module. This module takes into account the goal model representing the overall functional requirement R and its relationship with the NFPs, the repository E of available services, and the sequence of NFP sets γ_i in order of preference starting from γ_1. For each γ_i, the module identifies

- the set of services X_i^+ that realize functional requirements which have *only positive impacts* on the non-functional properties in γ_i; and
- the set of services X_i^- that realize functional requirements which have *some negative impacts* on the non-functional properties in γ_i.

Consider the NFP set $\gamma_4 = \{a, c, d\}$ given above, along with the goal model in Figure 6.2. Based on the dependencies between services that satisfy functional requirements (hexagons in Figure 6.2) and the NFPs that each service satisfies, ServSelect identifies $X_4^+ = \{$*PhoneCall*, *CreditCard*, *PrintedReport*$\}$ and $X_4^- = \{$*SMSCall*, *Cash*, *BankChq*$\}$. Services in X_4^+ have only positive impacts on the NFPs in γ_4, while services in X_4^- have a negative impact on some NFP in γ_4.

Next, ServSelect solves the Boolean satisfaction problem encoded by the goal model to identify the set $\mathbf{S}_R$ of all sets of services C such that the composition of all services in C realizes a set of functional requirements which, when satisfied, result in satisfaction of the overall requirement R. Note that the presence of OR-nodes in the goal model allows R to be satisfied in multiple ways. Finally, ServSelect verifies $X_i^+ \subseteq C$ and $X_i^- \cap C = \emptyset$ for the selected C. Satisfaction of these conditions ensures that C is the most preferred set of services that satisfy both R and the non-functional properties in γ_i. If the conditions are not satisfied by any assignment (set of components) C, the module considers γ_{i+1} from the sequence of

γ_is. This is repeated until a suitable service set C is obtained. In the worst case, the least-preferred (empty) NFP set γ_n will be used; when this occurs, $X_n^+ = X_n^- = \emptyset$, making the above conditions vacuously true. Therefore, a non-empty set C will always be obtained if one exists.

Initially, ServSelect uses the goal model in Figure 6.2 and the repository of services that includes all services in Figure 6.4 to identify all possible compositions of available services that may satisfy the overall functional requirement R. Recall the sequence of NFP sets identified by the NextPref module previously. Observe in Figure 6.2 that there exists no combination of low-level functionalities that leads to satisfaction of γ_1 (all NFPs), γ_2, or γ_3. Fortunately, the set of services $C = \{$*PhoneCall*, *Vehicle*, *WorkShop*, *Mechanic*, *CreditCard*, *PrintedReport*$\}$ satisfies the required conditions for γ_4: $X_4^+ \subseteq C$ and $X_4^- \cap C = \emptyset$.

6.5 Constructing an Optimal Solution and Verifying Correctness

Now that one or more preferred system designs have been identified, the final step in the development process is to verify that a preferred system design $S = f(C)$ satisfies one of the sets of traits in its verification obligation $vo(f(C), R)$. This can be done by identifying the verification methods M_i that are to be used to verify each trait ψ_i in the verification obligation. (Note that any traits which have not yet been fully specified must be given a specification and a verification method before proceeding any further.) Because the composition function f tracks both the verification obligation for the given system design (set of components) and the verification methods M_i used to verify the appropriate set of traits, it is possible to automate the process of calling the necessary verification methods (through one or more modules that provide an interface to the method M_i specified by each trait ψ_i in the verification obligation), tracking the verification results for each required trait, and finally determining whether the verification obligation has been satisfied.

At this point, one of the identified optimal (most preferred correct) system designs is selected to be composed and verified. If desired, the stakeholders may review the available optimal system designs and choose a design that they think appears especially promising; otherwise, an optimal system design can be selected arbitrarily by the framework. The appropriate modules are called to compose the selected components into a unified system and then verify that the resulting system satisfies its verification obligation. If both composition and verification of the system are completed successfully, then the system development workflow is completed and an optimal correct component-based system has been created to solve the given problem. Otherwise, another optimal system design is selected to be composed and verified.

This process is repeated until the set of most preferred system designs is exhausted or until a system design is successfully composed and verified against its verification obligation. If no optimal system can be verified to satisfy its verification obligation, then the next-most-preferred correct system designs are identified and verified one by one; these form the new set of most preferred solutions. The framework proceeds by identifying, composing, and verifying gradually less preferred solutions one by one until a system passes verification or until all possible systems are exhausted.

Example. In the framework from [58] that the developers of the *HelpMeOut* composite service are employing, the OrchAndVerif module takes as input the set $\mathbf{S}_R$ of sets of services C from the ServSelect module and the set of low-level behavioral constraints Ψ expressed in CTL. The OrchAndVerif module then verifies whether there exists an orchestration of the services in C that satisfies Ψ. The core of the verification technique, which was primarily developed by Ali et al., is a tableau algorithm that takes services in C and constructs their orchestration in a goal-directed fashion, possibly including interleaving of services; details of the technique are available in [1]. If the verification fails, a different C is selected from $\mathbf{S}_R$ and the process is repeated until a suitable C is identified (*success*) or all elements of $\mathbf{S}_R$ have been considered (*failure*). Successful termination of the process results in a set of services which (1) satisfies the overall requirement R, (2) satisfies all behavioral constraints Ψ, and (3) is most preferred with respect to CI-net preferences over the set of NFPs.

Figure 6.5 presents the successfully generated orchestration of the most preferred set of services (given in the previous subsection) that fulfills the given behavioral constraints. Recall that the *PhoneCall* service serves as an interface only, so it is omitted from Figure 6.5 to reduce complexity.

