GENERATING MEMBERS OF A SOFTWARE PRODUCT LINE USING COMBINATORY LOGIC

By

Armend Hoxha

A Thesis

Submitted to the Faculty

of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

May 2015

I. ABSTRACT

A *Product Line Family* contains similar applications that differ only in the sets of supported features from the family. To properly engineer these product lines, programmers design a common code base used by all members of the product line. The structure of this common code base is often an Object-Oriented (OO) framework, designed to contain the detailed domain-specific knowledge needed to implement these applications. However, these frameworks are often quite complex and implement detailed dynamic behavior with complex coordination among their classes. Extending an OO framework to realize a single product line instance is a unique exercise in OO programming. The ultimate goal is to develop a consistent approach, for managing all instances, which relies on *configuration* rather than programming.

In this thesis, we show the novel application of *Combinatory Logic* to automatically synthesize correct product line members using higher-level code fragments specified by means of *combinators*. Using the same starting point of an OO framework, we show how to design a repository of combinators using *FeatureIDE*, an extensible framework for Feature-Oriented Software Development.

We demonstrate a proof of concept using two different Java-based frameworks: a card solitaire framework and a multi-objective optimization algorithms framework. These case studies rely on *LaunchPad*, an Eclipse plugin developed at WPI that extends FeatureIDE.

The broader impact of this work is that it enables framework designers to formally encode the complex functional structure of an OO framework. Once this task is accomplished, then, generating product line instances becomes primarily a configuration process, which enables correct code to be generated by construction based on the *combinatory logic*.

II. ACKNOWLEDGEMENTS

When I initially got introduced to this research project, by Prof. George T. Heineman, I thought to myself "Is it really possible?". It turned out to be not only possible, but also one of the most interesting projects I've worked on so far. Therefore, I want to thank him, as my advisor, for his great support, who has helped me achieve this feat with his invaluable constant advice and encouragement.

To be able to work on this project I needed to stand on the shoulders of giants. Besides my advisor, I want to thank Boris Düdder and Jakob Rehof from the Technical University of Dortmund in Germany, who, together with Prof. Heineman, have made all this work possible by providing the tools and ideas that I have experimented with.

I also want to thank the reader of my thesis, Prof. Kathi Fisler, whose critiques helped me better shape this work.

I am very grateful to USAID Kosovo, too, for helping me realize my dream by granting me a full scholarship. I should also thank American Councils for administering the program I was enrolled in and overseeing my overall progress over these two years of graduate studies at WPI.

And, how can I forget? I want to thank my family for supporting me in every aspect of my life; in particular my father, who (while alive) has ingrained in me the urge for education; and my sister Mimoza, who has first introduced me to a computer (it was a turning point in my life) and has provided material and moral support throughout my studies in high-school and undergraduate school. And finally, the one who has been with me every single day, every step of the way, Divya.

III. TABLE OF CONTENTS

IV. LIST OF FIGURES

V. LIST OF TABLES

1. Introduction and Motivation

One common goal in software engineering is to assemble software systems from reusable software units rather than developing entirely from scratch [1]. Computer software has long used *code libraries* to support development, but these are meant to be used "as is" and do little to support extensibility or assembly. Instead, to solve the reuse challenge, one must identify the proper structure of a *unit* and the means by which the units are *composed*.

Many approach this goal by following a component-based development (CBD) strategy that relies on the assembly of systems from functional code units which can be independently constructed by third-party developers [2]. In the most general case, CBD ignores the actual language used to develop the components, relying instead on a component model implementation [3] that enables the assembly and integration of binary components. Other developers rely on an Object-Oriented (OO) language, such as Java or C++, to design abstract classes to represent fundamental abstractions and use inheritance and polymorphism to ensure reuse. This language based approach enables extensibility of existing code – something not easily supported by code libraries – but many believe there is an increased code in developing both the original code and the extensions. To address the problem of extensibility, programmers typically develop Object-Oriented (OO) frameworks that ease the burden of those seeking to extend them. Hereafter, whenever we say framework, we refer exclusively to an object oriented framework.

For our purposes, "*a framework is a set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of framework classes*" [4]. Framework developers provide common functionality for a family of applications so programmers need to write as little new code as possible to make their applications functional. A framework leverages domain experience of expert programmers and is designed to be reusable, extensible and specialized.

Generally, frameworks focus on a specific domain, therefore domain knowledge is encoded into a framework, and the design is abstract, because it's not a complete software application; it is meant to be extended by implementing extra classes to complete the application [5]. Usually the flexibility provided by a framework is not all needed by the application being developed, since applications are much more specific than frameworks. However, programmers still have to have a thorough understating of the framework to be able to use and extend it. The framework dictates the software architecture, design patterns and the flow of control of the application being developed. Thus, there is much more than just reusing the code provided by the framework, it's the abstraction the framework is built upon. Therefore, the learning curve of a framework is quite steep, typically 6-12 months [5]. All these obstacles make frameworks hard to use; often they can only be extended by the very programmers who developed the framework, which significantly decrease their reuse potential.

To maximize the use of a framework, its designers must rely on documentation to explain the behavior of a framework. It must provide the necessary information for a programmer to start using the framework. Documenting each class and method in the framework is important and needed, but it fails to accurately describe the abstractions upon which the framework is designed. As it is very hard to include abstractions in the code and design of a framework, they usually remain only in the heads of designers. The lack of knowledge about abstractions used to develop a framework makes it hard to extend or even use the framework.

Framework designers provide examples of how to use the framework, which is a good and quick starting point to using the framework. However, instead of simply providing sample code that programmers must read to identify the abstractions, should there be a way (other than coding) to represent the abstractions embedded in the code.

A software product line is a set of software-intensive systems developed from a common set of core assets in a prescribed way [6]. It is quite common to design the product line using an object-oriented application framework [7]. This project targets the process of generating product line members in a software product line. More specifically, the idea is to convert an object-oriented framework extension problem into a configuration problem, namely generation of product line members in a product line family.

1.1. Grand Challenge

The grand challenge addressed by this research is to synthesize software from modular units and guarantee that the resulting code is "correct by construction". This is a phrase adopted by the formal methods community that uses provably correct refinement steps to transform a specification into a design, which is ultimately transformed into an implementation that is correct by construction [8]. In this methodology, the specification is provided *a priori* and must typically be complete before the process can start. The correct by construction paradigm has been applied successfully to VLSI chip design [9] and safety critical applications [10]. It is hard to generally apply this method to arbitrary software because it relies on having a complete behavioral specification in advance.

Instead of relying on refinement as the process that produces code in stepwise fashion, we seek to synthesize code from modular units which already are defined and typed in relation to an existing framework. As argued by Robert Constable, "When types are rich enough to completely specify a task", then one can programmatically achieve correct by construction code [11].

The challenge is to identify the language needed to specify these modular units (are they fully implemented code units or do they only contain partial fragments?) and develop a tool supported by type theory to ensure that these units are composed properly to produce the synthesized code.

We refer to these modular units as combinators [12]. Thus, a combinator is a component in our repository of components, which is specified using a higher level language that we refer to as L2 language [12]. A combinator consists of two parts: definition and implementation. The definition part of a combinator is characterized by its name and its intersection type [13], the implementation part is a blend of the L2 and L1 languages, where L1 represents any of the programming languages (i.e. Java, C#, C, C++ etc.). For more details see Chapter 2.

1.2. Requirements

Earlier work by Heineman on solving this problem [14] used the AHEAD tool suite developed by Batory [15]. But this work had limited impact for a number of reasons [16]. First, the AHEAD tool suite generated solutions using Java classes formed by a hierarchy of abstract classes. This resulting code is unreadable and is so radically different from any code that one normally would expect to see produced by programmers. Second, while the composition process is governed by algebraic specifications, it was not possible to formally state anything about the code being synthesized. In other words, it was impossible to make any claims about the synthesized code.

As the current research program progressed, the research team (Boris Düdder, Jakob Rehof and George Heineman) identified the following requirements, which apply to the modular units that form the repository as well as the synthesized code that results.

- *Readable* – Programmers must be able to easily read and understand the synthesized code
- *Debuggable* – Programmers must be able to understand the runtime behavior of the synthesized code when executing the code within an IDE debugger
- *Type Safe* – Programmers must be able to assign logical type specifications to the modular units as well as the synthesized code generated from these units
- *Scalable* – Programmers must be able to intellectually manage a repository containing hundreds of combinators, much like they can manage object-oriented repositories with hundreds of classes
- *Compatible with IDEs* – Programmers must be able to continue to use existing integrated development environments (IDEs) when synthesized code is integrated into an actual project

We seek to improve the ability for software engineers to migrate existing OO frameworks into a software product line structure. The fundamental issue is that framework designers cannot cleanly encapsulate the abstractions of the framework using an OO language nor describe *in that same language* how the abstractions are intended to compose and interact with each other.

This thesis makes progress towards the overall goal of using configuration to synthesize individual members of a product line by (a) modeling the extension and usage patterns of the OO framework; and (b) building a repository of modular units for composition to enable code synthesis. We had already envisioned applying this research to an existing software product line developed at WPI, but we wanted to ensure the wider applicability of this approach to work with independently designed OO frameworks.

We can restate the grand challenge in the context of extending an existing object oriented framework. There are two possible paths to follow (see Figure 1). The first purely logical path (labeled **Semi-automated Support**) starts by designing a complete specification of the framework and then relying on advanced techniques to refine the logical specification into working code. The second approach (labeled **Significant Manual Effort**) relies on highly effective programmers who can understand the framework code base from documentation and sample code. We propose to make progress on a third approach which relies on specifications written by the framework designers, but which are associated with modular units that are fine-grained. With appropriate tool support, the programmer would be able to work more readily with these modular units to synthesize the same code that otherwise would have required significant effort to develop. At the same time, the resulting code is also functionally equivalent to the code that would have been synthesized from the logical specifications, but which is actually readable and understandable by programmers.

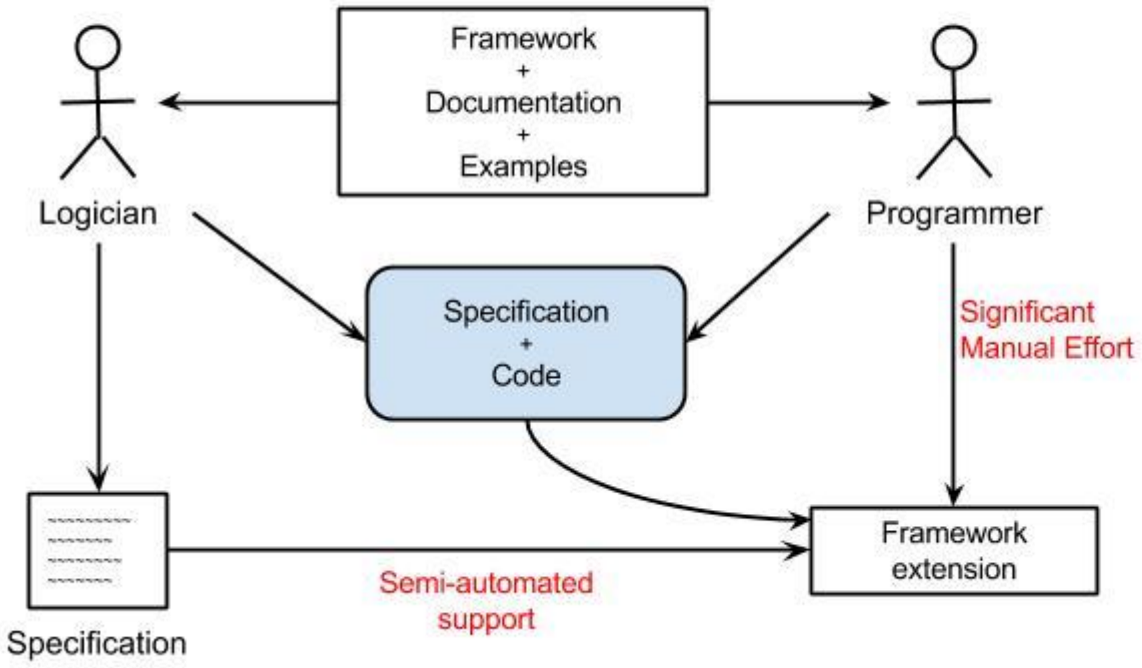


Figure 1: Competing Visions for Extending OO Framework

1.3. Overview

The remainder of the thesis presents the effort towards achieving our goal. In **Chapter 2** we describe the idea and the tools we have used to design our solution. We introduce the *Combinatory Logic Synthesis* (CLS) and the *inhabitation problem*, which lie at the heart of our approach towards code synthesis; then we describe the tools which realize CLS and the *inhabitation problem*. In **Chapter 3** we document in detail two case studies that we have conducted in two different object-oriented frameworks as a proof of concept that this approach can be used to generate non-trivial software applications. In **Chapter 4** we present the related work along with the results from our research on the documentation of OO frameworks. **Chapter 5** talks about the evaluation process of our approach, and we close with **Chapter 6**, which gives a perspective on the future work.

2. Design of Solution

In this chapter we explain the tools and the approach used in designing our solution. We briefly introduce *Combinatory Logic Synthesis,* which is the foundation of our approach; and the *inhabitation problem*, whose solution synthesizes code from a repository of modular units. We close by demonstrating the tools we used for our case studies.

2.1. Combinatory Logic Synthesis

Combinatory logic synthesis is a *type-based* approach to component-oriented synthesis [16]. The basic idea of CLS is to automate the composition from a repository using combinatory logic [17]. Here we use the term "component" in a general sense to denote a *combinator* [12]. A CLS repository is modeled as a finite combinatory type environment Γ containing type assumptions $X: \tau$, where X is a combinator symbol and τ is its implementation type. Following standard practice in type theory, we write $\tau \to \sigma$ to denote the type of a component with input type τ and the result type σ. Since types are used as specifications for synthesis problem in CLS, we need to express semantic concepts *at the type level*, so that types can be used to specify the meaning and purpose of combinators. We extend implementation types with the *intersection type* operator [18], denoted $\cap$. Intuitively, a statement of the form $X: \tau \cap \sigma$ means that X has both type τ and type σ.

Temp	$: Sensor \to real$	*Extracts temperature measurement from a sensor*
F2C	$: real \to real$	*Converts Fahrenheit to Celsius*
Sens	$: Sensor$	*Denotes a sensor*

Figure 2: Simple Sensor Example

To illustrate these ideas by a simple example, consider the repository Γ with typed combinators as shown in Figure 2. The combinators shown (**Temp, F2C, Sens**) name implementations written in an *implementation language* of choice (for example, Java), which we refer to as L1. The types associated with the combinators are types of the corresponding implementations (the types are trivially rewritten from the implementation language types using our type-theoretical notation). With these implementation types, it is impossible to specify the goal of synthesizing an expression that computes a sensor's temperature in Celsius. But we can enrich the specifications with semantic concepts using the intersection operator as shown in Figure 3.

The semantic types F^0 and C^0 denote the units Fahrenheit and Celsius, Ms indicates the result of a measurement, and $Conv$ denotes a unit conversion function. To identify an expression of type $real \cap C^0$ from these elements in Γ (that is, apply combinators to arguments of the right types), the synthesized L1 expression is **F2C** (**Temp Sens**).

Temp	$: Sensor \rightarrow (real \cap F^0 \cap Ms)$
F2C	$: ((real \cap F^0) \rightarrow (real \cap C^0)) \cap Conv$
Sens	$: Sensor$

Figure 3: Enhanced Sensor Example with Intersection Types

This simple example demonstrates how CLS can formalize and automate the process of computing type-correct combinator expressions (applicative combinators of combinator symbols) from a given set Γ of typed combinators. The logical foundation of this idea is to consider the *inhabitation* relation in combinatory logic: Given an environment Γ and a type τ, does there exist a combinatory expression e such that $\Gamma \vdash e: \tau$, that is, an expression e with type τ in the context of type assumptions Γ? An algorithm that solves inhabitation problems can be used to compute (or enumerate) such expressions e, referred to as *inhabitants* of the type τ.

To increase the flexibility of CLS, staged composition synthesis (SCS) [19] introduces a functional meta-language, L2, in which component implementation code (referred to as L1) can be manipulated. The meta-language is essentially the $\lambda_e^{\Box\rightarrow} -$ calculus of Davies and Pfenning [20], which introduces a modal type operator $\Box$ to inject L-1-types into the type-language of L2. Intuitively, a type $\Box\tau$ describes a fragment of L1 code that is of L1-type τ. The repository contains *composition components* with implementations in L2. Synthesis automatically composes both L1- and L2-components, resulting in more flexible and powerful forms of composition since complex (and usually context-specific) L1-code-manipulations, including substitutions of code into L1-templates, are cleanly encapsulated in composition components.