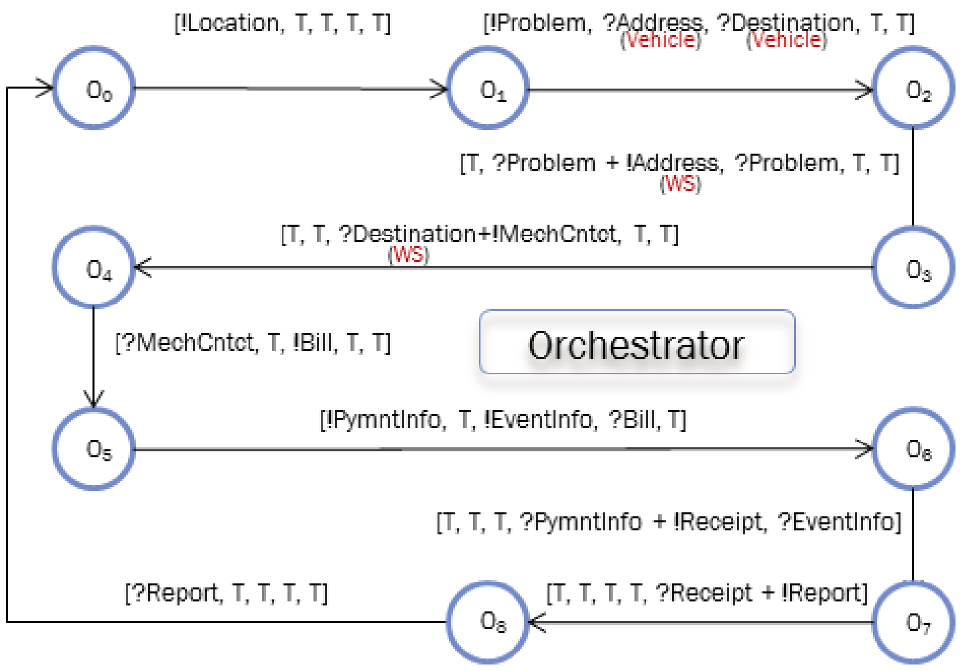

[!Location, T, T, T, T]
[!Problem, ?Address, ?Destination, T, T]
(Vehicle)
(Vehicle)
O0
O1
O2
[T, ?Problem + !Address, ?Problem, T, T]
(WS)
[T, T, ?Destination+!MechCntct, T, T]
(WS)
O4
O3
[?MechCntct, T, !Bill, T, T]
Orchestrator
[!PymntInfo, T, !EventInfo, ?Bill, T]
O5
O6
[T, T, T, ?PymntInfo + !Receipt, ?EventInfo]
[?Report, T, T, T, T]
[T, T, T, T, ?Receipt + !Report]
O8
O7

Figure 6.5 States and transitions of the synthesized service composition for *HelpMeOut*. Services are ordered as: [*Vehicle*, *WorkShop*, *Mechanic*, *CreditCard*, *PrintedReport*].

6.6 Implementation and Evaluation Plan

We define an *instance* of our component-based system development framework as a collection of modules that collectively provide sufficient support for stakeholders and system designers to (1) define the problem space, (2) formulate the specific problem to solve, (3) define the solution space for the problem being solved, (4) identify optimal potential solutions from this solution space, and (5) construct and verify an optimal component-based system so that the completed system is proven to satisfy its verification obligation. Note that as a direct result of the modular design of our framework, essentially any related technique for preference reasoning, problem analysis, component composition, system verification, etc. can be freely substituted into any instance of our framework as long as it is encapsulated within a "module" that describes the interface between that technique and the rest of the framework. In fact, one of the goals of developing this framework is to provide a testbed within which different tools, techniques, and algorithms for each part of the component-based system development process can be examined empirically in a well-defined and controlled environment.

We have implemented one complete instance of our system development framework: the VeriComp framework for Web service composition [58] that has been used as a running example throughout this section. Because VeriComp incorporates a set of shared assumptions about all Web services (self-describing, publicly available, etc.), it *defines the space of problems* that can be solved using Web service composition. VeriComp combines our work on goal-model analysis to *define the problem to solve* and preference reasoning to *identify optimal solutions* [60, 61] with techniques developed by Ali et al. in [1] specifically for composing Web services and verifying required traits of the resulting compositions (*realizing and verifying an optimal solution*). The VeriComp framework also *defines the solution space* by combining customizable connections to searchable repositories of available services (i.e., components) with novel methods introduced by Ali et al. in [1] for expanding the space of possible service composition solutions by resolving "data mismatches", which cause semantically compatible input-output pairs to be rejected by many service discovery tools because they are syntactically incompatible (e.g., different names). It is clear that the VeriComp framework is an instance of our more generic component-based system development framework.

Additionally, we have developed several modules that could be combined with other available modules to form another instance of our framework. The stand-alone goal-oriented requirements engineering tool that we presented in [60, 61] supports formal definition of the *specific problem* to solve in terms of a goal model, partial specification of the *solution space* in the form of different combinations of tasks that would satisfy the root goal for the system, and support for *identifying optimal solutions* by reasoning with a CI-net containing stakeholders' preferences [63] along with *verifying realized solutions* to ensure they fulfill the goals they promise to fulfill. The missing parts of this framework instance, which can (and should) be customized depending on the available components, are modules to define the problem space, locate available components that can be included in the system, and construct a system from a set of selected components. For example, if the system stakeholders prefer to use a preference reasoning formalism other than CI-nets, such as the Pareto optimality-based method applied in [62], this other formalism can be freely substituted into the framework by "plugging in" a module for the appropriate preference reasoning formalism. In fact, the service composition meta-framework in [62] provides an infrastructure to combine numerous formal specification languages with essentially any preference formalism, adding another level of modularity.