It is a nice consequence of the operational semantic theory of $\lambda_e^{\Box\rightarrow}$ that computation can be *staged*. For a composition e of type $\Box\tau$, it is guaranteed that all L2-operations can be computed away in a first *composition time* stage, leaving a well-typed L1-program (implementation type correctness) of type τ to be executed in a following *runtime* stage. SCS extends λ-calculus to include boxing of L1-code, box[L1-code]. A dual operation **letbox var={meta-code}** in **{·}** binds the unboxed code to a variable **var** which can then be used to manipulate the synthesized L1-program. To illustrate SCS, we extend our example L1-repository Γ with the **ConApp L2**-combinator shown in Figure 4.

$$\textbf{ConApp: } \Box((\alpha \to \beta) \cap Conv) \to \Box(\alpha \cap Ms) \to \Box(\beta \cap Ms)$$

Figure 4: Adding ConApp combinator to sensor example

This combinator expresses the new idea that a unit conversion ($Conv$) preserves the measurement property (i.e., it can be applied to a measurement to yield a measurement). **ConApp** has the L2-implementation shown in Figure 5.

```
λf: □((α → β) ∩ Conv).
        λx: □(α ∩ Ms).
                letbox F = { f } in {
                   letbox X = { x } in { box[ F(X) ] } }
```

Figure 5: Implementation of ConApp in Sensor Example

The combinator is polymorphic because of the α and β type variables. Thus, if we ask for inhabitation of the type $real \cap C^0 \cap Ms$ (insisting that we get the result of a measurement), the solution is **ConApp F2C (Temp Sens)**, which reduces in L2 to the expression box[**F2C(Temp Sens)**], a boxed piece of well-typed L1-code we showed earlier.

This example illustrates a fundamental idea in SCS, namely that an L2-combinator is a *higher-order polymorphic function acting on L1-arguments*, producing L1-code; this can be done *even though L1 might be a first-order monomorphic language* (see [19] for more details). Thus, introducing L2 leads to powerful functional and type-theoretic abstractions that can be made to interact with a quite different implementation language, L1.

A (CL)S tool [21] implements SCS and is used as the platform for our experiments. Despite exponential worst-case complexity of CLS/SCS, the *average* synthesis time in our experiments is less than 5 seconds on a standard Windows Desktop PC because of heuristic optimizations included in (CL)S [12]. Table 1 lists the mathematical operators and their corresponding (CL)S operator representation used in the (CL)S input grammar [22].

Table 1: Mathematical operators and corresponding expressions in (CL)S

	Mathematical example	(CL)S representation example
Atoms	τ, σ	tau, sigma, τ, σ
Variables	α, β	alpha, beta, α, β
→	$\tau \to \sigma$	tau −> sigma *or* $\tau \to \sigma$
∩	$\tau \cap \sigma$	[tau, sigma] *or* $[\tau, \sigma]$
Covariant constructor	$C(\tau_1, \ldots, \tau_n)$	C(tau1, … , taun)
≤	$\tau \leq \sigma$	tau <= sigma *or* $\tau \leq \sigma$
Subst.	$S(\alpha) = \tau$	$\{\alpha\} => \{\tau\}$ $\{\alpha\} \sim > \{\tau\}$

2.2. Inhabitation Problem

Inhabitation problem is in the core of our approach to synthesizing code from ready-to-use software components, which are called combinators. Anytime we synthesize code, it's the inhabitation problem that gets solved. In this section we explain in more detail what it is and give some examples that illustrate the nature of this problem.

Recall that the logical foundation behind how (CL)S can formalize and automate the process of computing type-correct combinator expressions from a given set of typed combinators Γ is to consider the *inhabitation* relation in combinatory logic [17]. This can be explained in the following manner:

Consider an environment Γ and a type τ, the question is does there exist a combinatory expression e such that $\Gamma \vdash e{:}\,\tau$ (i.e. an expression e with type τ in the context of type assumptions Γ)?

An algorithm that solves inhabitation problems can compute expressions *e* referred to as *inhabitants* of the type τ. The repository represented by Γ may change, hence there is a need for a generalized form of combinatory inhabitation known as *relativized inhabitation*. Under this form, the environment Γ is not constant as opposed to standard combinatory logic that considers a fixed base. This relativized inhabitation is versatile and can be used to define a Turing-complete notion of computation even in simple types or propositions.

While the general inhabitation problem is undecidable, in practice the (CL)S tool uses heuristics to synthesize code. Optimization techniques for (CL)S are provided in [22]; it discusses two independent families of approaches to optimizing the relativized inhabitation problem. The first approach, addresses the optimization of the theoretical algorithm; and the second one, proposes a distributed implementation of the relativized inhabitation algorithm.

Depending on the components comprising the Γ repository and the query expression (i.e., the inhabitant), there are cases when there exists only one inhabitant that satisfies the conditions; at other times, there will be many solutions to the inhabitation problem. In the latter case, the search tree expands to finding all the inhabitants, each of which leads to different new synthesized code that is part of the final solution. To better illustrate this situation, we describe two examples in which the inhabitation algorithm finds one single inhabitant (first example) and multiple inhabitants (second example).

Example 1

Given a Γ repository with the combinators shown in Figure 6. Note, we show only the definition of the combinators, as the implementation does not affect the inhabitation problem at all. The definition of a combinator includes its name and intersection type, the implementation could be anything, Java code, C#, C++, configuration, properties etc.

```
Γ = {
        type ~> A
        NameRule : #[A, namerule]
        Moves :    #[A, moves]
        Game :     #[namerule, alpha.type] ->
                   #[moves, alpha.type] ->
                   #[game, alpha.type]
}
```

Figure 6: Repository with Single Inhabitant Example

Now we can ask inhabitation questions of the form $\Gamma \vdash ?: \tau$ using this repository and target intersection type τ. If this type is not computable from the combinators, then the resulting answer is "No Solution".

We can ask the question $\Gamma \vdash ?: \#[b]$, or whether any inhabitant can be found of type *b* or have a subtype of *b*. The algorithm will process the Γ repository, checking if there's any combinator satisfying the rule, namely having the type 'b' or any subtype of 'b', and report that there are no inhabitants that satisfy the rule.

Another question to ask is $\Gamma \vdash ?: \#[A, namerule]$. In this case there is a combinator with the intersection type [*A, namerule*] and it is the *NameRule* combinator, therefore the answer will be *Namerule.* Figure 7 illustrates this scenario. The next step is to replace the *NameRule* combinator with its implementation, but we won't discuss it further here, since the implementation is not relevant for the inhabitation problem, and replacing the combinator with its corresponding implementation is a straightforward process.

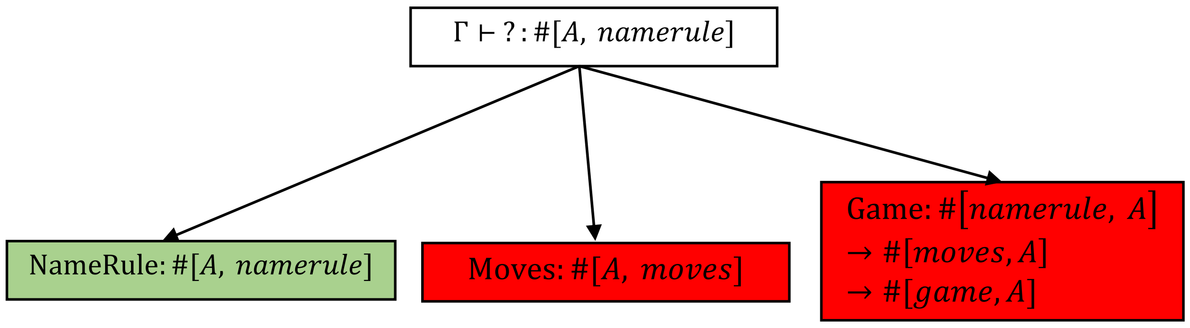


Figure 7: Search tree with single inhabitant

Red colored leaves indicate an unsuccessful search while green one (the lighter one) indicates success.

Let's see what happens when we ask the question $\Gamma \vdash ? : \#[A, game]$. Initially, only the *Game* combinator satisfies the rule, but this combinator will force to evaluate the other two combinators, *Moves* and *NameRule*, since it expects two parameters whose intersection types match with *Moves* and *NameRule*. Figure 8 shows the expansion of the search tree for this scenario.

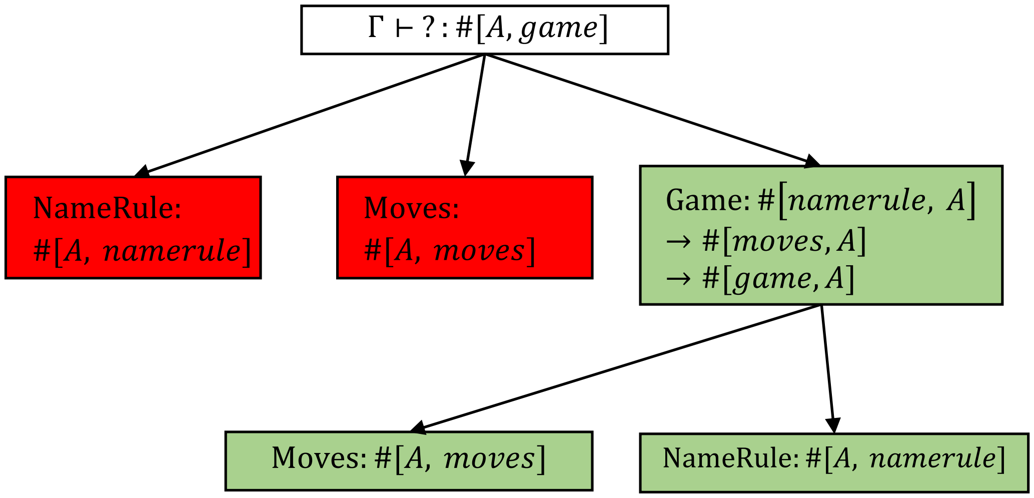

$\Gamma \vdash ? : \#[A, game]$
NameRule: $\#[A,\ namerule]$
Moves: $\#[A,\ moves]$
Game: $\#[namerule,\ A] \to \#[moves, A] \to \#[game, A]$
Moves: $\#[A,\ moves]$
NameRule: $\#[A,\ namerule]$

Figure 8: Search tree with expanded single inhabitant

The successful path, in the tree above, is marked with green (lighter color, in monochrome printouts). The answer in this case is the composition of combinators *NameRule* -> *Moves* -> *Game*.

```
Γ = {
        type ~> A

        NameRule : #[A, namerule]

        Move1 :    #[A, move_1]

        Move2 :    #[A, move_2]

        Moves :    [ #[A, move_1] -> #[A, moves],
                     #[A, move_2] -> #[A, moves]]

        Game :     #[namerule, alpha.type] ->
                   #[moves, alpha.type] ->
                   #[game, alpha.type]
}
```

Figure 9: Multiple Inhabitant Repository

Example 2

In this example, shown in Figure 9, we enrich the Γ repository with two more combinators and modify the *Moves* combinator so that it is defined as a function table.

With the same query $\Gamma \vdash ?: \#[A, game]$ only the *Game* combinator satisfies the rule. In the process of evaluating the *Game* combinator, the algorithm finds *NameRule* and *Moves*. Since the *Moves* combinator is a function table, which contains two combinators of the intersection type #[A, moves], it causes the search tree to branch into two successful leaves, consequently giving two solutions.

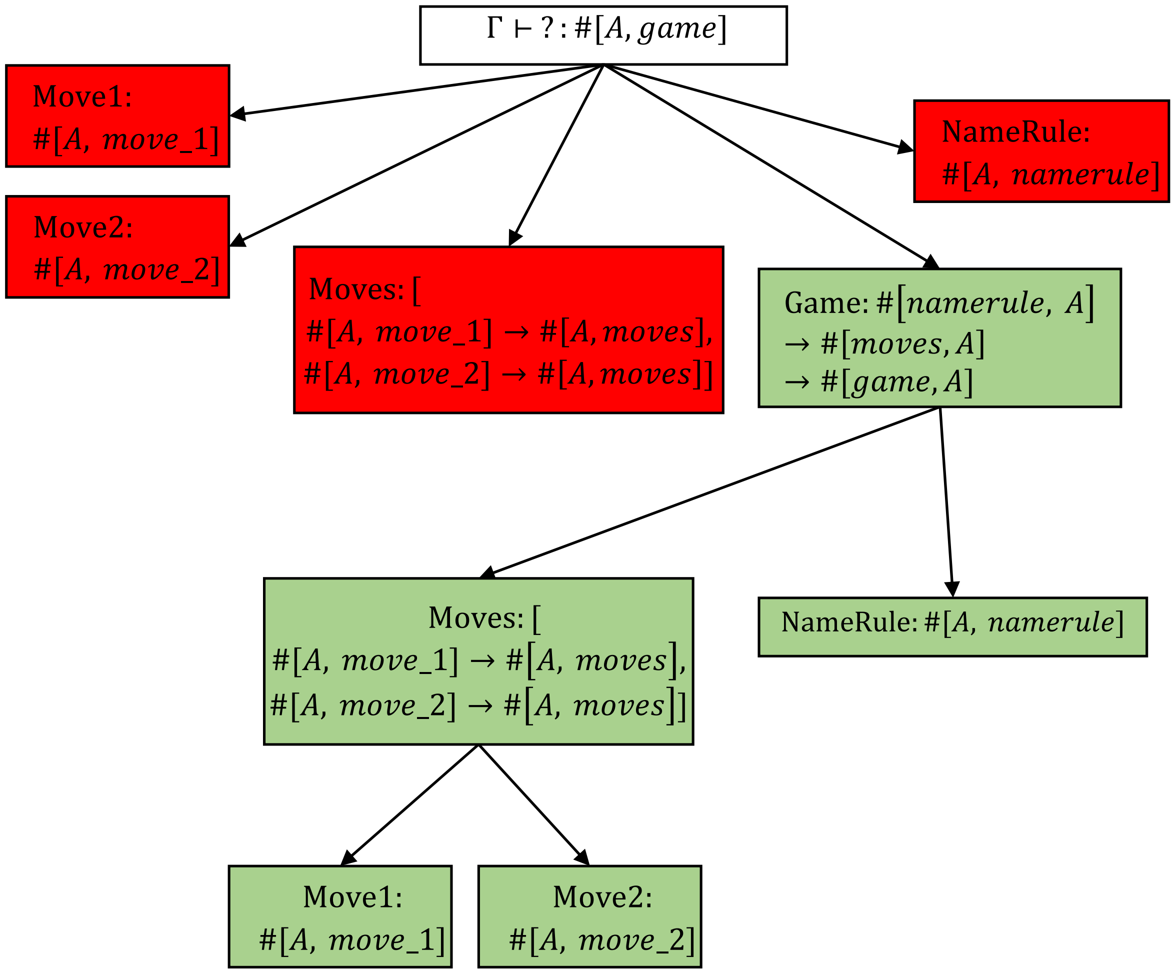

$\Gamma \vdash ?: \#[A, game]$
Move1: $\#[A,\ move\_1]$
Move2: $\#[A,\ move\_2]$
Moves: [$\#[A,\ move\_1] \to \#[A, moves]$, $\#[A,\ move\_2] \to \#[A, moves]$]
NameRule: $\#[A,\ namerule]$
Game: $\#[namerule,\ A] \to \#[moves, A] \to \#[game, A]$
Moves: [$\#[A,\ move\_1] \to \#[A,\ moves]$, $\#[A,\ move\_2] \to \#[A,\ moves]$]
NameRule: $\#[A,\ namerule]$
Move1: $\#[A,\ move\_1]$
Move2: $\#[A,\ move\_2]$

Figure 10: Search tree 3

Each successful node contains unsuccessful leaves, but we are not showing them for the sake of brevity. In this case we have two solutions: *NameRule* -> *Move1* -> *Game* and *NameRule* -> *Move2* -> *Game*.

2.3. Tool Support InhabConsoleClient

In our experiments with the KombatSolitaire and MOEA frameworks, as well as other examples shown in this thesis, we used the tool implemented at the Technical University of Dortmund, Germany (for details see [21]). Details about the implemented inhabitation algorithm, employed heuristic optimizations and other theoretical and technical details are discussed in Boris Düdder's dissertation [22]. Figure 11 shows a very high level design of the tool. The "Types" and "Impls" boxes represent the definition and implementation of combinators, respectively.

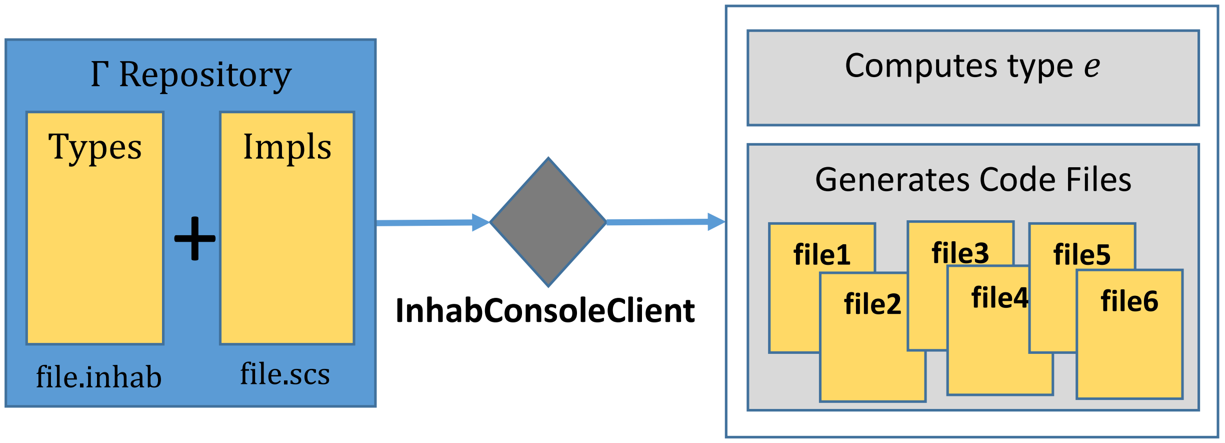
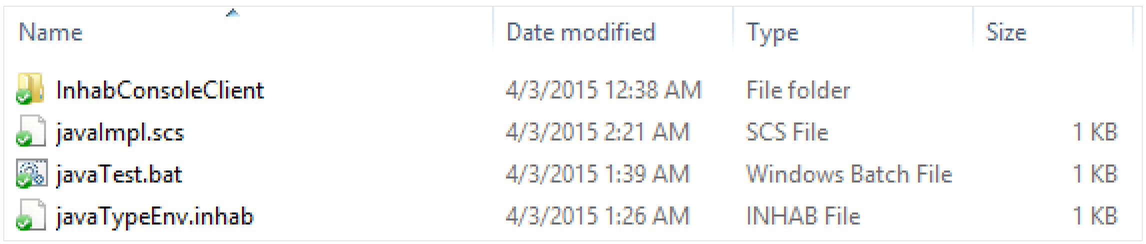


Figure 11: InhabConsoleClient - High level design

This tool, **InhabConsoleClient** or **CLS tool** as we refer to it sometimes, provides the functionality of solving the inhabitation problem given a Γ repository and an intersection type (the goal) as discussed in the previous sections. In this section we will show how to use this tool and some special "hacks" needed to serve our purpose of synthesizing executable code.

InhabConsoleClient is provided as an executable file, which requires two input files to run – an *inhab* file that contains the combinators' definitions and the inhabitation question, and an *implementation* file that contains the combinators' implementation. We now show a simple example which synthesizes a Java class for converting temperature values. This example simply shows how to use the tool. Create three files, as in Figure 12.



Name	Date modified	Type	Size
InhabConsoleClient	4/3/2015 12:38 AM	File folder	
javaImpl.scs	4/3/2015 2:21 AM	SCS File	1 KB
javaTest.bat	4/3/2015 1:39 AM	Windows Batch File	1 KB
javaTypeEnv.inhab	4/3/2015 1:26 AM	INHAB File	1 KB

Figure 12: Running the InhabConsoleClient

The folder **InhapConsoleClient** contains the tool that we need to run, more exactly the file named **InhabConsoleClient.exe**. The file **javaTypeEnv.inhab** contains the definition of the combinators, **javaImpl.scs** contains the implementation code for the combinators, and **javaTest.bat** contains the script to run the tool with the *inhab* and *scs* files as input parameters. The contents of these three files is given below.

javaTest.bat

```
InhabConsoleClient\InhabConsoleClient.exe javaTypeEnv.inhab javaImpl.scs
```

javaTypeEnv.inhab

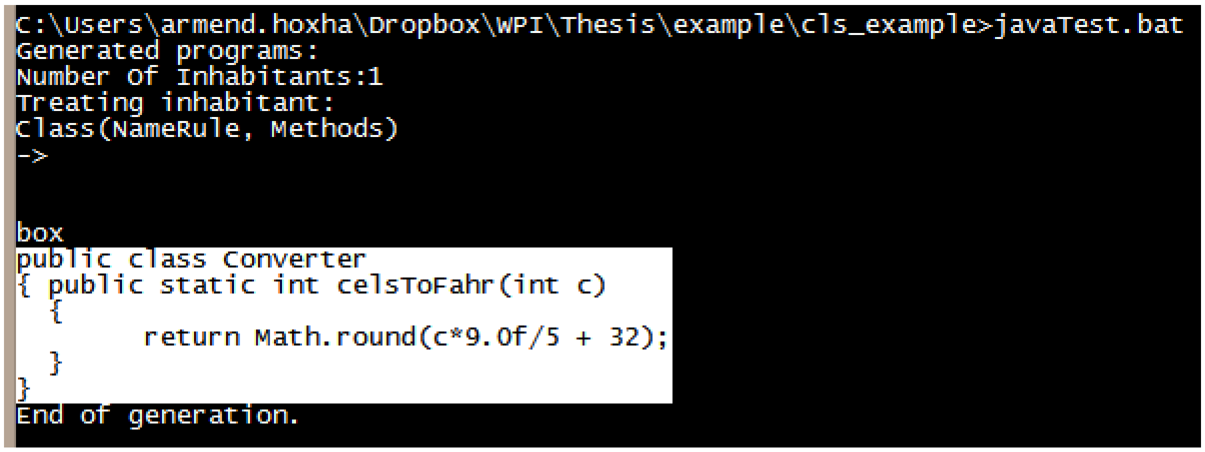
```
kinds
type ~> c

combinatorsD
NameRule : #[c, namerule]

Methods : #[c, method]

Class :  #[namerule, alpha.type] ->
         #[method, alpha.type] ->
         #[class, alpha.type]

|- ? : #[c, class] //inhabitation question
```

javaImpl.scs

```
declarations

NameRule : { box["Converter"] }

Methods : { box["public static int celsToFahr(int c)
  {
     Math.round(c*9.0f/5 + 32);
  }"] }

Class : {
      λname.   {letbox NameParameter = {name}    in {
      λmethod. {letbox Methods       = {method}  in {
box[ "
public class " NameParameter "
{ " Methods "
}"]
}}}}
}
inhabitant
```

Note, the *Class* combinator is defined to have two parameters: [*namerule*, *alpha.type*] and [*method*, *alpha.type*]. The order of parameters matters, therefore the parameters in the implementation should match the order in the definition. In the case of the *Class* combinator, the first parameter named *NameParameter* will be whatever combinator matches the intersection type [*namerule*, *alpha.type*], as specified in the definition. If there are two or more combinators of the same intersection type, only one will be returned and the selection is non-deterministic. The same rule applies to other parameters, too.

Now we can execute **javaTest.bat** from the command line and see the result, as shown in the screenshot below.



```
C:\Users\armend.hoxha\Dropbox\WPI\Thesis\example\cls_example>javaTest.bat
Generated programs:
Number Of Inhabitants:1
Treating inhabitant:
Class(NameRule, Methods)
->


box
public class Converter
{ public static int celsToFahr(int c)
  {
        return Math.round(c*9.0f/5 + 32);
  }
}
End of generation.
```

The highlighted code is the solution that was found in the repository. As we can see, the solution is printed out on the console, which is not the place where we would want our code to be placed on. To modify the implementation of the combinatory so its generated code is saved to a file, change the *Class* combinator implementation in the **javaImpl.scs** file, so that the code inside the box[] is preceded with **"=======" Name of the file "======="**. The *Class* combinator now will look like this (the added code is colored red).

```
Class : {
      λname.   {letbox NameParameter = {name}    in {
      λmethod. {letbox Methods       = {method}  in {
box["=======" NameParameter ".java=======
public class " NameParameter "
{ " Methods "
}"]
}}}}
}
```

The number of equality signs (=) should be exactly 7 before and 7 after the name of the file. Now if we run the **javaTest.bat** script again, it will print the result on the console and also dump it into a file named *Converter.java*, as determined by the *NameRule* combinator.

In this example we are saving, into a file, only the final step of the solution. It is possible to save into different files the intermediate steps of the solution, too. Suppose we want to create a class to test our *Converter* class, basically, create a class that has a `main(…)` method, which tests the `celsToFahr(int c)` method. Add a *Program* combinator that causes the evaluation of the *Class* combinator, and creates the class with the `main(…)` method described above. We will modify the **javaTypeEnv.inhab** file as shown below.

javaTypeEnv.inhab

```
Kinds
type ~> c

combinatorsD
NameRule : #[c, namerule]

Methods : #[c, method]

Class :  #[namerule, alpha.type] ->
         #[method, alpha.type] ->
         #[class, alpha.type]

Program: #[c, class] ->
         #[c, namerule] ->
         #[c, program]

|- ? : #[c, program] //inhabitation question
```

Figure 13: A sample of combinator definitions in CLS

Note that, besides adding the *Program* combinator, we have also modified the inhabitation question; now we are asking for a solution of the intersection type [*c*, *program*], before we asked for [*c*, *class*]. Since *Program* expects a [*c*, *class*] combinator, it will cause the *Class* combinator to be evaluated, thus creating the file named *Converter.java*. We need to add an implementation for the *Program* combinator to the **javaImpl.scs** file. The code snippet below shows how we implement the *Program* combinator. Recall from Section 2.1 that the implementation of a combinator, which is a blend of L2 and L1 code, is placed inside a **box,** enclosed in double quotes (e.g. **box[**"implementation"**]**).

```
Program : {
   λclass.  {letbox ClassParam = {class}   in {
   λname.   {letbox NameParam  = {name}    in {
   box[ "=======Test" NameParam ".java=======
    public class Test" NameParam " {
        private int temp = 20;
        public int getTemp()
        { return temp; }
        "
       ClassParam ″
       =======+Test" NameParam ".java=======
      public static void main(String[] args)
      {
         Test" NameParam " tn = new Test" NameParam "();
         System.out.println(" NameParam ".celsToFahr(tn.getTemp()));
      }
17.   }"]}}}}
18. }
```

Figure 14: Implementation of the Program combinator

Line 4 (ignore the string “**box[**“) is a way (hack) to tell the tool to save the synthesized output of this combinator to a file named *TestConverter.java* (since *NameParam* is bound to “Converter”, in this case). In line 10 we are using the *ClassParam* combinator, which in this case won’t be replaced by its implementation, but rather serve as a binding with the parameter, causing the first parameter of *Program* to be evaluated, in this case the *Class* combinator. In line 11, the second synthesized part of *Program* is appended to the *TestConverter.java* because of the “=======+” notation. If we don’t add the code in line 11, the `main(…)` method will wind up being in *Converter.java* rather than *TestConverter.java*.

2.4. Tool Support LaunchPad Eclipse Plugin

FeatureIDE [23] is an open-source framework for feature-oriented software development (FOSD) based on Eclipse. It successfully integrates a number of composition tools through a well-documented extensible interface. We wanted to integrate the `InhabConsoleClient` tool and make it easy for developers to use and write their own combinators. The first decision was to design a text-based representation for combinators that would be easier for programmers to use by eliminating the λ syntax that appears in the standard representations of combinators.

Figure 15 contains the LaunchPad [24] equivalent of some of the combinators from Figure 13 and Figure 14. These combinators appear in files that are associated with the individual features (see Section 2.5 for details). The syntax was designed to more closely resemble modern programming languages. Each intersection type $A \cap B$ is written textually as **[A, B]** where the order of these strings is irrelevant. A combinator is defined by its type and its implementation. Using the common concepts found in OO, one can design an abstract combinator by only defining its type specification in a combinator file; alternatively, it is possible to override the implementation of an existing combinator by providing an implementation of an existing combinator.

In the implementation of the **Program** combinator (Figure 15) note how the L1-embedded Java code has “<NameParameter>” which will ultimately be replaced by the L1-code associated with the L2-variable NameParameter, which in this case is the constant “Converter”. These files are edited within a *LaunchPad* editor that properly formats the L1-embedded Java code, making it easier for programmers to read.

Upon selecting a configuration file in FeatureIDE (see Section 2.5 for details), the LaunchPad plugin composes the associated combinator files and constructs the necessary λ combinator specification files which it passes to the `InhabConsoleClient` tool to synthesize the final code. The code is generated within the **src/** source folder in an Eclipse project, which allows programmers to easily include the generated code into their own projects by simply linking to the FeatureIDE project in which the generated code exists.

```
define {
        [c, namerule]               NameRule -> Converter;
}

type Program {
        ClassParam                  [alpha.type, class];
        NameParameter               [alpha.type, namerule];
                                    [alhpa.type, program];
}
implementation Program (converter/Test<NameParameter>.java) {
package converter;
public class Test<NameParameter> {
        private int temp = 20;
        public int getTemp();
        {return temp;}
        <ClassParam>
        public static void main(String[] args){
                Test<NameParam> tn = new Test<NameParam>();
                System.out.println("<NameParam>".celsToFahr(tn.getTemp()));
        }
}}
```

Figure 15: Equivalent LaunchPad Combinators

LaunchPad [12] is implemented by three Eclipse plugins, one to handle editing files, one to integrate the InhabConsoleClient executable [21], and one to install the KS plugin example.

LaunchPad can be retrieved from its update site, http://combinators.org/launchpad/update-site; naturally, one must first install FeatureIDE. LaunchPad is implemented in about 7,500LOC.

2.5. Tool Support FeatureIDE

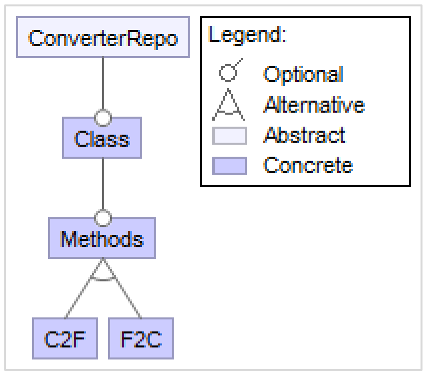
In this section we briefly explain FeatureIDE and give references for further details, a helper tool that we have used to develop the Γ repository for the two frameworks within which we have conducted our case studies.

Computer programs that share many features are grouped together to form the so called program families, which are also known as product lines, and the idea is to build not individual programs but the family of similar programs [25]. An architectural model and a design methodology that implements the idea of program families is the AHEAD model (for details see [15]). FeatureIDE is an open source tool provided as an Eclipse plugin, which supports the development of program families following the AHEAD architecture model [25].

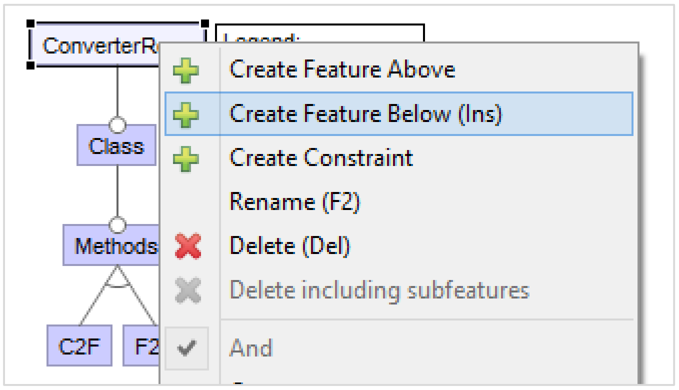
FeatureIDE supports *Feature Oriented Software Development (FOSD)*, which is a paradigm for designing and implementing applications based on features [23]. A feature represents a characteristic in a software system, and FOSD enables modularizing a software into units which represent features (see [23] for details). To do this, FeatureIDE supports an extensible interface allowing for different composition tools to synthesize or generate code based upon a selected configuration of features.

We will refer to the same example used in Section 2.3. Here we develop the repository using the FeatureIDE rather than pure L2 language. For details on how to install the FeatureIDE plugin and get started see [26].

In Eclipse, each FeatureIDE project has a *model* that defines features of a program family. Figure 16(a) shows the model for the Converter repository described in section 2.3. We can add features above or below the selected feature by selecting a feature, right-clicking on it and choosing the desired operation from the menu shown in Figure 16(b). Each rectangle represents a feature, the legend on the right-hand side of Figure 16(a) describes the meaning of the symbols and shapes in the model.