The main long-term goal for this framework is to provide a unified workflow for component-based system development supported by a standardized user interface, which will guide system stakeholders and software engineers alike through the stages of system development illustrated in Figure 6.1. In our view, the key factor that will allow for a unified system development workflow is the existence of a standard for specifying modules that will be compatible with our framework. This will require us to define a minimal sufficient interface for modules that aim to provide support for each stage of the development process, including a set of well-defined baseline inputs that our framework must expect from a module that targets each stage. Modules should also have the option to delegate the provision of some of these inputs to sub-modules in order to preserve maximal flexibility in the system development process. If such a general module specification standard can be established, it will be a significant step toward a truly "plug-and-play" modular component-based system development framework.

The current lack of a standardized user interface for our framework also limits its usefulness for component-based system developers. Part of the vision for our framework is a user interface that helps stakeholders and system designers visualize the consequences of the ways in which the problem to be solved and the stakeholders' preferences are modeled. We would like to provide a user interface that allows stakeholders to easily answer questions of the form, "What happens if we make this change to the model?" In our view, allowing stakeholders to explore the problem and solution spaces before doing the work of realizing and verifying an actual system is likely to reduce the total effort required for system development while resulting in component-based systems that better satisfy stakeholders' needs. We have previously developed (with Samik Basu, Ganesh Ram Santhanam, Carl Chapman, and Katarina Mitchell) a prototype graphical user interface (GUI) for CI-net and TCP-net preference reasoning, which could serve as a building block for a future unified GUI that can manage the entire system development framework. In addition to helping visualize aspects of the problem under consideration and its potential solutions, this GUI would be able to locate available modules and manage the logistical details of adding modules to or removing modules from the overall framework. This functionality would greatly simplify the process of gathering experimental data for comparing the performance of techniques proposed for use in component-based system development.

Future evaluation of our framework will attempt to answer two questions: whether the framework reduces development costs (both time and money) for component-based systems and whether the systems developed using our framework satisfy their stakeholders' needs better than systems developed using previous methodologies. Table 6.1 shows some of the criteria that we anticipate using to determine the contributions of our framework in these areas. The criteria in the left column, which are primarily meant to address the first question, are quantitative metrics that can be easily tracked while developing a system using our framework. These metrics can then compared to the same metrics for similar component-based system development projects completed without the help of our framework. The right column contains criteria that are targeted toward answering the second question. These criteria are more qualitative in nature, as they deal with stakeholders' and system developers' evaluations of the system development process as well as the final delivered system. Data for these criteria can be elicited through surveys administered both to system stakeholders and to system developers at various points in the development process.

One possible method of doing a preliminary evaluation along these lines involves asking students in software engineering courses to design component-based systems to solve any one of several problems, which could all be solved using components selected from a single repository of software entities (as defined in Chapter 3). Each problem would be solved by two teams of students: one team would solve the problem using our component-based system development framework with appropriate modules, while the other team would solve the same problem with the same basic techniques, but without our framework to tie them together into a well-defined process. The goal of this approach would be to measure the contribution provided by the modular end-to-end framework compared to simply using the individual modules separately.

Another possible evaluation approach would involve a software development group in industry with experience developing component-based systems adopting (and perhaps adapting) our framework to build two or more component-based systems for stakeholders within their organization. If the development group has metrics available from their prior component-based

Table 6.1 Criteria for evaluating improvements in component-based system development processes and results caused by our framework

Change in development cost and effort	*Satisfaction with process and results*
Directly observable project data, including: • Total developer effort (person-hours) • Total money spent on project • Total time from start of project to final system acceptance • Number of intermediate systems delivered before final system • Number of changes in requirement or preferences during development • Amount of developer effort per change in requirement or preferences	Data obtained from surveys of developers and stakeholders, such as: • Attitudes regarding component-based system development • Level of confidence in system designers and developers • Level of confidence in trait and requirement specifications • Level of understanding between stakeholders and developers • How much input or control stakeholders feel they have in process • How well final system meets stakeholders' needs compared to past systems

development projects, it may be possible to quantitatively compare our framework to their previous approaches in terms of developer effort, time or cost overruns, or other measures. Surveys would be administered to both the development group and the stakeholders for each system at the start of the development process, during development, and after acceptance of the initial production version of the system. The initial survey would be designed to measure prior attitudes and expectations about component-based system development and the system to be developed; the second survey would obtain opinions about the process, specifically communication between stakeholders and designers, during development; and the final survey would ask stakeholders to evaluate how well they feel the delivered system meets their needs in general as well as how the system compares to other component-based systems developed for them in the past.

We expect that our framework for component-based system development will produce systems that satisfy their stakeholders' needs and desires at least as well as systems developed using current methodologies, if not significantly better. Another benefit that we expect our framework to produce is a reduction in the number of intermediate systems, both prototypes and nearly complete systems, required to eventually produce a satisfactory system for the stakeholders; this will result in significantly reduced development costs and time-to-delivery. In addition, we anticipate that stakeholders' confidence in the overall process and in the developers producing the system will be significantly greater than with existing methods because of the improved communication between these two groups that our framework can facilitate.