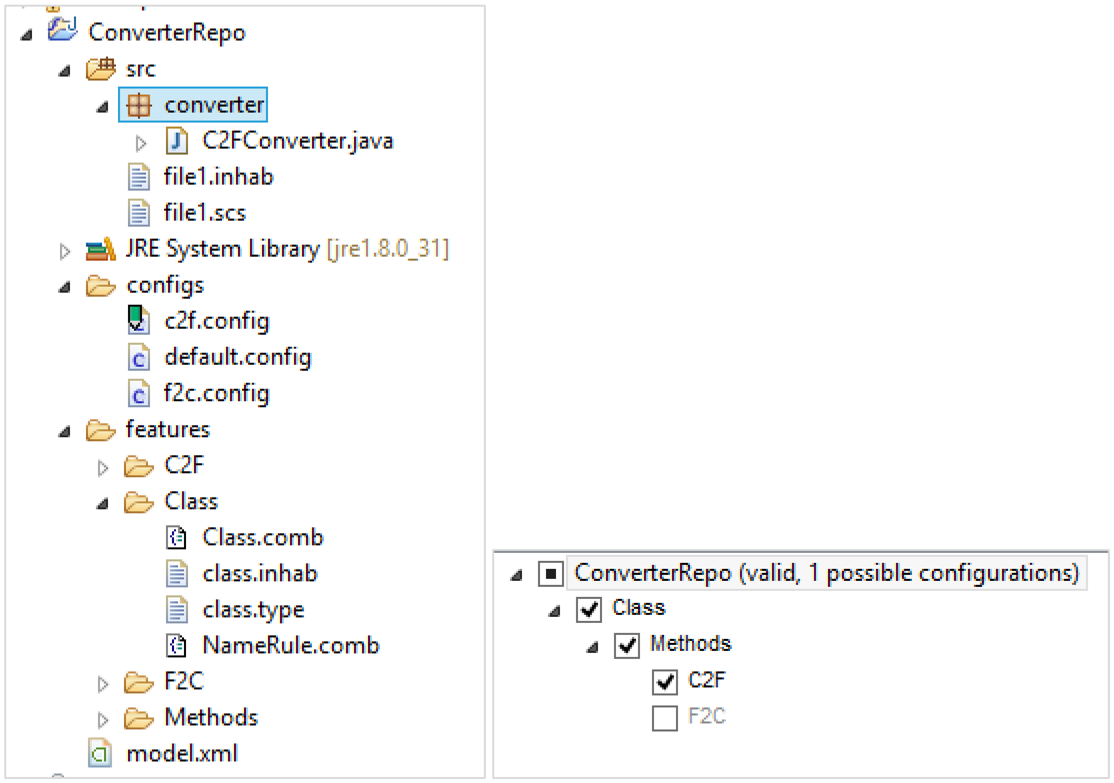
(a)



(b)

Figure 16: The model of the Converter Repository

For each feature there is a corresponding folder in which we can put the combinators associated with that feature. Apart from the model, we have the configuration files, which, unlike the model, can be more than one. In the configuration file we can choose which features we want for our program. In case of multiple configuration files, we can right click on one file and choose the "Set as current configuration" option to run the inhabitation problem for the chosen features – this operation should generate the desired program by synthesizing code from the chosen combinators and put it in the *src* folder.



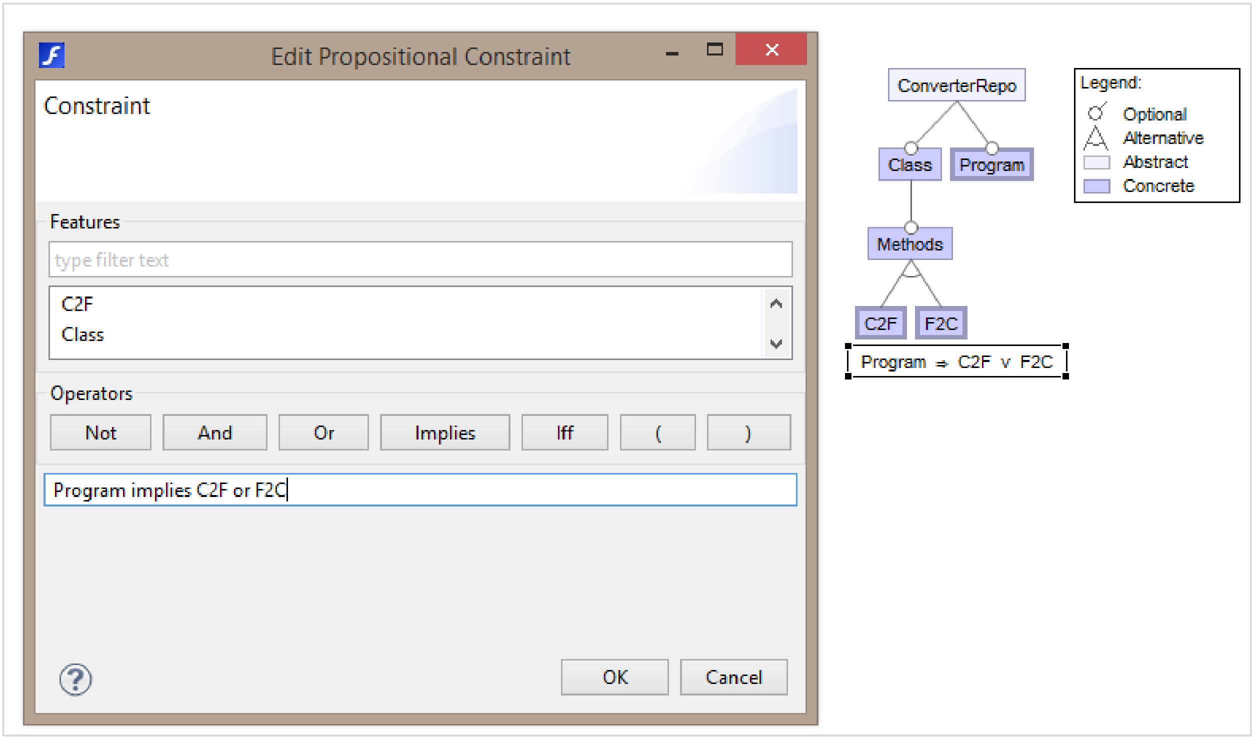
```
ConverterRepo
  src
    converter
      C2FConverter.java
      file1.inhab
      file1.scs
  JRE System Library [jre1.8.0_31]
  configs
    c2f.config
    default.config
    f2c.config
  features
    C2F
    Class
      Class.comb
      class.inhab
      class.type
      NameRule.comb
    F2C
    Methods
  model.xml
```

```
ConverterRepo (valid, 1 possible configurations)
  [✓] Class
    [✓] Methods
      [✓] C2F
      [ ] F2C
```

Figure 17: Configuration file (right) and the folder structure of the Converter Repository (left)

In this model we have decided that we can choose either C2F feature or F2C, but not both (see Figure 16(a)), and it is imposed on the configuration file, therefore we cannot have a configuration with both features selected. This is just a design decision for this model, which could have very easily been different.

It is possible to add other constraints in the model. Consider the case when we want to execute any of the converters. So we add a new feature named **Program**, for which we create a combinator that creates a class with a `main` method, inside `main` it creates a `Converter` object and calls its conversion method. We won't go into implementation details, as they are not relevant to the topic we are talking about in this section, but we will rather explain how we can add constraints on feature selection. We can make sure that no one will ever be able to create a **Program** without choosing one of the methods **F2C** or **C2F.** We do so, by adding a constraint in the model, which basically says "If you select **Program**, then you must select either **F2C** or **C2F**" or in a formal manner **Program** $\Rightarrow$ **C2F** $\vee$ **F2C**.


Edit Propositional Constraint

Constraint

Features

type filter text

C2F
Class

Operators

Not | And | Or | Implies | Iff | (|)

Program implies C2F or F2C

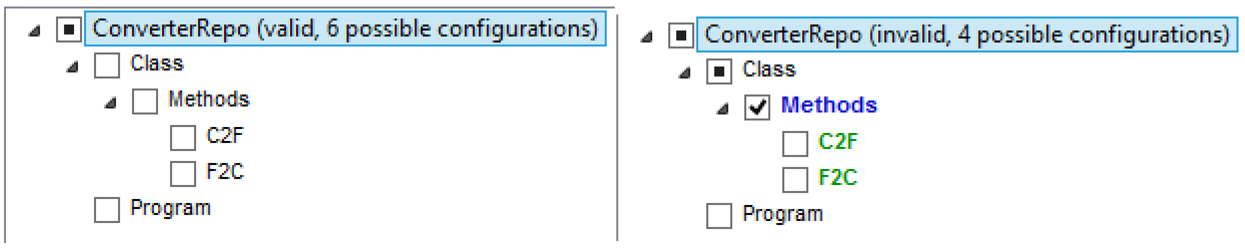
OK | Cancel

Figure 18: Constraints on feature selection

Observe that the other way around is not constrained, which means we can select F2C or C2F without selecting the Program feature.

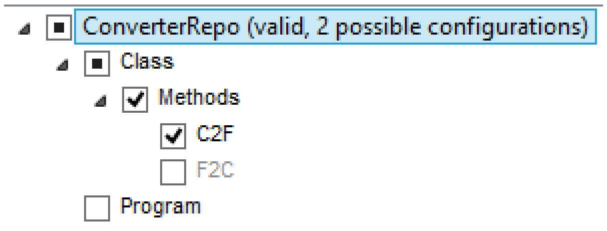
As seen in Figure 16, features are organized in a hierarchical model, consequently the parent-child relationship is implied, which means, if you select C2F you have already selected *Methods* and *Class*.

FeatureIDE counts the number of possible valid configurations for a particular selection of features in a config file. If we deselect all the features, it will show the total possible valid configurations; after selecting some features, it will show the number of possible valid configurations left to be selected.


ConverterRepo (valid, 6 possible configurations)
- Class
  - Methods
    - C2F
    - F2C
- Program

ConverterRepo (invalid, 4 possible configurations)
- Class
  - ✓ Methods
    - C2F
    - F2C
- Program

a) b)



c)

Figure 19: Possible configurations: a) total, b) possible configs after the current selection of features (invalid config), c) possible configurations (valid config)

The number of possible configurations depends on the constraints in the model, hence we call it the valid number of possible configurations. Figure 19(b) shows an invalid configuration; it is invalid because either C2F or F2C must be selected after Methods has been selected and in this configuration none of them is selected.

We end this chapter by emphasizing the importance of these tools in conducting our research. As we'll see later in the following chapters, Γ repositories can grow really big over time and contain hundreds and maybe thousands of combinators. FeatureIDE makes it possible to manage such a large number of combinators. The standard representation of combinators uses the λ syntax, which is not easy to write and makes it even more difficult to read; besides, it also requires everything to be put into two files (definition and implementation files), which is very impractical to deal with when the number of combinators exceeds 50. LaunchPad provides a macro language, which looks like a modern programming language and is much easier to read and write. It also allows the programmer to break the code into many combinators (files), which gives an advantage as opposed to having all in just two files. And finally, the InhabConsoleClient tool provides the implementation of the CL(S), which provides the algorithm for solving the inhabitation problem in our repository of combinators.

3. Case Study

We have two case studies, conducted in two different object oriented frameworks. The first one was conducted within an OO framework called KombatSolitaire (KS), a framework that had earlier been developed over a number of years by Prof. George Heineman as part of an undergraduate course in software engineering. KS is a Java framework that enables head-to-head competition of solitaire variations played simultaneously over the Internet. First we implemented a solitaire variation called FourteenOut in native Java code, then we implemented the same variation in Lambda/Combinator language called L2.

One of the goals of this thesis, is to find out whether the combinatory logic approach to software development can be exploited with any OO framework or not. Therefore, we picked a third-party framework, which would take this approach one step ahead. The second case study is conducted within an open source framework hosted on www.sourceforge.com. It is called MOEA (Multi-Objective Evolutionary Algorithms) framework; it offers many ready-to-use algorithms for solving multi objective optimization problems, and provides the interface for encoding any optimization problem, which then can be solved using the algorithms provided by the framework. (For details see the **MOEA Framework** section).

3.1. FourteenOut Experience

FourteenOut is a variation of the solitaire game, in which the player can take two cards out if the sum of their ranks equals 14. This is called a legal move. The move is done by first selecting one card and then another one. After the second card has been selected, the ranks of the selected cards are checked if they add up to 14; if yes, the cards are taken out; if not, the selected cards are deselected.

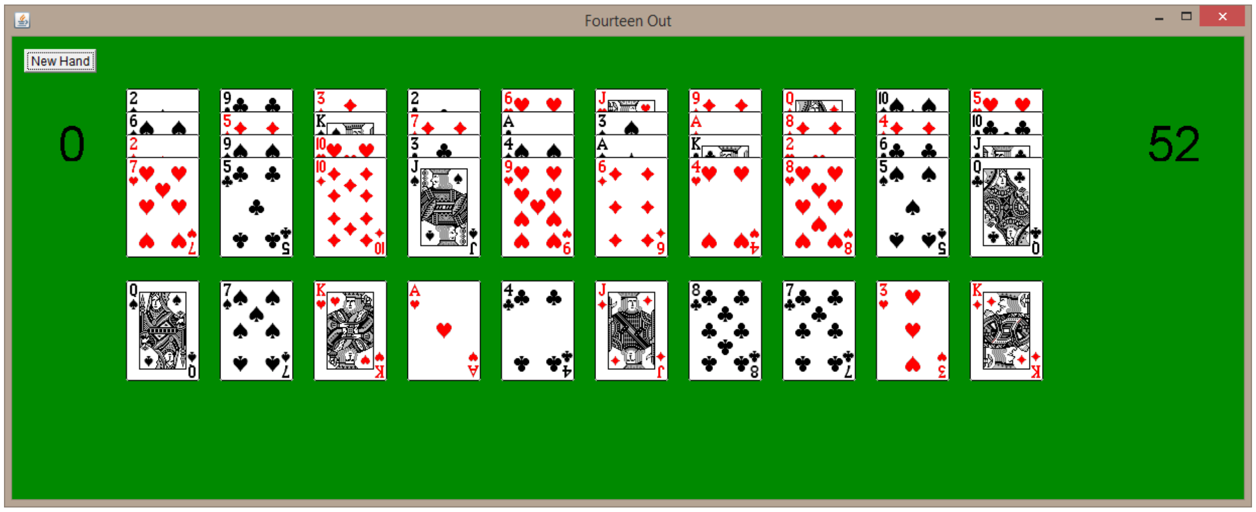


Figure 20: FourteenOut - the first row represents columns, the second one represents piles.

All 52 cards are laid out in an equal number of piles and columns. A pile contains cards that are piled up and the player can see and select only the top most card. A column contains cards that are arranged in such a way that the player can see all the cards, but select only the topmost one. Figure 20 shows what the game layout looks like.

3.1.1. Native Java implementation

In order to write my own variation of solitaire, namely FourteenOut, by taking advantage of the framework - not writing everything from scratch, I needed to understand the design of this framework, so that I could extend it and produce the desired flavor of the game.

Most of the functions you need for a solitaire variation game, are provided by the framework; all you have to do is provide the classes that behave the way you want your game to be, and know where to hook them up.

I used the tutorial provided by Prof. Heineman to understand the design of the framework, and find out the hot spots - points where a framework allows extension. The tutorial describes how to write the Narcotic game, which is another variation of the solitaire game.

- **Tutorial for the Narcotic variation**

The framework is designed using EBC (Entity-Boundary-Controller) pattern. The entity, boundary and controller objects share the relationship as shown in the following picture.

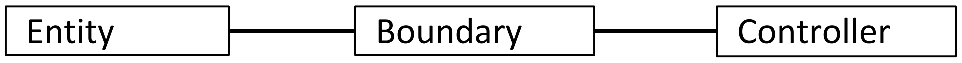


Figure 21: Design pattern of the KombatSolitaire framework

Entity objects

```
Deck deck
Pile pile1
Pile pile2
Pile pile3
Pile pile4
MutableInteger score
MutableInteger numLeft
```

Boundary objects

Boundary objects represent the entity objects visually.

```
DeckView deckView
PileView pileView1
PileView pileView2
PileView pileView3
PileView pileView4
IntegerView scoreView
IntegerView numLeftView
```

Each Boundary View widget can be associated with a MouseAdapter (responsible for handling PRESS, RELEASE and CLICK), MouseMotionAdapter (responsible for handling MOVE and DRAG), UndoAdapter (responsible for handling UNDO). The DRAG controllers are provided for you free of charge, but you still need to construct them. For each view object, you need to calculate where they will be displayed in the (x,y) plane as shown in the figure below (Figure 22). Note that (0,0) is in the upper left corner of the screen. To aid in your calculations, you can use the `CardImages` class, which provides the following static methods: `getWidth()`, `getHeight()`, and `getOverlap()`. The `getOverLap()` method returns the number of pixels by which cards should overlap themselves within a Column or Row.

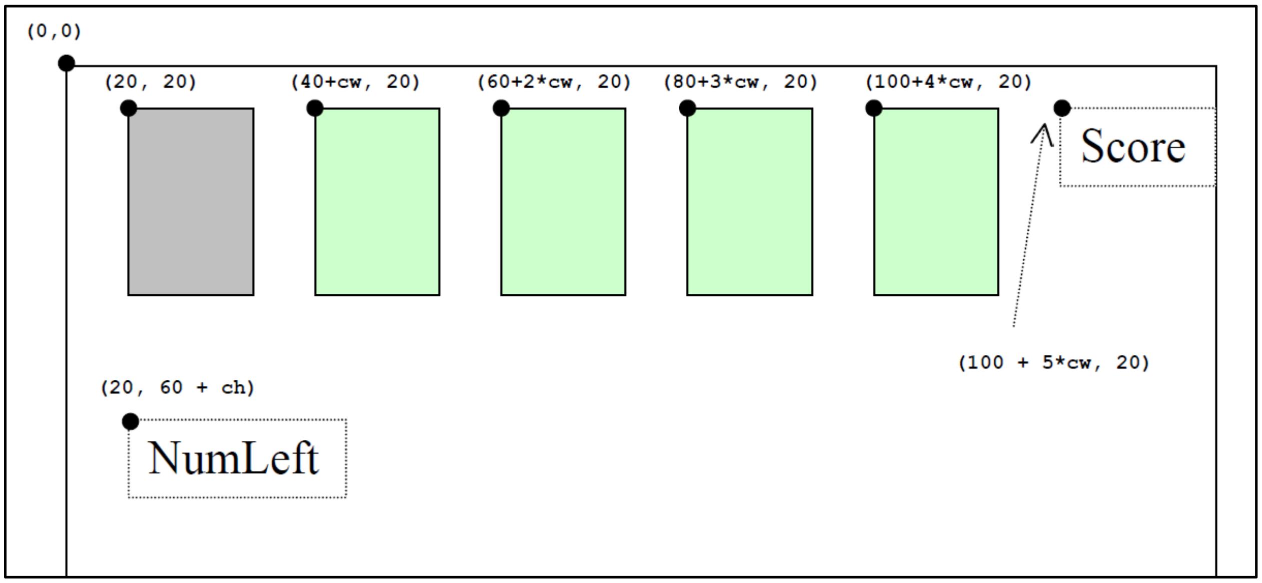


Figure 22: Narcotic Layout

Controller Objects

Controllers are used to map the mouse interaction into moves recognized by the Narcotic solitaire variation, as shown in the following table:

	Mouse Mapping
Deal	Mouse Press on DeckView
Move	Mouse Press on PileView, followed by Mouse Release on a second PileView

Reset	Mouse press on DeckView
RemoveAll	Double Click on the left-most pile

NarcoticDeckController Class

This class is responsible for processing a deal move (if the deck is not empty) and the reset move (when the deck is empty).

```
NarcoticDeckController extends MouseAdapter
--------------------------------------------------
NarcoticDeckController (Narcotic theGame)
void mousePressed (MouseEvent me)
void mouseRelease (MouseEvent me)
--------------------------------------------------
# Narcotic narcoticGame
```

The `mousePressed()` method is responsible for dealing cards; the `mouseReleased()` method will handle situations as described in the section ***NarcoticReleasedAdapter Class*** below.

NarcoticPileController Class

This class is responsible for processing requests to move cards between Piles (PRESS on a source PileView and RELEASE on target PileView), removeAll Cards (double click on the leftmost Pile).

```
NarcoticPileController extends MouseAdapter
--------------------------------------------------
NarcoticPileController (Narcotic theGame, PIleView src)
void mousePressed (MouseEvent me)
void mouseClicked (MouseEvent me)
void mouseReleased (MouseEvent me)
--------------------------------------------------
# Narcotic narcoticGame;  // point to game
# PileView sourse;  //which PIleView (if any) to match with
RELEASE
```

The `mouseClicked()`, `mousePressed()`, `mouseReleased()` methods are responsible for controlling the entities as stored within the Narcotic solitaire Plug-In.

SolitaireMouseMotionAdapter Class

This class is responsible for processing any DRAG request and is provided as is. During design, you will learn how to interact with the container so drag will work properly.

SolitaireMouseMotionAdapter **extends** MouseMotionAdapter
SolitaireMouseMotionAdapter (Solitaire theGame) void mouseDragged (MouseEvent me)
Solitaire theGame; //point to game

If you are curious as to how this method operates, look at the Solitaire source code.

SolitaireUndoAdapter Class

This class is responsible for processing any right click events to Undo requests. This class is provided for you as is. During design you will learn its interface and how to interact with it. For now (in Analysis) we will not discuss Undo further.

SolitaireUndoAdapter **extends** UndoAdapter
SolitaireUndoAdapter (Solitaire theGame) void undoRequested()
Solitaire theGame; // point to game

NarcoticReleaseAdapter Class

What if the user is executing the drag of an element (such as a card) from one Pile to another, but the mouse is not released on another Pile? The mouse Press has already recorded the source of the interaction, and therefore we must move the actively dragged card back to its original location. This controller must be associated with every view element that is visible and which you would normally have thought of as being "Passive". To ease the implementation, you will find the `returnWidget(w)` method quite useful (see the Widget class documentation).

NarcoticReleaseAdapter **extends** MouseAdapter
NarcoticPileController (Narcotic theGame) void mouseReleased (MouseEvent me)
Narcotic narcoticGame; // point to game

Game Object

This is the “super object”, the one that is the center of the solitaire plug in. This special class must extend the Solitaire class. To complete the functionality you must design several move classes.

```
NarcoticGame extends SolvableSolitaire
---
void initialize()
Enumeration availableMoves ()
---
# Deck deck
# Pile pile1, pile2, pile3, pile4
# DeckView deckView
# PileView pileView1, pileView2, pileView3, pileView4
# IntegerView numLeftView, scoreView
```

The `availableMoves()` method is only to be implemented if you will produce a plug-in that is capable of auto-play. In this case, note that the superclass of NarcoticGame is SolavbleSolitaire.

DealFourMove Class

```
DealFourMove extends Move
---
DealFourMove (Deck d, Pile p1, Pile p2, Pile p3, Pile p4)
boolean doMove (theGame)
boolean undo (theGame)
boolean valid (theGame)
---
# Deck deck
# Pile pile1, pile2, pile3, pile4
```

This Move class represents the dealing of four cards from the deck to the four piles. As with all Move objects, it is responsible for determining whether a given move (as represented by a Move object) is valid. A move encodes the logic required to perform a move (in this case, dealing four cards) and undoing that move (returning the cards from the respective piles back to the deck in order).

MoveCardMove Class

```
MoveCardMove extends Move
---
MoveCardMove (Pile from, Card movingCard, Pile to)
boolean doMove (theGame)
boolean undo (theGame)
boolean valid (theGame)
# boolean toLeftOf(Pile pile1, Pile pile2)
-----------------------------------------
# Pile from, to
# Card movingCard
```

This Move class represents the moving of a card from one column to another. As with all Move objects, it is responsible for determining whether a given move (as represented by a Move object) is valid. The helper method toLeftOf (p1, p2) is used to validate a move.

RemoveAllMove Class

```
RemoveAllMove **extends** Move
-----------------------------------------
RemoveAllMove (Pile p1, Pile p2, Pile p3, Pile p4)
ReMoveAllMove (Pile p1, Pile p2, Pile p3, Pile p4,
               Card c1, Card c2, Card c3, Card c4)
boolean doMove (theGame)
boolean undo (theGame)
boolean valid (theGame)
-----------------------------------------
# Pile pile1, pile2, pile3, pile4
# Card removedCard1, removedCard2, removedCard3, removedCard4
```

This Move class represents the removal of all cards from the table. As with all Move objects, it is responsible for determining whether a given move (as represented by a Move object) is valid. Note that the object must remember the removed cards so the undo can be properly executed. The second constructor (with 8 arguments) is never called during interactive play, but becomes useful when you consider writing code to automatically play Narcotic.

Summary

For this solitaire plug-in, we need the following control objects.

```
NarcoticDeckController deckController
NarcoticPileController pileController
SolitaireMouseMotionAdapter standardDragController
SolitaireUndoAdapter standardUndoController
NarcoticReleasedAdapter releasedAdapter
```

The associations will be as follows:

{deckView, pileView1, pileView2, pileView3, pileView4} ➜ standardUndoController
{scoreView, numLeftView} ➜ standardUndoController
{deckView, pileView1, pileView2, pileView3, pileView4} ➜ standardDragController
{scoreView, numLeftView} ➜ standardDragController
{pileView1, pileView2, pileView3, pileView4} ➜ pileController
{deckView} ➜ deckController
{scoreView, numLeftView} ➜ releasedAdapter

Finally: You need to have some default controller to handle the Container; that is, mouse events that do not occur over any specific view widget.

{container} ➜ releasedAdapter
{container} ➜ standardDragController
{container} ➜ standardUndoController

By going over this tutorial, I could understand how to create a deck, a pile and interact with them. Basically, after exercising the Narcotic example, it was easy to write any variation which has a deck and any number of piles. In spite of that, I still didn't have enough understanding of the framework to complete the FourteenOut variation, because it was slightly different. I needed columns, which were not described in the tutorial, and the moves happened to be different from the moves in Narcotic. Therefore I looked at two or more other variations developed by students, to see how they customized the widgets and implemented various moves.

Designing and implementing the FourteenOut variation

The Fourteen-Out variation, since it extends the KS framework, follows the EBC (Entity - Boundary - Controller) pattern. Figure 23 shows a high level design of this variation. Later, we describe in more detail classes that were written for FourteenOut.

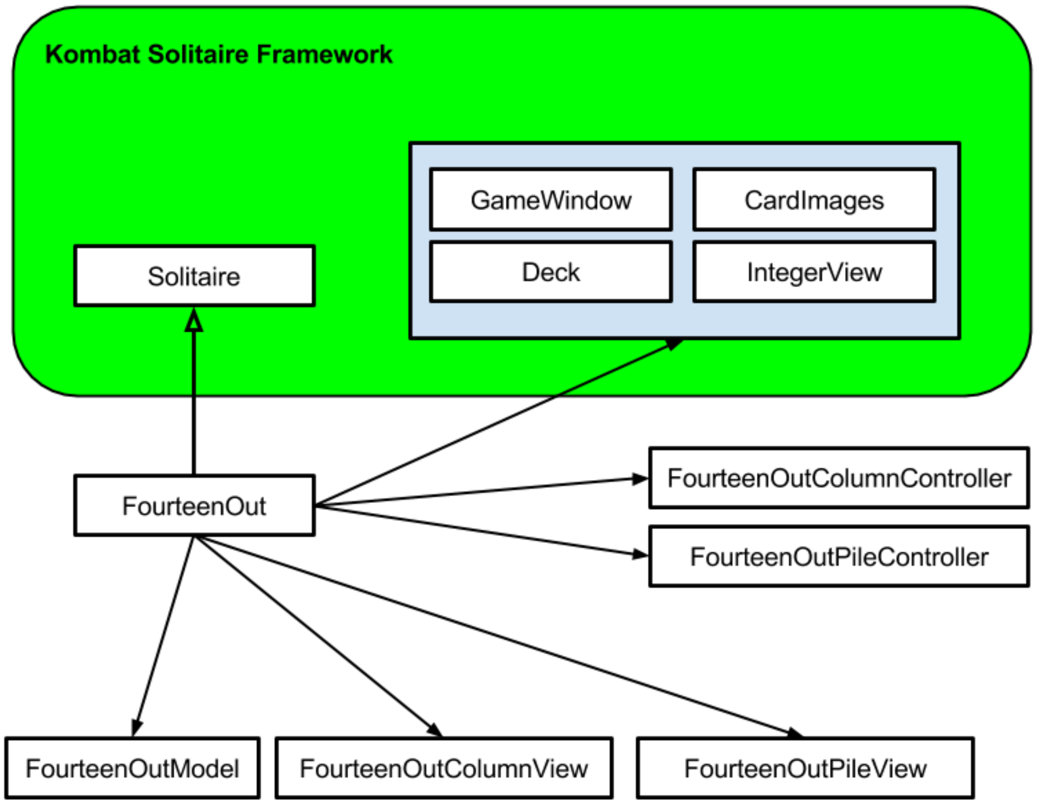


Figure 23: A high level design of the FourteenOut solitaire variation

FourteenOut.java

This is the class that combines all the pieces together, creates the model and the view, and starts the game. It does so by extending the `Solitaire` class and overriding `initialize()`, `hasWon()`, and `getName()` methods. Note: even though FourteenOut doesn't have a deck, we still need to use the `Deck` class to shuffle and deal the cards to columns and piles. The `IntegetView` class is used to show the score and number of cards left. `CardImages` lets us get the dimensions of a card, based on which we know how to lay out the view elements.

FourteenOutModel.java

This class is responsible for creating and maintaining the model elements of the game. In the picture below we show the relations between this class and other components.

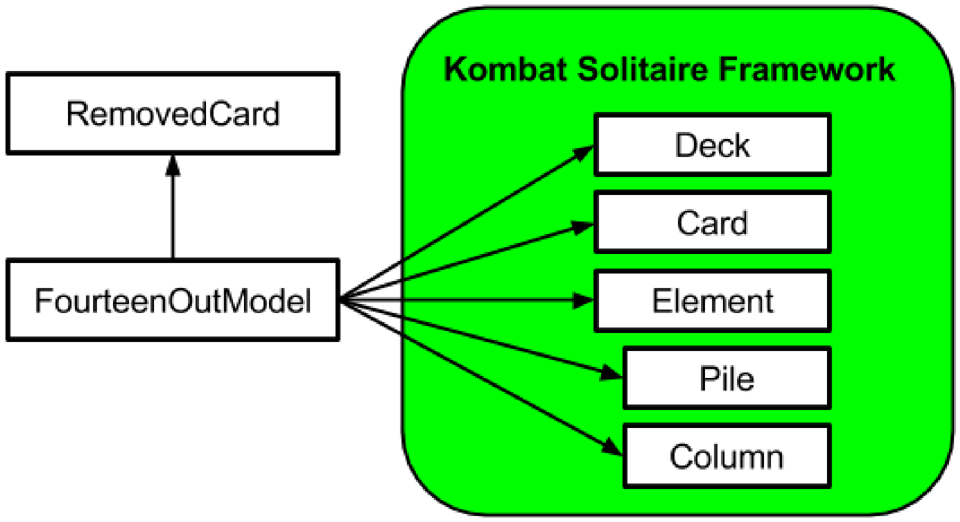


Figure 24: The model component of FourteenOut

The `Deck` class is used behind the scenes to shuffle the cards and deal them. The number of columns and piles is configurable; the game allows as many piles as there are columns. By default, there are 4 columns and 4 piles. Each column contains 12 cards, and the remaining 4 cards are dealt in 4 piles. Recall that a move consists of two steps: 1. select the first card, 2. select the second card; if their ranks add up to 14, then remove them, otherwise deselect both. The model keeps track of the selected card (if there's one, which means the first step was taken), and the list of all the removed cards. `RemovedCard` is nothing but a data structure that knows which card was removed, and from which column or pile. We keep track of the removed cards to able to implement the undo operation that the framework offers a hook for (this is accomplished by overriding the `undo(Solitaire s)` method of `Move`). The '`boolean isEmpty()`' method of `FourteenOutModel` is used to tell whether the game is won or not. This method checks all the columns and piles to determine if all the cards are removed.

FourteenOutColumnView.java *and* ***FourteenOutPileView.java***

The KS framework provides `ColumnView` and `PileView` classes, which can be used directly to draw a Column and a Pile, respectively. Since our column and pile widgets have a slightly different behavior from what is already provided by the framework, we needed to override the `redraw()` method of `CoumnView` and `PileView` so that it can show the selected card to the player.

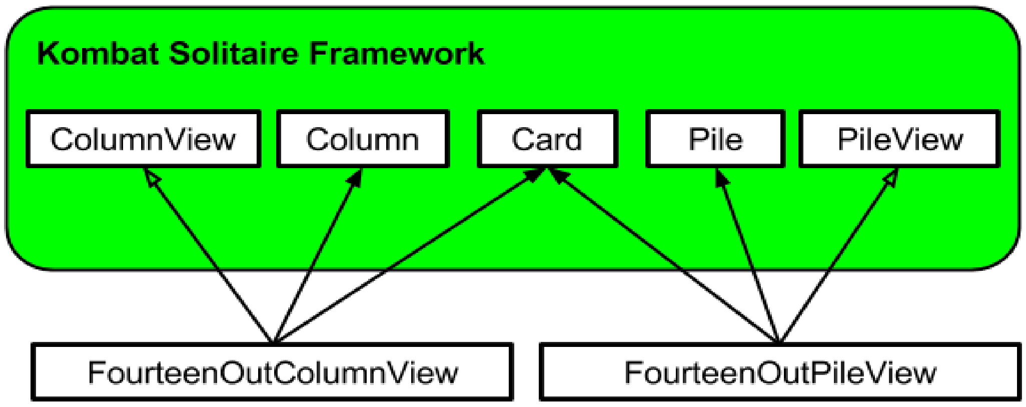


Figure 25: Customized widgets for ColumnView and PileView

FourteenOutColumnView extends ColumnView and FourteenOutPileView extends PileView. Each of them overrides the draw() method of its superclass, so that the selected card can be distinguished from others.