CHAPTER 7. CONCLUSION

7.1 Summary

This dissertation presents and explains our novel framework for developing component-based systems that are *optimal*, meaning that they are *correct* relative to their specified requirement and *most preferred* with respect to their stakeholders' preferences over optional traits of the system. Although many approaches to solving this problem have been proposed and developed previously, this "component-based system for developing component-based systems" was motivated by two primary drawbacks of existing component-based system development frameworks. Most such frameworks do not provide support for modeling and verifying system requirements using diverse formal methods; rather, it is often expected that all properties to be formally verified will be specified using the same formal specification language, even though this may not produce ideal results. Additionally, current component-based development frameworks generally represent and reason with system stakeholders' preferences regarding non-functional properties and other optional traits of the system using primarily quantitative methods. Such methods are often time-consuming for stakeholders to use, produce results that may be difficult for stakeholders to correctly understand, and may identify "preferred" options that do not actually reflect the stakeholders' true preferences.

In response to these shortcomings of existing frameworks, the modular design of our framework for component-based system development allows support for various formal specification languages, diverse formal verification techniques, and a broad spectrum of qualitative preferences over sets of optional properties of the system to be incorporated directly into the system development process. Following the techniques described in Chapter 4, our framework guides stakeholders through the process of defining the problem that they wish to solve by decomposing the overall requirement for the component-based system into a collection of self-contained traits, some of which are optional and some of which are required. Each trait is specified in a certain way, ideally using a formal specification, and each trait provides a method by which its specification can be verified. The problem decomposition function for the given problem is then used in combination with a repository of software components to identify a set of components that are likely to provide a sufficient number of required traits to fulfill the overall system requirement.

As possible system designs (i.e., sets of components) that are likely to satisfy the overall requirement for the desired system are identified, qualitative preferences over the possible values of the optional traits under consideration are examined. If the given preferences are inconsistent, as determined by the methods described in Chapter 5, our framework explains to the stakeholders which of their preferences are inconsistent with each other and gives them an opportunity to resolve those inconsistencies. The framework then identifies which set(s) of components will be most preferred under this consistent set of qualitative preferences by using the techniques from Chapter 5 or other appropriate techniques. One such set of components is then composed using techniques and tools specified in a module for the type(s) of components being used. Finally, our framework verifies this optimal composition (if one can be realized) against its verification obligation, which is the entire set of required and optional traits that it is expected to provide, to ensure that it will fulfill its commitments to the system stakeholders. We provided an overview of the entire framework and an example of its use in Chapter 6. Any component-based system that is identified, composed, and verified according to this framework is an *optimal* (i.e., correct and most preferred) solution to the given component-based system development problem as long as the underlying modules produce correct results.

The advantages of our new framework for component-based system development compared to previous frameworks are:

1. The *modular* design of our framework allows it to be customized for a specific class of component-based systems by "plugging in" the appropriate modules to handle the common characteristics of that class of systems.

2. Because of this modular design, our framework is *generic* enough to use for developing any type of component-based system.

3. Our framework is *end-to-end*, meaning that it guides the new system's stakeholders through each step of the system development process.

4. Decisions about the problem being solved, the available components, and the preferred types of solutions are made by the stakeholders, but the difficult and time-consuming work of finding optimal solutions based on the given preferences and verifying candidate systems against the given requirements is automated by our *semi-automatic* framework.

7.2 Future Research

Three principal directions for future research in the context of our framework for component-based system development are prioritizing and reasoning with the preferences of multiple system stakeholders, developing more efficient techniques for automatically verifying the consistency of a given set of preferences, and providing formal support for partial satisfaction of traits throughout the system development framework. We explain the motivation for each of these research directions, along with the ways in which each one might affect the framework as it currently exists.

Reasoning with Multiple Stakeholders' Preferences. One shortcoming of our formulation of the component-based system design problem as given in Section 3 is that all stakeholders must agree on a single set of preferences over the traits of the system. In practice, this single set of preferences may reflect the desires of some stakeholders (e.g., senior management) while ignoring the preferences of other stakeholders. This can be solved by applying techniques for *representing and reasoning with multiple stakeholders' preferences*, such as those described in [95].

The basic approach that we envision involves modeling the preferences of each stakeholder along with a graph representing the relative hierarchy of the stakeholders with respect to their role in the system design problem. In general, the preferences of stakeholders whose roles place them higher in the hierarchy (e.g., CIO, project manager, or customer representatives) would be given precedence over preferences held by stakeholders who are lower in the hierarchy (e.g., programmers or support staff). Wherever higher-level stakeholders do not specify preferences between two sets of traits, lower-level stakeholders' preferences would be taken into account. Conflicts between higher- and lower-level preferences would be noted and presented to all involved stakeholders, as this could provide a useful communication channel to indicate possible future problems or justifications for preferences; this may facilitate better cooperation throughout the system development process. However, the currently envisioned multi-stakeholder preference reasoning framework would provide the option to automatically resolve conflicts between preferences at different levels as part of the process of identifying optimal systems.

Our existing preference reasoning tool already includes user interface support for multi-stakeholder preference reasoning according to this model. We plan to initially incorporate our multi-stakeholder preference reasoning method in the context of our existing tool for goal-oriented requirements engineering [60], which will serve as a prototype application of this method. If this prototype is successful, we will generalize the multi-stakeholder preference reasoning method so it can be fully incorporated into our framework and then applied to handle preferences for any type of component-based system.