Controllers

The job of these controllers is to listen to mouse events and act accordingly.

FourteenOutPileController is the class that listens to mouse events by extending the MouseAdapter class; if there is a mouse click over a pile it creates an object of FourteenOutPileMove to register the move. FourteenOutPileMove is the class that extends the Move class and is responsible for validating a move and registering it, if it's valid. It is also responsible for implementing the undo action, which is the reverse of the move action. In the same way FourteenOutColumnController handles the moves of cards in a column. The picture below shows the structure of the controller component.

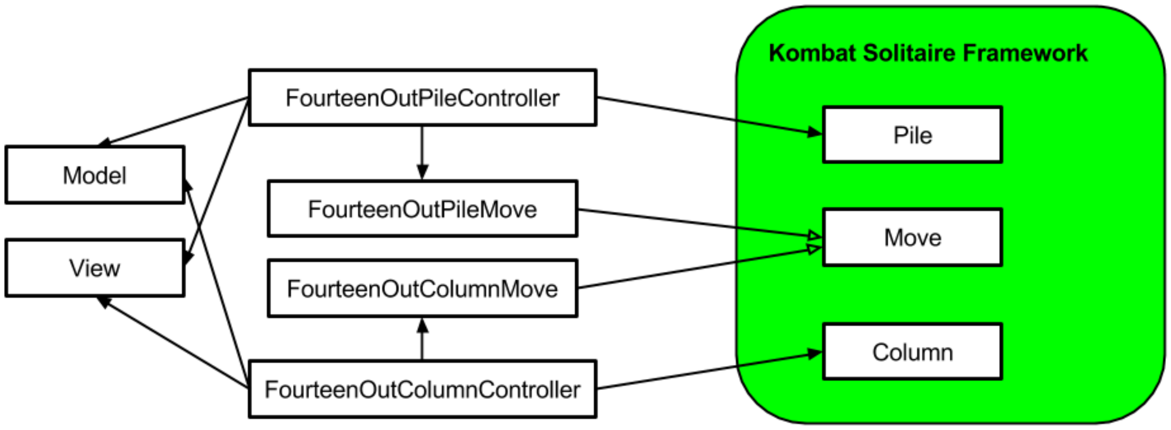

Kombat Solitaire Framework
FourteenOutPileController
Model
View
FourteenOutPileMove
FourteenOutColumnMove
FourteenOutColumnController
Pile
Move
Column

Figure 26: The controller component of FourteenOut. The Model and View boxes represent classes that we described earlier in this section.

3.1.2. Native Lambda / Combinator implementation

Here we present a completely different approach to developing the FourteenOut variation using the KombatSolitaire framework. We define combinators, which represent the building blocks of a solitaire variation. When defining a combinator, the aim is to make it as generic as possible, so that it can be reused among variations that share the behavior provided by this combinator. We use the CLS tool to synthesize the desired code from these combinators. Combinators are written in a language that we call L2; apart from lambda notations, it contains Java code fragments as well.

The first version contains the combinators that create the Model and View, at this point no move is possible. The process of writing the combinators for FourteenOut, at this point, is a migration process; since we already have the solution implemented in native Java code, we use that code to define the combinators out of which the CLS tool will produce the "same" code in terms of functionality.

WinRule : { box [""] **}**

The above line defines a combinator named "WinRule", which will be used to determine if the game is won. Inside double quotes of `box[""]` we can write any java code, it can be any valid java code, a single line, a block of lines, a method or a whole class. It can also contain other combinators. Java code and combinators can be interleaved inside a box structure; Java code should be enclosed in double quotes, whereas combinators not. Here's the list of combinators for the first, the most basic, version of FourteenOut.

Table 2: The list of combinators for the first version of FourteenOut

NameRule : { box ["FourteenOut"] }	Wherever "NameRule" is used, when synthesized it will be replaced with the string "FourteenOut".

```
Game : {
  λtc.      {letbox TestCases           = {tc}      in {
  λhelps.   {letbox HelperFunctions     = {helps}   in {
  λcontrs.  {letbox Controllers         = {contrs}  in {
  λmethod.  {letbox MethodDeclarations  = {method}  in {
  λfield.   {letbox FieldDeclarations   = {field}   in {
  λwin.     {letbox WinParameter        = {win}     in {
  λinit.    {letbox InitializeParameter = {init}    in {
  λname.    {letbox NameParameter       = {name}    in {
              box ["//... java code ..."]
      }} }} }} }} }} }} }} }}
}
```
	The "Game" combinator will produce the final pure java code for FourteenOut. The combinators, preceded with λ, inside the "Game" combinator are parameters of the "Game" combinator.

InitializationSteps : { λname. {letbox NameParameter = {name} in { box ["//... **java code** ..."] }} }	Steps used to initialize the game.
Fields : { box ["//... **java code** ..."] }	Fields used in the model.

In this table we show only the implementation part of combinators, they also have the definition part, which we are omitting here. By running the CLS tool we get the first version of FourteenOut constructed through combinators. It doesn't have any functionality, all it has is the model and some elements of the view. The picture below shows how it looks like.

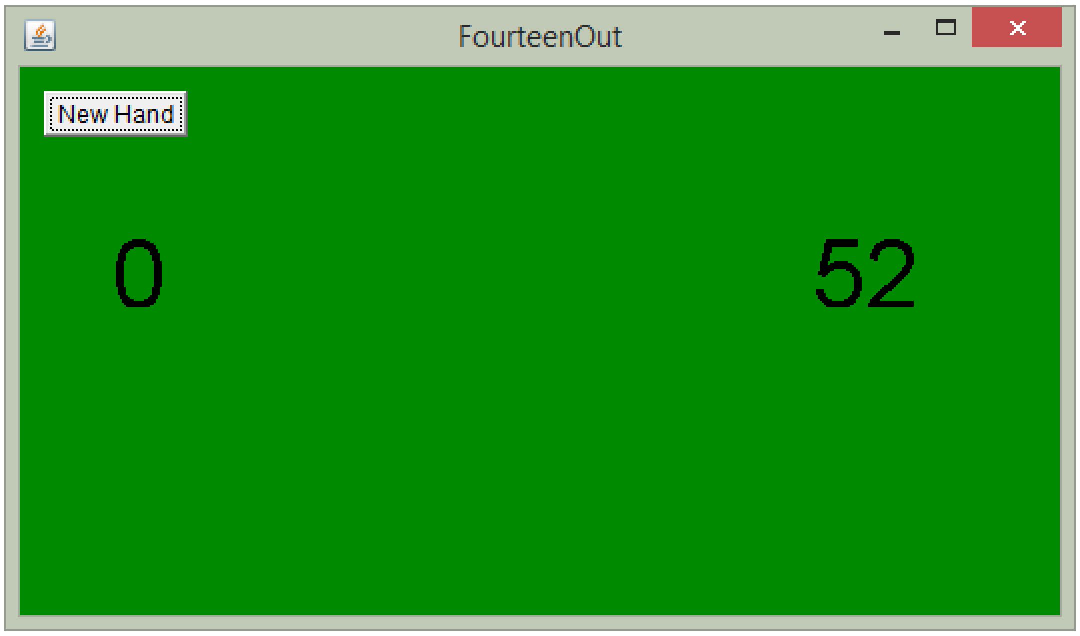


Figure 27: The first version of FourteenOut implemented through combinators

In this version we have only four combinators: **NameRule**, **Game**, **InitializationSteps** and **Fields**. The **NameRule** combinator contains the string "FourteenOut", which basically defines the name of the variation. Combinators can have parameters; such is the **Game** combinator. It contains the Java code for implementing the class which extends the `Solitaire` class, and it uses other combinators for building up the game. Similarly, **InitializationSteps** and **Fields** contain the code for initializing the game and specifying the fields needed, respectively.

The process of developing combinators is incremental, we make a small step and try it out. If it works we move on to the next step. This way we prove by construction that the combinators generate the correct code, and we can easily step back if something is done wrong and correct the mistakes.

In the following table we show all the combinators in the final version of FourteenOut, for the sake of brevity we skip the intermediate versions of the developing process.

Table 3: The list of the combinators for the final version of FourteenOut

WinRule : { box ["// **... java code ...**"] }	Determines when the game is won.
NameRule : { box ["FourteenOut"] }	Wherever "NameRule" is used, when synthesized it will be replaced with the string "FourteenOut".
PileRule : { box ["10"] }	Defines the number of piles and columns, in this case 10 of each.

Game : {
    λtc. {letbox TestCases = {tc} in {
    λmoves. {letbox MoveFunctions = {moves} in {
    λcontrs. {letbox Controllers = {contrs} in {
    λmethod. {letbox MethodDeclarations = {method} in {
    λfield. {letbox FieldDeclarations = {field} in {
    λwin. {letbox WinParameter = {win} in {
    λinit. {letbox InitializeParameter = {init} in {
    λname. {letbox NameParameter = {name} in {
        box [
            "//**... java code ...**"
            NameParameter "**... java code ...**"
            NameParameter "//**... java code ...**"
            NameParameter "// **... java code ...**"
            NameParameter "// **... java code ...**"
            FieldDeclarations "//**... java code ...**"
            Controllers "// **... java code ...**"
            MethodDeclarations "// **... java code ...**"
            WinParameter "// **... java code ...**"
            InitializeParameter "//**... java code ...**"
            NameParameter "// **... java code ...**"
            NameParameter "// **... java code ...**"
            MoveFunctions
            TestCases]
            }} }} }} }} }} }} }} }}
    }
	The "Game" combinator will produce the final pure java code for FourteenOut. The combinators, preceded with λ, inside the "Game" combinator are parameters of the "Game" combinator. Combinator parameters are replaced with their corresponding implementation code.

InitializationSteps : {
        λname. {letbox NameParameter = {name} in {
	Steps used to initialize the model and the view.

box ["//... **java code** ..."] }} }	
Methods : { box ["// **... java code ...**"] }	This combinator contains the implementation of methods for initializing the view and the model.

Fields : {
λpiles. {letbox NumPiles = {piles} in {
box ["//**... java code ...**"
NumPiles "//**... java code ...**"]
}
	Fields used in the model. We have added the "NumPiles" parameter to this combinator, which was not in the first version.

FOColumnView : {
λname. {letbox NameParameter = {name} in {
box ["//**... java code ...**"
NameParameter "//**... java code ...**"]
}} }
	Contains the **FourteenOutColumnView** class described in the section "**Designing and implementing the FourteenOut variation**"

FOPileView: {
λname. {letbox NameParameter = {name} in {
box ["//**... java code ...**"
NameParameter "//**... java code ...**"]
}} }
	Contains the **FourteenOutPileView** class described in the section "**Designing and implementing the FourteenOut variation**"

RemovedCard : {
λname. {letbox NameParameter = {name} in {
box ["//**... java code ...**"
NameParameter "// **... java code ...**"]
}} }
	Contains the **RemovedCard** class described in the section "**Designing and implementing the FourteenOut variation**"

Model: {
λname. {letbox NameParameter = {name} in {
box ["//**... java code ...**"
NameParameter "// **... java code ...**"]
}} }
	Contains the **FourteenOutModel** class described in the section "**Designing and implementing the FourteenOut variation**"

FourteenOutColumnMove: {
λname. {letbox NameParameter = {name} in {
box ["//... java code ..."
NameParameter "// ... java code ..."]
	Contains the **FourteenOutColumnMove** class described in the section "**Designing and**

}} }	**implementing the FourteenOut variation**”

FourteenOutPileMove: {
λname. {letbox NameParameter = {name} in {
box ["//**... java code ...**"
NameParameter "// **... java code ...**"]
}} }
	Contains the **FourteenOutPileMove** class described in the section “**Designing and implementing the FourteenOut variation**”

FourteenOutColumnController: {
λname. {letbox NameParameter = {name} in {
box ["//**... java code ...**"
NameParameter "// **... java code ...**"]
}} }
	Contains the **FourteenOutColumnController** class described in the section “**Designing and implementing the FourteenOut variation**”

FourteenOutPileController: {
λname. {letbox NameParameter = {name} in {
box ["//**... java code ...**"
NameParameter "// **... java code ...**"]
}} }
	Contains the **FourteenOutPileController** class described in the section “**Designing and implementing the FourteenOut variation**”

Controllers : {
λc. {letbox Controller = {c} in {
box [Controller]
}} }
	This combinator combines all defined controllers together.

Moves : {
λm. {letbox Move = {m} in {
box [Move]
}} }
	This combinator combines all defined moves together.

AllTestCases : { box ["// **... java code ...**"]}	Contains all test cases.

The final version contains 18 combinators and all of them are shown in the above table. The CLS tool generates code that is human readable and behaves exactly the same way as the one implemented in native Java language. Nevertheless we're not supposed to touch the generated code even if a single change needs to be made. We always go back to the combinators and make the changes, run CSL tool again and see if we got the desired result.

Note that combinators are totally reusable, which is not the case with pure object oriented programming. Consider a solitaire variation that is 95% similar to **FourteenOut** but has some additional moves. We can reuse all the combinators from the repository as they are, and just add the extra moves we want our variation to have. But, this is not the case with pure OOP. Consider the winning condition. We need to override the `hasWon()` method of the `Solitaire` class to determine when the game is won. Every time we write a new variation we need to repeat the step even though the condition may be the same. So we end up copy-pasting code, which is not the definition of reuse. What if we need to modify a little bit the winning condition in all variations that have the same winning condition? We have to go off and change the code wherever it is used. In case of the lambda/combinator implementation, we only change the **WinRule** combinator and rerun the CLS tool.

3.1.3. Review of the LaunchPad repository implementation of Fourteen-Out

Besides having it implemented in L2 language, we have implemented the Γ repository of Fourteen-Out in the LaunchPad tool as well. In this section we will review the combinators implemented using the LaunchPad tool and compare them with the ones in the previous section. The goal is to show the advantages of using this tool instead of manually developing combinators in L2. It doesn't mean that we are completely avoiding the usage of the L2 language, but rather use a more user-friendly tool for development, which automatically generates the cumbersome L2 code.

One of the major differences is that in LaunchPad we don't use the lambda expressions at all, though they get generated behind the scenes, so that the CLS tool will be able to synthesize code from those combinators. Another difference is that we don't have to have everything in only two files, where we put all the definition code into one file and the implementation code into another, but we can break down the code into smaller and more manageable modules. Unlike in L2, where implementation and definition reside in two different files, here we keep them together in one file.

In Table 3 we have shown only the implementation of the combinators, omitting the definition part of it. Let's have a look at the *Game* combinator, its definition and implementation.

```
//definition
Game : #[testcases, alpha.gameType, all] ->
         #[moves,    alpha.gameType] ->
         #[allControllers, alpha.gameType] ->
         #[methods, alpha.gameType] ->
         #[fields, alpha.gameType] ->
         #[winrule, alpha.gameType] ->
         #[initializationsteps, alpha.gameType] ->
         #[namerule, alpha.gameType] ->
         #[game, alpha.gameType]
```
```
//implementation
Game : {
     λtc.       {letbox TestCases          = {tc}       in {
     λmoves.    {letbox MoveFunctions      = {moves}    in {
```

```
     λcontrs.   {letbox Controllers              = {contrs}   in {
     λmethod. {letbox MethodDeclarations  = {method}  in {
     λfield.    {letbox FieldDeclarations       = {field}      in {
     λwin.      {letbox WinParameter             = {win}        in {
     λinit.      {letbox InitializeParameter    = {init}        in {
     λname.    {letbox NameParameter          = {name}    in {
        box [
             "//... java code ..."
             NameParameter "... java code ..."
             NameParameter  "//... java code ..."
             NameParameter "// ... java code ..."
             NameParameter "// ... java code ..."
             FieldDeclarations "//... java code ..."
             Controllers "// ... java code ..."
             MethodDeclarations "// ... java code ..."
             WinParameter "// ... java code ..."
             InitializeParameter "//... java code ..."
             NameParameter "// ... java code ..."
             NameParameter "// ... java code ..."
             MoveFunctions
             TestCases ]
            }} }} }} }} }} }} }} }}
}
```

Below is the code for the *Game* and other combinators in LauchPad; we are omitting the Java code for the sake of brevity.