Verifying Consistency of Preferences More Efficiently. A major challenge standing in the way of this approach for multi-stakeholder preference reasoning is the need for more efficient methods to check the consistency of a CI-net containing stakeholders' preferences. The model checking-based approach in Section 5.3.2 has the twin advantages of being both simple to implement (as long as an SMV-based model checking tool such as NuSMV or Cadence SMV is available) and provably correct, but a major drawback of this approach is its high time and space complexity. The model checker decides the consistency of a CI-net by attempting to verify that every path through the Kripke-structure representation of the induced preference graph eventually reaches a state where every optional property is satisfied. Unfortunately, because of the CI-net semantics (particularly the monotonicity rule) and because the number of nodes in

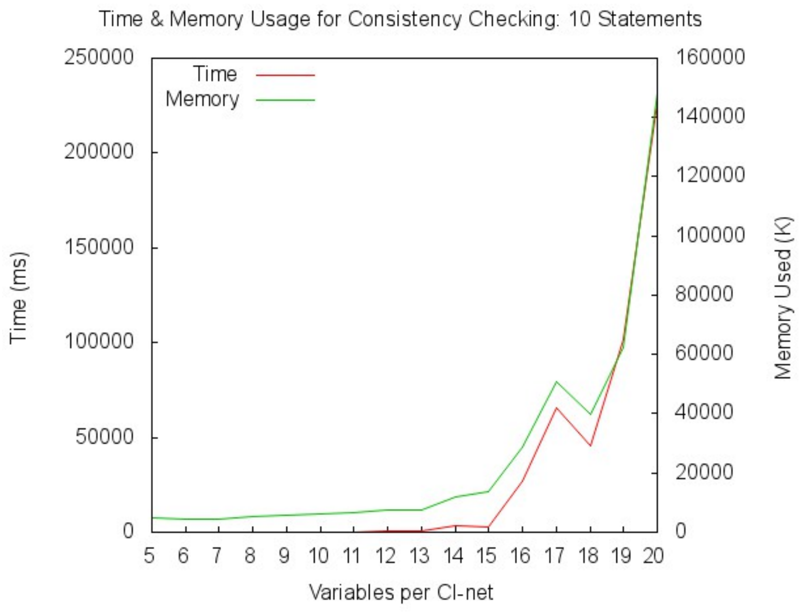


Figure 7.1 Time and memory usage for checking CI-net consistency with Cadence SMV [63]

the induced preference graph is exponential in the number of optional traits being considered, the number of paths to be verified by the model checker in the worst case is doubly exponential in the number of optional traits. This "exponential blow-up" in complexity, which is caused by the model checker's need to build the entire formal model before performing verification, makes automated consistency checking for CI-nets via model checking essentially intractable when dealing with large numbers of optional traits (variables) in the CI-net.

Figure 7.1 displays the results of an experiment we conducted to test the performance of consistency checking using a model checker in [63]. In this experiment, the exponential blow-up was especially apparent when performing consistency checking on CI-nets with more than 15 variables and exactly 10 preference statements using the Cadence SMV model checker. On average, consistency checking for CI-nets with at least 17 variables and exactly 10 statements required time on the order of minutes and memory on the order of tens of megabytes.

However, we hypothesize that it is possible to take advantage of the properties of the CI-net consistency checking problem to obtain an algorithm that simulates the model checking-based approach without always requiring all of the overhead of a model checking tool or even of

Tarjan's algorithm for detecting strongly connected components of a graph [85]. Such an algorithm would essentially perform an explicit least fixed point computation over the CI-net's induced preference graph, which is identical in nature to the least fixed point computation that is implied by the semantics of the *EF* construct in CTL (as applied in Section 5.3.2). Unlike the approach presented in [60, 63] and unlike Tarjan's algorithm, this new algorithm would not need to store the entire preference graph in memory, improving space efficiency and perhaps also time efficiency.

Modeling and Verifying Partial Satisfaction of System Traits. Another useful feature that we plan to add to our framework in order to better represent stakeholders' needs and desires is a *partial satisfaction semantics* for traits within a given system development problem. The inspiration for including a partial satisfaction semantics in our framework comes from the work of Mylopoulos et al. in goal-oriented requirements engineering [19, 35, 80, 93]. In [93], some goals are classified as *softgoals*, which are not entirely "satisfied" or "unsatisfied" in the traditional sense. Instead, softgoals are *satisficed* (a term coined by Simon [81]) if they are fulfilled "well enough" to satisfy the stakeholders' needs or *denied* otherwise. Mylopoulos and his collaborators define four types of contributions: MAKE (strong contribution toward satisficing), HELP (weak contribution toward satisficing), HURT (weak contribution toward denying), and BREAK (strong contribution toward denying). A softgoal is considered to be satisficed in the model of Mylopoulos et al. if it has enough positive contribution links to outweigh the negative contribution links. In [58] and [60], as well as in our examples in Chapters 5 and 6, we use a simplified version of this model that includes only MAKE and BREAK valuations.

Allowing for partial satisfaction of traits in the preference model could have a number of potential complicating effects on our component-based system development framework:

- Verification methods for partially satisfied traits must be able to indicate to what degree a given trait is satisfied.
- The system requirement may indicate the particular degree of satisfaction to be achieved by a given system.

- When composing entities, the partial satisfaction valuations for each trait must be aggregated using a consistent, possibly trait-specific aggregation function. This function must specify, among other things, whether (and if so, when) two entities that partially satisfy a trait can be composed in such a way that their composition fully satisfies the trait.

- It becomes possible to specify preferences of the form, "If traits A and B are both satisfied well enough (satisficed), then I prefer that A be more strongly satisfied than B." Such preferences are not supported by the qualitative preference reasoning methods we have used in the past, so new methods may need to be created or existing methods extended for this purpose.