Table 4: The list of combinators for the final version of Fourteen-Out in LaunchPad

```
/**
 Sets the structure for the primary extension class in Solitaire. Each of the bound variables pulls
in different elements as needed.
 */
type Game {
  NameParameter          [alpha.gameType, namerule];
  MethodDeclarations     [alpha.gameType, methods];
  FieldDeclarations      [alpha.gameType, fields];
  WinParameter           [alpha.gameType, winrule];
  InitializeSteps        [alpha.gameType, initializationsteps];
                         [alpha.gameType, game];
}
implementation Game (<NameParameter>/<NameParameter>.java) {
package <NameParameter>;//... Java Code ...
    <NameParameter> //... Java Code ...
    <FieldDeclarations>
      <MethodDeclarations> //... Java Code ...
      <WinParameter>//... Java Code ...
      <InitializeSteps>//... Java Code ...
      "<NameParameter>"; //... Java Code ...
}
```

```
/* Each solitiare variation has a winning completion condition. */
type WinRule {
      [fourteenout, winrule];
}

/* Default implementation determines win once score reaches 52.*/
implementation WinRule {
      if (model != null) { return isEmpty(); }
}
```

```
/**
Building off of the default Moves combinator in the library, this table demonstrates how to launch
four different inhabitation searches, each one of which generates the appropriate helper code for
dealing with the dragging moves in this variation.

These moves are simpler than in most variations, so we don't use the DragMoves feature, but
rather implement these as standalone.
 */
table Moves {
  type {
      Move              [fourteenout, pile_remove_cards];
                        [fourteenout, moves, pilemoves]; }
  type {
      Move              [fourteenout, column_remove_cards];
                        [fourteenout, moves, columnmoves]; }
}
```

```
type FourteenOutPileMove {
      NameParameter       [fourteenout, namerule];
                          [fourteenout, pile_remove_cards];
}
// missing a close parentheses will drive you nuts because there won't be an implementation. Spot
// syntax error?
implementation FourteenOutPileMove
(<NameParameter>/FourteenOutPileMove.java) {
      package <NameParameter>;
      //... Java Code ...
      <NameParameter> model = (<NameParameter>) game;
      //... Java Code ...
}
```

```
type FourteenOutColumnMove {
      NameParameter       [fourteenout, namerule];
                          [fourteenout, column_remove_cards];
}
implementation FourteenOutColumnMove
(<NameParameter>/FourteenOutColumnMove.java) {
package <NameParameter>;
      //... Java Code ...
      <NameParameter> model = (<NameParameter>) game;
      //... Java Code ...
}
```

```
table AllControllers {
  type {
      Controller [fourteenout, pileController];
                 [fourteenout, allControllers];
  }

  type {
      Controller [fourteenout, columnController];
                 [fourteenout, allControllers];
  }
}
```

```
type ColumnDesignate {
      [alpha.gameType, columnControllerName];
}
implementation ColumnDesignate { // empty string }

type CMousePressed {
      [fourteenout, columnPressed ];
}
implementation CMousePressed {
    //... Java Code ...
}

type CMouseClicked {
      [fourteenout, columnClicked];
}
implementation CMouseClicked {// ignore }

type CMouseReleased {
      [fourteenout, columnReleased];
}
implementation CMouseReleased {// ignore}
```

```
type PileDesignate {
      [alpha.gameType, pileControllerName];
}
implementation PileDesignate {// empty string }

type PileMousePressed {
      NameParameter     [fourteenout, namerule];
                        [fourteenout, pilePressed];
}
implementation PileMousePressed {
      //... Java Code ...
}

type PileMouseClicked {
      NameParameter     [fourteenout, namerule];
                        [fourteenout, pileClicked];
}
implementation PileMouseClicked {// ignore}

type PileMouseReleased {
      [fourteenout, pileReleased];
}
implementation PileMouseReleased {// ignore}
```

```
/* Contains description of the extra fields to be added to FourteenOut structure. Required
parameters include the number of piles.  */
type Fields {
      NumPiles    [fourteenout, pilerule];
                  [fourteenout, fields];
}
implementation Fields {
      //... Java Code ...
      int numberOfColumns = <NumPiles>;
      //... Java Code ...
}
```

```
/* Initialization depends on key structural concepts, namely the number of piles. */
type InitializationFourteenOut {
      NameRule          [fourteenout, namerule];
                        [fourteenout, initializationsteps];
}
/* All structural elements and widgets are constructed; in addition, the expected controllers are
associated here. */
implementation InitializationFourteenOut {
      //... Java Code ...
}
```

```
/* When a combinator refers to a single token (a common occurrence) then use 'define' to
capture this relationship. */
define {
  [fourteenout, namerule]           NameRule -> FourteenOut;
}
```

```
/* Simple definition capturing the concept that there are ten piles within the game. */
define {
  [fourteenout, pilerule] PileRule -> 10;
}
```

```
type Methods {
      [fourteenout, methods];
}
implementation Methods {
      //... Java Code ...
}
```

```
type RemovedCard {
      NameParameter     [fourteenout, namerule];
                        [fourteenout, removedCard];
}
implementation RemovedCard (<NameParameter>/RemovedCard.java) {
      package <NameParameter>;
      //... Java Code ...
}
/* Each solitaire variation has a winning completion condition. */
type WinRule {
      [fourteenout, winrule];
}
/* Default implementation determines win once score reaches 52. */
implementation WinRule {
      if (model != null) { return isEmpty(); }
}
/* The Full codebase is generated once individual elements are generated. Each of these subtasks
*MUST* expand to its own file, otherwise there will be text that bleeds over from one abstraction
to the next. */
type Full {
  Moves             [fourteenout, moves];
  RemovedCard       [fourteenout, removedCard]; // helper class
  FOColViews        [fourteenout, columnview];  // special widget
  Controller        [fourteenout, allControllers];
  Game              [fourteenout, game];
                    [fourteenout, full];
}
```

The original FourteenOut in CLS was completed as a stand-alone project. To properly integrate this code into the Γ repository, we made several changes to properly align the new combinator code with the existing combinators. The resulting product line is thus formed from the intersection of the "globally useful" combinators used across multiple members. Naturally when working on successive solitaire variations using the Γ repository, future developers would start by using the existing combinators, and only would develop new ones relevant for their specific variation.

3.2. KombatSolitaire

The KombatSolitaire (KS) framework [24] is an OO framework developed over a number of years as part of an undergraduate course in software engineering. KS is a Java framework that enables head-to-head competition of solitaire variations played simultaneously over the Internet. KS contains about 67KLOC, of which 31KLOC form the core Solitaire-playing engine. The objective of the framework was to develop dozens, even hundreds, of solitaire plugins to be executed by KS. The framework designer wrote a tutorial showing students how to develop a sample variation from scratch (see Tutorial for The Narcotic Variation). Specifically, a Java programmer must implement a number of classes with designed interrelationships between them:

- Create a named class as subclass of Solitaire
- Define structure of element objects
- Define structure of widget objects

- Create move subclasses of Move for each move type
- Create controller classes to process mouse events to create move objects to be executed
- Determine logical condition for when game is over
- Write test cases that properly evaluate implementation

Following this approach, a typical implementation of the popular FreeCell solitaire variation requires ten classes and 1,565 commented lines of Java code. In doing so, the programmer applied the Model/View/Controller design pattern [4] and properly implemented the necessary coding protocols imposed by the framework. Hundreds of students have repeated this task, each one having to learn the abstractions encoded in the framework.

3.3. KombatSolitaire Γ Repository

Given the core OO framework [24] that supports the solitaire variations, it makes sense to create a product line from this OO framework.

NameRule$_{type}$: (*freeCell* ∩ *namerule*)
NameRule$_{term}$: { box["FreeCell"] }

HomePileRule$_{type}$: (*freeCell* ∩ *pilerule* ∩ *homePile*)
HomePileRule$_{term}$: { box["4"] }

WinRule$_{type}$: (*freeCell* ∩ *pilerule* ∩ *homePile*) → (*freeCell* ∩ *winrule*)
WinRule$_{term}$: {

```
 λ piles. {letbox NumPiles = {piles} in {
  box["       boolean won = true;
              for (int i = 0; i < " NumPiles "; i++) {
              if (fieldHomePiles[i].count() != 13) {
              won = false;
              } }
              if (won) { return true; }
      "]
 }}}
```

Figure 28: Sample Combinators for FreeCell Variation

The challenge is to make this process configurable where one can synthesize individual product line members by selecting features from a feature diagram.

Starting from the original KS tutorial, we created a repository of combinators that encodes the logic required to extend the OO framework to implement a solitaire variation. During this process, we iteratively identified the core abstractions in the OO framework and mapped them to combinators at different levels of granularity. The sample combinators for a FreeCell variation shown in Figure 28 construct L1-level code fragments that can be composed with other code fragments. The **HomePileRule** combinator maps the concept number of home card piles (i.e., where the Aces are placed) to the integer value 4. By encoding this concept into a single combinator, the designer has separated concerns which can be reused in other combinators. The **WinRule** combinator produces a Java code fragment that determines whether the game has been won by checking whether all home card piles are full. This combinator depends on having the **HomePileRule** combinator so it can generate code using the appropriate number of piles. The code resulting from **WinRule** is synthesized from the L1-level code fragment by replacing the L2-level variable **NumPiles** with the L1-level code fragment, 4. Finally, the **NameRule** combinator maps to a string constant which refers to the name of the top-level class of the plugin implementation.

$$\begin{aligned}\mathbf{Game}_{\text{type}} :\ & (\mathit{testcases} \cap \mathit{alpha.gameType} \cap \mathit{all}) \rightarrow \\ & (\mathit{moves} \cap \mathit{alpha.gameType}) \rightarrow \\ & (\mathit{allControllers} \cap \mathit{alpha.gameType}) \rightarrow \\ & (\mathit{methods} \cap \mathit{alpha.gameType}) \rightarrow \\ & (\mathit{fields} \cap \mathit{alpha.gameType}) \rightarrow \\ & (\mathit{winrule} \cap \mathit{alpha.gameType}) \rightarrow \\ & (\mathit{initializationsteps} \cap \mathit{alpha.gameType}) \rightarrow \\ & (\mathit{namerule} \cap \mathit{alpha.gameType}) \rightarrow \\ & (\mathit{game} \cap \mathit{alpha.gameType})\end{aligned}$$

Figure 29: Game Combinator

The power of this approach comes from its ability to assign type information to intermediate code fragments synthesized from combinators. Referring to the inhabitation problem from Section 2.2, the code for the FreeCell solitaire plugin is generated by applying the generic Game combinator shown in Figure 29 in a query, viz. $\Gamma \vdash e : (\mathit{game} \cap \mathit{freeCell})$. The inhabitation uses this goal query to drive the generation of code resources as required by the FreeCell solitaire variation.

Identifying this combinator is important because the individual clauses in this combinator directly parallel the bulleted list (Section 3.2) that described the tutorial steps to follow when extending the framework. There is no ability in Java alone to specify that these steps must be carried out. The very structure of the **Game** combinator actually provided guidance as we refactored the KS tutorial through a number of iterations. In the same way that the original KS tutorial guided students through a series of executable iterations – each one completing more features of the target solitaire variation – we were able to iteratively add combinators to the repository, always checkpointing our progress by executing the synthesized code to validate that the newly generated code was meeting its obligations.

```
define {
 [freeCell, namerule] NameRule -> FreeCell;
}
define {
 [freeCell, pilerule, homePile] HomePileRule -> 4;
}
type WinRule {
 NumPiles [alpha.gameType, pilerule, homePile];
                  [alpha.gameType, winrule];
}
implementation WinRule {
 boolean won = true;
 for (int i = 0; i < <NumPiles>; i++) {
   if (fieldHomePiles[i].count() != 13) { won = false; break; }
  } if (won) { return true; }
}
```

Figure 30: Equivalent LaunchPad Combinators

Ultimately a framework designer provides a library of combinators that encodes the logic required for extensions, and then the framework extender fills in the combinator implementations as required by the logic of their extensions. Note that the *alpha.gameType* type appearing in the clauses in Figure 30 clarifies that this combinator is reused "as is" for any solitaire variation, not just the FreeCell variation discussed here.

The KS tutorial (showing how to implement a solitaire variation known as Narcotic) was first converted into 24 combinators composed of 962 lines of L2-code, containing appropriate L1-code fragments (in Java). Synthesizing Narcotic generated seven classes consisting of 609 lines of L2-code, containing appropriate L1-code fragments (in Java). The documentation embedded with the L1-code becomes part of the synthesized result, thus improving its readability. In addition to the L1-code fragments that implement the variation, the combinators also embedded JUnit code test cases which validate the synthesized code.

We next synthesized a FreeCell variation requiring 58 combinators composed of 2,185 lines of L2-code which generated fourteen classes consisting of 1,250 lines of readable, commented Java code. In reviewing these two solitaire variations, we consolidated common logic, extracting seven generic combinators which formed the basis for a common repository Γ of combinators. While each of these implementations requires some unique combinators to represent the individual logic of the variations, they all share the common architecture defined by the collection of combinators in the repository that formally encodes the abstractions and the way these abstractions are composed and interact with each other. Upon reflecting on the effort to create these two variations, it was clear that we needed to "engineer" the design of these combinators in a systematic fashion. Naturally, this is the same impulse that leads to the development of software product lines in the first place.

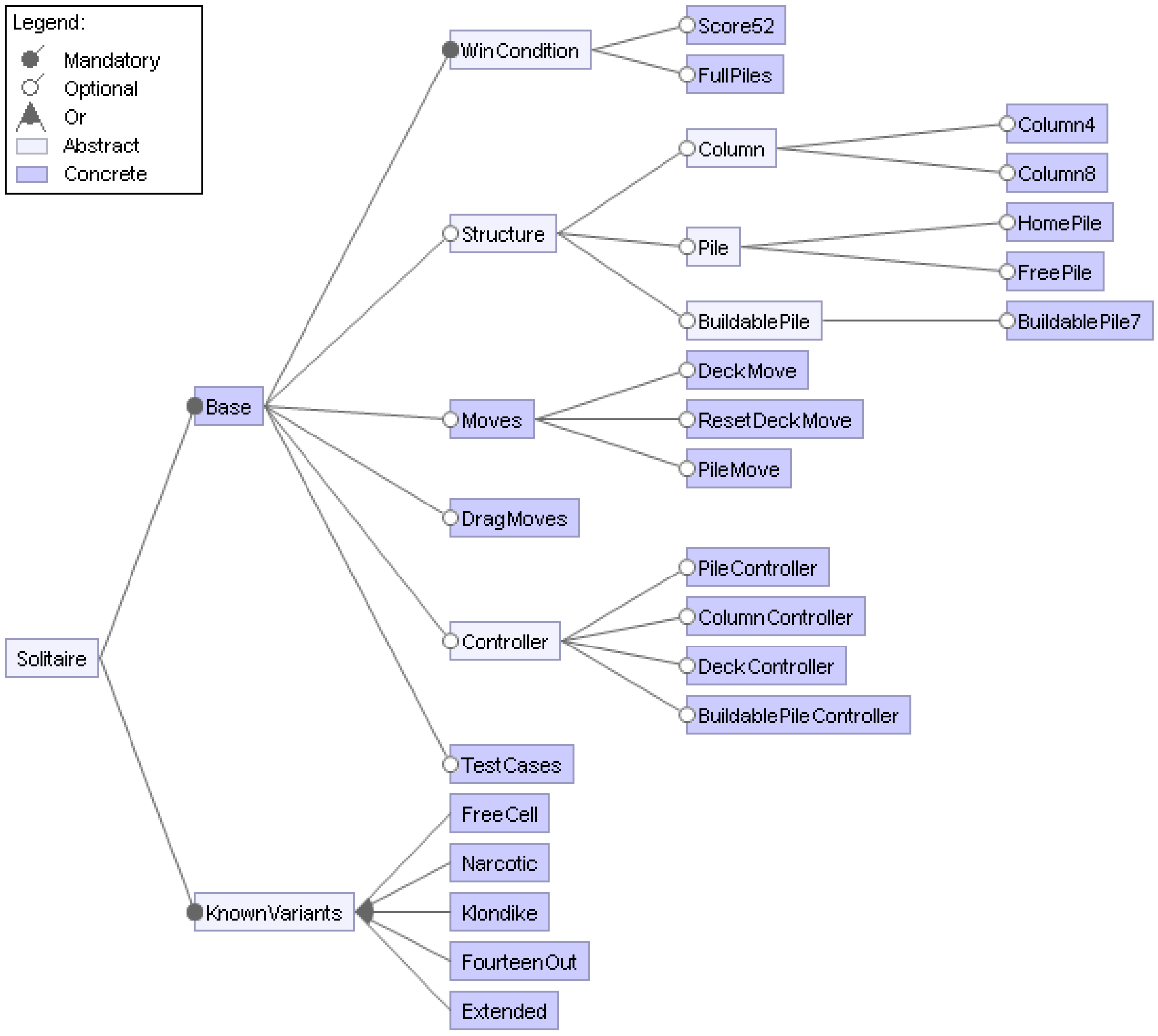


Figure 31: Solitaire Feature Model

Briefly, a solitaire variation determines its **WinCondition**, **Structure**, mouse **Controllers**, and the allowed **Moves** that provide the logic of the desired solitaire variation. As the Γ repository matured, the feature diagram evolved to incorporate an increasing number of generic combinators used to synthesize different variations. The final feature model of the Solitaire framework is shown in Figure 31. Note that throughout this process, the OO framework remained unchanged because all development work was focused on the product line code.