Many interesting research problems in the areas of requirements engineering, preference reasoning, knowledge representation, and others are contained in these research directions. Taken together, though, these problems all center on the same basic motivation: how do we create new or improved systems from existing components that truly satisfy the actual needs of the system's stakeholders and that best fulfill their preferences over optional traits of the system while spending the minimum amount of time and money to do so? We believe this framework represents a significant step on the journey to an ultimate solution for this problem, and we are looking forward to seeing where the next steps in this journey lead.

BIBLIOGRAPHY

[1] Syed Adeel Ali, Partha S. Roop, Ian Warren, and Zeeshan Ejaz Bhatti. Unified management of control flow and data mismatches in web service composition. In Jerry Zeyu Gao, Xiaodong Lu, Muhammad Younas, and Hong Zhu, editors, *SOSE*, pages 93–101. IEEE, 2011.

[2] Gustavo Alonso, Fabio Casati, Harumi A. Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2004.

[3] Mohammad Alrifai and Thomas Risse. Combining global optimization with local selection for efficient QoS-aware service composition. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek, and Wolfgang Nejdl, editors, *WWW*, pages 881–890. ACM, 2009.

[4] Luca Anselma, Diego Magro, and Pietro Torasso. Automatically decomposing configuration problems. In Amedeo Cappelli and Franco Turini, editors, *AI*IA*, volume 2829 of *Lecture Notes in Computer Science*, pages 39–52. Springer, 2003.

[5] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: Case studies and experiments. In *ICSE*, pages 482–491, 2013.

[6] Mark A. Ardis, Nigel Daley, Daniel Hoffman, Harvey P. Siy, and David M. Weiss. Software product lines: A case study. *Software: Practice and Experience*, 30(7):825–847, 2000.

[7] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[8] Don S. Batory. Feature models, grammars, and propositional formulas. In J. Henk Obbink and Klaus Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[9] Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso. Automated composition of Web services via planning in asynchronous domains. *Artificial Intelligence*, 174(3-4):316–361, 2010.

[10] Jennifer Black and Philip Koopman. System safety as an emergent property in composite systems. In *DSN*, pages 369–378. IEEE, 2009.

[11] Craig Boutilier, Ronen I. Brafman, Carmel Domshlak, Holger H. Hoos, and David Poole. CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *Journal of Artificial Intelligence Research*, 21:135–191, 2004.

[12] Sylvain Bouveret, Ulle Endriss, and Jérôme Lang. Conditional importance networks: A graphical language for representing ordinal, monotonic preferences over sets of goods. In Craig Boutilier, editor, *IJCAI*, pages 67–72, 2009.

[13] Ronen I. Brafman and Carmel Domshlak. Preference handling — an introductory tutorial. *AI Magazine*, 30(1):58–86, 2009.

[14] Ronen I. Brafman, Carmel Domshlak, and Solomon Eyal Shimony. On graphical modeling of preference and importance. *Journal of Artificial Intelligence Research*, 25:389–424, 2006.

[15] Frederick P. Brooks. *The Mythical Man-Month.* Addison Wesley Longman, anniversary edition, 1995.

[16] Diego Calvanese, Giuseppe de Giacomo, Maurizio Lenzerini, Massimo Mecella, and Fabio Patrizi. Automatic service composition and synthesis: the Roman model. *IEEE Data Engineering Bulletin*, 31(3):18–22, 2008.

[17] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language version 2.0 part 1: Core language. W3C Recommendation, World Wide Web Consortium, June 2007.

[18] Eng U. Choo, Bertram Schoner, and William C. Wedley. Interpretation of criteria weights in multicriteria decision making. *Computers & Industrial Engineering*, 37(3):527–541, 1999.

[19] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering.* Kluwer Academic, 2000.

[20] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An open-source tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[21] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[22] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, January 2000.

[23] James Coplien, Daniel Hoffman, and David M. Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, 1998.

[24] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *ICSE*, pages 472–481, 2013.

[25] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *SPLC*, pages 23–34. IEEE Computer Society, 2007.

[26] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[27] Jon Doyle and Richmond H. Thomason. Background to qualitative decision theory. *AI Magazine*, 20(2):55–68, 1999.

[28] Neil A. Ernst, John Mylopoulos, Alexander Borgida, and Ivan Jureta. Reasoning with optional and preferred requirements. In Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson C. Woo, and Yair Wand, editors, *ER*, volume 6412 of *Lecture Notes in Computer Science*, pages 118–131. Springer, 2010.

[29] Judy Goldsmith, Jérôme Lang, Miroslaw Truszczynski, and Nic Wilson. The computational complexity of dominance and consistency in CP-nets. *Journal of Artificial Intelligence Research*, 33:403–432, 2008.

[30] Neeraj K. Gupta, Lalita Jategaonkar Jagadeesan, Eleftherios Koutsofios, and David M. Weiss. Auditdraw: Generating audits the fast way. In *RE*, pages 188–197. IEEE Computer Society, 1997.

[31] Joyce El Haddad, Maude Manouvrier, and Marta Rukoz. TQoS: Transactional and QoS-aware selection algorithm for automatic Web service composition. *IEEE Transactions on Services Computing*, 3(1):73–85, 2010.

[32] Sylvain Hallé, Graham Hughes, Tevfik Bultan, and Muath Alkhalaf. Generating interface grammars from WSDL for automated verification of Web services. In Luciano Baresi, Chi-Hung Chi, and Jun Suzuki, editors, *ICSOC/ServiceWave*, volume 5900 of *Lecture Notes in Computer Science*, pages 516–530, 2009.

[33] Rachid Hamadi and Boualem Benatallah. A Petri net-based model for Web service composition. In *14th Australasian Database Conference*, pages 191–200. Australian Computer Society, Inc., 2003.

[34] Graham Hughes and Tevfik Bultan. Interface grammars for modular software model checking. *IEEE Transactions on Software Engineering*, 34(5):614–632, 2008.

[35] Ivan Jureta, Alexander Borgida, Neil A. Ernst, and John Mylopoulos. Techne: Towards a new generation of requirements modeling languages with goals, preferences, and inconsistency handling. In *RE*, pages 115–124. IEEE Computer Society, 2010.

[36] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[37] Ralph L. Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. John Wiley and Sons, 1976.

[38] Gary T. Leavens and Murali Sitaraman, editors. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[39] Mark Shmuilovich Levin. *Combinatorial Engineering of Decomposable Systems*. Kluwer Academic Publishers, 1998.

[40] Mark Shmuilovich Levin. *Composite Systems Design*. Springer, 2006.

[41] Jingyue Li, Reidar Conradi, Christian Bunse, Marco Torchiano, Odd Petter N. Slyngstad, and Maurizio Morisio. Development with off-the-shelf components: 10 facts. *IEEE Software*, 26(2):80–87, 2009.

[42] Jingyue Li, Anita Gupta, Jon Arvid Børretzen, and Reidar Conradi. The empirical studies on quality benefits of reusing software components. In *COMPSAC (2)*, pages 399–402. IEEE Computer Society, 2007.

[43] Sotirios Liaskos, Marin Litoiu, Marina Daoud Jungblut, and John Mylopoulos. Goal-based behavioral customization of information systems. In Haralambos Mouratidis and Colette Rolland, editors, *CAiSE*, volume 6741 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2011.

[44] Sotirios Liaskos, Sheila A. McIlraith, Shirin Sohrabi, and John Mylopoulos. Integrating preferences into goal models for requirements engineering. In *RE*, pages 135–144. IEEE Computer Society, 2010.

[45] Wayne C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30, 1994.

[46] Naiwen Lin, Ugur Kuter, and James A. Hendler. Web service composition via problem decomposition across multiple ontologies. In *IEEE SCW*, pages 65–72. IEEE Computer Society, 2007.

[47] Zhiming Liu and He Jifeng. *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*, volume 2 of *Series on Component-Based Software Development*. World Scientific, 2006.

[48] Carsten Lutz, Frank Wolter, and Michael Zakharyaschev. Temporal description logics: A survey. In Stéphane Demri and Christian S. Jensen, editors, *TIME*, pages 3–14. IEEE Computer Society, 2008.

[49] Mike Mannion. Using first-order logic for product line model validation. In Gary J. Chastek, editor, *SPLC*, volume 2379 of *Lecture Notes in Computer Science*, pages 176–187. Springer, 2002.

[50] Kenneth L. McMillan. Cadence SMV (software). Release 10-11-02p1. Available at: http://www.kenmcmil.com/smv.html, 2002.

[51] Marcílio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. SAT-based analysis of feature models is easy. In Dirk Muthig and John D. McGregor, editors, *SPLC*, volume 446 of *ACM International Conference Proceeding Series*, pages 231–240. ACM, 2009.

[52] Andrew P. Moore. The specification and verified decomposition of system requirements using CSP. *IEEE Transactions on Software Engineering*, 16(9):932–948, 1990.

[53] Wonhong Nam, Hyunyoung Kil, and Dongwon Lee. Type-aware Web service composition using boolean satisfiability solver. In *CEC/EEE*, pages 331–334, 2008.

[54] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.

[55] OASIS Web Services Business Process Execution Language Technical Committee. Web Services Business Process Execution Language version 2.0. OASIS Standard, OASIS, April 2007.

[56] OASIS Web Services Secure Exchange Technical Committee. WS-SecurityPolicy 1.3. OASIS Standard, OASIS, February 2009.

[57] Ivana Ognjanovic, Dragan Gasevic, Ebrahim Bagheri, and Mohsen Asadi. Conditional preferences in software stakeholders' judgments. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 683–690. ACM, 2011.

[58] Zachary J. Oster, Syed Adeel Ali, Ganesh Ram Santhanam, Samik Basu, and Partha S. Roop. A service composition framework based on goal-oriented requirements engineering, model checking, and qualitative preference analysis. In Chengfei Liu, Heiko Ludwig, Farouk Toumani, and Qi Yu, editors, *ICSOC*, volume 7636 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2012.

[59] Zachary J. Oster, Ganesh Ram Santhanam, and Samik Basu. Decomposing the service composition problem. In Antonio Brogi, Cesare Pautasso, and George Angelos Papadopoulos, editors, *ECOWS*, pages 163–170. IEEE Computer Society, 2010.

[60] Zachary J. Oster, Ganesh Ram Santhanam, and Samik Basu. Automating analysis of qualitative preferences in goal-oriented requirements engineering. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *ASE*, pages 448–451. IEEE, 2011.

[61] Zachary J. Oster, Ganesh Ram Santhanam, and Samik Basu. Automating analysis of qualitative preferences in goal-oriented requirements engineering. Technical Report 11-06, Department of Computer Science, Iowa State University, 2011.

[62] Zachary J. Oster, Ganesh Ram Santhanam, and Samik Basu. Identifying optimal composite services by decomposing the service composition problem. In *ICWS*, pages 267–274. IEEE Computer Society, 2011.

[63] Zachary J. Oster, Ganesh Ram Santhanam, Samik Basu, and Vasant Honavar. Model checking of qualitative sensitivity preferences to minimize credential disclosure. In Corina Pasareanu and Gwen Salaün, editors, *FACS*, volume 7684 of *Lecture Notes in Computer Science*, pages 205–223. Springer, 2012.

[64] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

[65] Jyotishman Pathak. *Interactive and Verifiable Web Services Composition, Specification Reformulation, and Substitution.* PhD dissertation, Iowa State University, 2007.

[66] Jyotishman Pathak, Samik Basu, and Vasant Honavar. Modeling Web services by iterative reformulation of functional and non-functional requirements. In Asit Dan and Winfried Lamersdorf, editors, *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 314–326. Springer, 2006.

[67] Jyotishman Pathak, Samik Basu, and Vasant Honavar. Composing Web services through automatic reformulation of service specifications. In *IEEE SCC*, pages 361–369. IEEE Computer Society, 2008.

[68] Jyotishman Pathak, Samik Basu, Robyn R. Lutz, and Vasant Honavar. Parallel Web service composition in MoSCoE: A choreography-based approach. In *ECOWS*, pages 3–12. IEEE Computer Society, 2006.

[69] Rodrigo Mantovaneli Pessoa, Eduardo Goncalves da Silva, Marten van Sinderen, Dick A. C. Quartel, and Luís Ferreira Pires. Enterprise interoperability with SOA: a survey of service composition approaches. In Marten van Sinderen, João Paulo A. Almeida, Luís Ferreira Pires, and Maarten Steen, editors, *EDOCW*, pages 238–251. IEEE Computer Society, 2008.

[70] Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and Annapaola Marconi. Automated synthesis of composite BPEL4WS Web services. In *ICWS*, pages 293–301. IEEE Computer Society, 2005.

[71] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[72] Lianyong Qi, Ying Tang, Wanchun Dou, and Jinjun Chen. Combining local optimization and enumeration for QoS-aware Web service composition. In *ICWS*, pages 34–41. IEEE Computer Society, 2010.

[73] Thomas L. Saaty. Decision making with the Analytic Hierarchy Process. *International Journal of Services Sciences*, 1:83–98, 2008.

[74] Ganesh Ram Santhanam, Samik Basu, and Vasant Honavar. TCP-Compose* — a TCP-net based algorithm for efficient composition of Web services using qualitative preferences. In Athman Bouguettaya, Ingolf Krüger, and Tiziana Margaria, editors, *ICSOC*, volume 5364 of *Lecture Notes in Computer Science*, pages 453–467, 2008.

[75] Ganesh Ram Santhanam, Samik Basu, and Vasant Honavar. Dominance testing via model checking. In *AAAI*, pages 357–362. AAAI Press, 2010.

[76] Ganesh Ram Santhanam, Samik Basu, and Vasant Honavar. Efficient dominance testing for unconditional preferences. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *KR*. AAAI Press, 2010.

[77] Ganesh Ram Santhanam, Samik Basu, and Vasant Honavar. Representing and reasoning with qualitative preferences for compositional systems. *Journal of Artificial Intelligence Research*, 42:211–274, 2011.

[78] Stan Schenkerman. Use and abuse of weights in multiple objective decision support models. *Decision Sciences*, 22:369–378, 1991.

[79] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *RE*, pages 136–145. IEEE Computer Society, 2006.

[80] Roberto Sebastiani, Paolo Giorgini, and John Mylopoulos. Simple and minimum-cost satisfiability for goal models. In *CAiSE*, pages 20–35, 2004.

[81] Herbert A. Simon. *The Sciences of the Artificial.* MIT Press, 1981.

[82] Hongyu Sun, Samik Basu, Vasant Honavar, and Robyn R. Lutz. Automata-based verification of security requirements of composite Web services. In *ISSRE*, pages 348–357. IEEE Computer Society, 2010.

[83] Clemens Szyperski. Components and the way ahead. In Leavens and Sitaraman [38], chapter 1, pages 1–20.

[84] Clemens Szyperski and Cuno Pfister. Workshop on component-oriented programming (WCOP96), summary. In Max Mühlhäuser, editor, *Special Issues in Object-Oriented Programming — ECOOP96 Workshop Reader*, pages 127–130. dpunkt Verlag, Heidelberg, 1997.

[85] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[86] Maurice H. ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Web service composition approaches: From industrial standards to formal methods. In *ICIW*, page 15. IEEE Computer Society, 2007.

[87] Maurice H. ter Beek, Stefania Gnesi, Nora Koch, and Franco Mazzanti. Formal verification of an automotive scenario in service-oriented computing. In *ICSE*, pages 613–622, New York, NY, USA, 2008. ACM.

[88] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE*, pages 249–263. IEEE Computer Society, 2001.

[89] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. John Wiley and Sons, 2009.

[90] Yingzi Wang, Xiaolin Zheng, and Deren Chen. An ontology-driven discovery architecture to support service composition. In *ICEBE*, pages 365–370. IEEE Computer Society, 2009.

[91] David Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[92] Jung-Woon Yoo, Soundar R. T. Kumara, Dongwon Lee, and Seog-Chan Oh. A Web service composition framework using integer programming with non-functional objectives and constraints. In *CEC/EEE*, pages 347–350, 2008.

[93] Eric S. K. Yu and John Mylopoulos. Understanding 'why' in software process modelling, analysis, and design. In *ICSE*, pages 159–168, 1994.

[94] Kaizhi Yue. Validating system requirements by functional decomposition and dynamic analysis. In *ICSE*, pages 188–196, 1989.

[95] Yuanyuan Zhang, Mark Harman, Anthony Finkelstein, and S. Afshin Mansouri. Comparing the performance of metaheuristics for the analysis of multi-stakeholder tradeoffs in requirements optimisation. *Information & Software Technology*, 53(7):761–773, 2011.