Each product line member has at least one inhabitation file that determines the inhabitation query used to generate the code. For FreeCell, this target is the intersection type `[freeCell, full]`. LaunchPad is flexible enough to support multiple code generation steps as desired by the product line designer.

Each valid product line member is defined by a configuration which represents a valid subset of the features defined in the feature model, based upon the semantics of the diagram. Table 5 lists three variations – FreeCell, FourteenOut, and Narcotic – and the features that are included in their respective configurations. Figure 32 depicts a sample execution of the synthesized FreeCell variation.

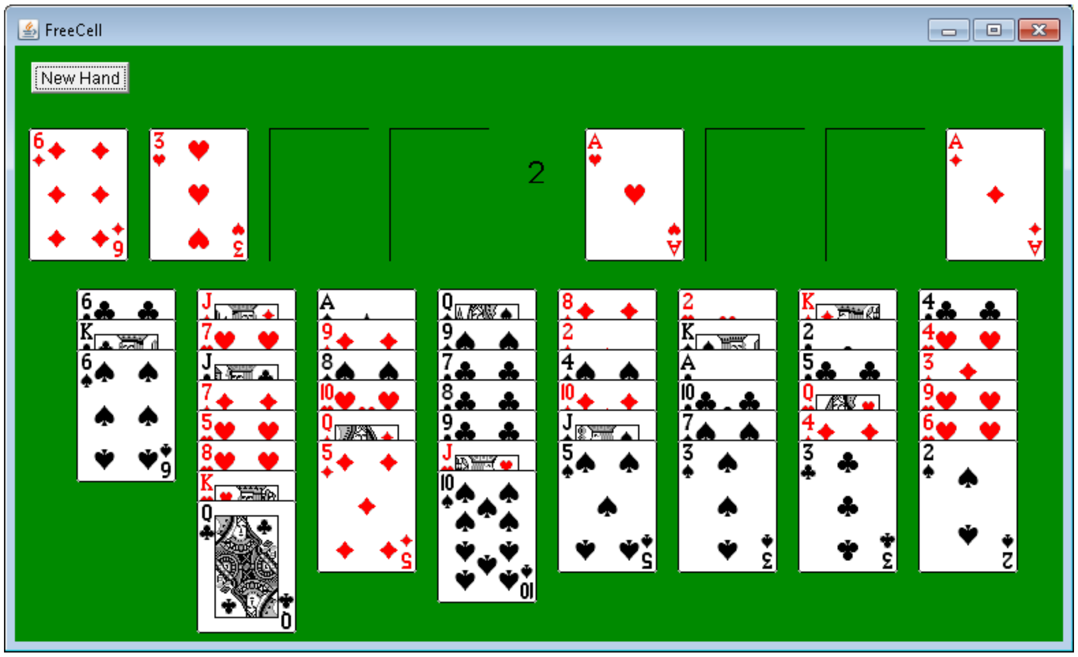


Figure 32: Sample FreeCell Execution

Table 5 identifies that some features are used by all product line members, while others are used by one or two of the members. We continue to increase the number of variations in this repository, which will only improve the reuse of combinators.

Table 5: Sample Solitaire Configurations

	FreeCell	**FourteenOut**	**Narcotic**
Base	✓	✓	✓
Moves	✓	✓	✓
PileController	✓	✓	✓
Column8	✓		
DragMoves	✓		
FreeCell	✓		
FreePile	✓		
FullPiles	✓		
HomePile	✓		

ColumnController	✓	✓	
TestCases	✓		✓
DeckController			✓
DeckMove			✓
FourteenOut		✓	
Narcotic			✓
PileMove		✓	✓
ResetDeckMove			✓
Score52			✓
Generation Statistics	18 classes 1438 LOC	7 classes 696 LOC	7 classes 562 LOC

To complete the support for migration we needed to integrate the command line utility (called InhabConsoleClient [21]) into an integrated development environment. We selected Eclipse because of its existing support for FeatureIDE [23]. See Section 2.4 for details on the LaunchPad Eclipse plugin.

3.4. MOEA Framework

The MOEA framework [27] provides a base set of algorithms and defined problems[1], but can easily be extended by adding new algorithms and problems. The manual focuses on explaining how to define new problems and gives detailed examples of two different problems. We will be experimenting with the possibility of defining new problems and using the built-in algorithms to solve any optimization problem that we can define within the MOEA framework. The variability of optimization problems, Knapsack problems for instance, gives us a good example of defining and using combinators in the MOEA Framework Repository implementation (see section 3.5 for details).

In this section, we will give a brief introduction to the framework, how to use it to solve optimization problems and define new problems, which then can be solved using the algorithms provided by the framework.

Most of the functionality provided by the framework is spread out across three classes: *Executor*, *Instrumenter* and *Analyzer*. We will explore the *Executor* class only, as the two others have to do with performance and analysis of the algorithms and are not quite relevant for us. The *Executor* class is used to construct and execute an algorithm. There are three pieces of information needed to run an algorithm:

1. the problem to be solved
2. the algorithm for solving the problem

[1] In this section we use the words *"problem"* and *"optimization problem"* interchangeably

3. the number of objective function evaluations allocated to solve the problem

The code snippet below shows how the *Executor* object is constructed and run.

Table 6: Solving the UF1 problem using the NSGA-II algorithm.

```
1.   NondominatedPopulation result = new Executor()
2.                      .withProblem("UF1")
3.                      .withAlgorithm("NSGAII")
4.                      .withMaxEvaluations(10000)
5.                      .run();
```

Line 1 creates a new instance of the *Executor* class; lines 2, 3, and 4 set the problem, algorithm and the maximum number of objective function evaluations, respectively. This example shows how to solve the two-objective UF1 problem using the NSGA-II algorithm. And finally, line 5 runs the algorithm and returns the result as *NondominatedPopulation*. In the code above, the problem and algorithm are provided by the framework, therefore we set them as Strings. Changing the problem or the algorithm is as easy as changing the String parameter. For example, if we want to use a different algorithm, let's say GDE3 (Generalized Differential Evolution 3), we call the *withAlgorithm("GDE3")* method and off we go, now the problem will be solved using the GDE3 algorithm. For the complete list of the algorithms provided by the framework see the API.

Once the execution is finished, we can access the result, which is saved in the variable named 'result'.

Table 7: Printing the solution obtained by the Executor in Table 6.

```
 for (Solution solution : result) {
    System.out.println(solution.getObjective(0) + " " +
    solution.getObjective(1));
 }
```

The algorithms provided by the MOEA framework have many parameters, which, if not set explicitly, are assumed to have the default values. The code snippet in Table 6 uses the NSGA-II algorithm with the default parameterization. If we want to set the values of any of its parameters different from default ones, we can do so by calling the `setProperty` method. The code snippet in Table 8 shows an example.

Table 8: Setting the parameter values for NSGA-II.

```
NondominatedPopulation result = new Executor()
                .withProblem("UF1")
                .withAlgorithm("NSGAII")
                .withMaxEvaluations(10000)
                .withProperty("populationSize", 50)
                .withProperty("sbx.rate", 0.9)
                .withProperty("sbx.distributionIndex", 15.0)
                .withProperty("pm.rate", 0.1)
                .withProperty("pm.distributionIndex", 20.0)
                .run();
```

Each algorithm has its own parameters. Refer to the API documentation for a complete and exact parameter keys.

3.4.1. Defining a new problem

The real power of the MOEA framework comes from the possibility of introducing any multi-objective optimization problem, which can be solved using the algorithms that this framework provides. The problems can be introduced in Java, C/C++, and in scripting languages. But, since our target language is Java, we'll explain how to do it only in Java. For other implementations see the manual [51].

All problems in the MOEA framework implement the `Problem` interface, therefore we can introduce to the framework any multi-objective optimization problem by implementing this interface. It defines methods for characterizing a problem, defining the representation of the problem, and evaluating solutions to the problem. Another hot-spot of the framework, which in practice is used more than the `Problem` interface, is the `AbstractProblem` class. This class provides default implementations for many of the methods required by the `Problem` interface. We'll explain briefly two examples of defining new problems, which are taken from the manual provided with the framework. For detailed implementations see the manual [51].

Kursawe Problem

The Kursawe problem is formally defined as:

$$\underset{x \in R^L}{minimize} \quad F(X) = (f_1(x), f_2(x))$$

where

$$f_1(x) = \sum_{i=0}^{L-1} -10\, e^{-0.2\sqrt{x_i^2+x_{i+1}^2}}$$

$$f_2(x) = \sum_{i=0}^{L} |x_i|^{0.8} + 5sin(x_i^3)$$

The MOEA Framework only works on minimization problems. If any objectives in our problem are to be maximized, we can negate the objective value to convert from maximization into minimization. In other words, by minimizing the negated objective, we are maximizing the original objective.

Table 9: Implementation of the Kursawe problem by extending the AbstractProblem class.

```
public class Kursawe extends AbstractProblem
```

```
public Kursawe(){...}
```

```
@override
public Solution newSolution() {...}

@override
public void evaluate(Solution solution) {...}
```

After having implemented the `Kursawe` class we can use it with the MOEA framework. The code snippet below shows how to solve the Kursawe problem using the NSGA-II algorithm. Note that in this case we don't use the `withProblem` method, but `withProblemClass`, since we are providing our own definition of the problem.

Table 10: Solving the Kursawe problem using the NSGA-II algorithm.

```
new Executor()
        .withProblemClass(Kursawe.class)
        .withAlgorithm("NSGAII")
        .withMaxEvaluations(10000)
        .run();
```

To print the solutions, we use the same method as described in the Table 7.

Knapsack Problem

In this section we will solve a problem, which is a multi-objective version of the famous Knapsack problem (discussed in detail in [28]). This is the problem of choosing which items to carry in a knapsack to maximize the value of the items without exceeding the weight capacity of the knapsack.

The formal definition of the problem is:

We are given N items. Each item has a profit denoted as $P(i)$, and a weight denoted as $W(i)$, for i = 1, 2, … , N. Let $d(i)$ represent the decision of including the i-th item in the knapsack, where $d(i) = 1$ meaning the item is included, and $d(i) = 0$ meaning the item is excluded. Let C be the weight capacity of the knapsack, then the problem is defined as:

$$Maximize \sum_{i=1}^{N} d(i) * P(i), such\ that \sum_{i=1}^{N} d(i) * W(i) \le C$$

The left-hand side summation calculates the profit we get by putting the items in the knapsack, and the right-hand side summation is a constraint, which ensures that the capacity of the knapsack is not exceeded.

The problem that we will introduce to the MOEA framework, is similar to this problem, except that it has two knapsacks to hold the items. Additionally, the weights and profits of items vary depending on which knapsack is holding them. For example, an item may have the profit of \$25 and weight of 4 kg in the first knapsack, but in the second knapsack it may have the profit of \$15 and weight of 5 kg. It seems unusual, but this is how it is defined in the literature. Since we have two knapsacks now, the profit is defined as $P(i,j)$and weight is defined as $W(i,j)$, where $j = 1{,}2$ is the knapsack index. In this case, each knapsack has its own capacity defined as C_1 and C_2. The Two-Knapsack problem is defined as:

$$Maximize \sum_{i=1}^{N} d(i) * P(i,1), such\ that \sum_{i=1}^{N} d(i) * W(i,1) \le C_1$$

$$Maximize \sum_{i=1}^{N} d(i) * P(i,2), such\ that \sum_{i=1}^{N} d(i) * W(i,2) \le C_2$$

The information required by the Knapsack problem - capacities, profits, weights - is loaded from a text file. The data is saved in a format developed by Eckart Zitler and Marco Laumanns (see [29]).

Table 11: Input file format for the multi-objective knapsack problem.

```
knapsack problem specification (2 knapsacks, 2 items)
=
knapsack 1:
 capacity: +251
 item 1:
   weight: +94
   profit: +57
 item 2:
   weight: +74
   profit: +94
=
knapsack 2:
 capacity: +190
 item 1:
   weight: +55
   profit: +20
 item 2:
   weight: +10
   profit: +19
```

In practice, the number of items is larger, but for the sake of brevity we're showing only two items.

The Knapsack problem is encoded into the MOEA framework by implementing the `Problem` interface. Without going into implementation details, we'll explain how one could introduce a problem to the MOEA framework by implementing the `Problem` interface rather ran extending the `AbstractProblem` class. For the full implementation see the manual [51].

Table 12: Implementation of the Knapsack problem by implementing the Problem interface.

```
public class Knapsack implements Problem

private int nsacks; /** The number of sacks.*/
private int nitems; /** The number of items.*/
private int[][] profit;
private int[][] weight;
private int[] capacity;

public Knapsack(File file) {...}
public Knapsack(InputStream inputStream) {...}
public Knapsack(Reader reader) {...}

private void load(Reader reader) throws IOException {...}
@Override
public void evaluate(Solution solution) {...}
@Override
public String getName() {...}
@Override
public int getNumberOfConstraints() {...}
@Override
public int getNumberOfObjectives() {...}
@Override
public int getNumberOfVariables() {...}
@Override
public Solution newSolution() {...}
@Override
public void close() {...}
```

The key points are the `newSolution` and `evaluate` methods. The `newSoluton` method creates a solution using a three argument constructor. The three argument constructor of the `Solution` class is used to define constraints. Below, in Table 13, we are defining a problem with 1 decision variable, `nsacks` - objectives and `nsacks` - constraints, one objective and one constraint for each knapsack. The second line, sets the one decision variable to be a bit string (binary encoding) for the items included in or excluded from the knapsacks, which represent the values of $d(i) \in \{0,1\}$.

Table 13: Obtaining a Solution object for the knapsack problem

```
@Override
public Solution newSolution() {
    Solution solution = new Solution(1, nsacks, nsacks);
    solution.setVariable(0, EncodingUtils.newBinary(nitems));
    return solution;
}
```

The `evaluate` method calculates the knapsack equations mentioned above. We extract the bits from the solution we are evaluating; if the bit is 1 then the corresponding item is placed in both knapsacks, if it is 0 the item is not included. After that, we sum up the profits and weights in both knapsacks and check if any of the weights exceed the capacity of the corresponding knapsack. If the weight is less than or equal to the capacity, then the constraint is satisfied and we set its value to 0, if the weight exceeds the capacity, the constraint is violated and we set its value to a non-zero (positive or negative). To reiterate, we know that constraints equal to zero are satisfied, those non-equal to zero are violated. The last two lines, set the objective and constraint values, respectively. Note that objective values are negated, this is because we are trying to maximize the values, but the MOEA framework works only on minimization problems.

Table 14: Evaluating the values for the knapsack equations.

```
@Override
public void evaluate(Solution solution)
{ boolean[] d = EncodingUtils.getBinary(solution.getVariable(0));
  double[] f = new double[nsacks];
  double[] g = new double[nsacks];
  // calculate the profits and weights for the knapsacks
  for (int i = 0; i < nitems; i++){
     if (d[i]){ for (int j = 0; j < nsacks; j++)
                 {f[j] += profit[j][i]; g[j] += weight[j][i]; }}
  }
  // check if any weights exceed the capacities
  for (int j = 0; j < nsacks; j++){
     if (g[j] <= capacity[j]){g[j] = 0.0;}
     else{g[j] = g[j] - capacity[j];}
  }
  // negate the objectives since Knapsack is maximization
  solution.setObjectives(Vector.negate(f));
  solution.setConstraints(g);
}
```

Now that we have defined the Knapsack problem, we can use the MOEA framework to solve it. Unlike the `Kursawe` class defined in the previous section, the `Knapsack` class has constructors that require parameters. In the code snippet below, we show how these parameters can be passed to a `Knapsack` object.

Table 15: Solving the Knapsack problem using the NSGA-II algorithm.

```
NondominatedPopulation result = new Executor()
      .withProblemClass(Knapsack.class, new File("knapsack.5.2"))
      .withAlgorithm("NSGAII")
      .withMaxEvaluations(50000)
      .run();
```

Note that in this case the `withProblemClass` method takes two parameters: `Knapsack.class` and `new File("knapsack.5.2")`. The second parameter (supposing we have a text file names `knapsack.5.2`) is a parameter that will be passed on to the `Knapsack` object.

Now, from the `result` object we can get the solutions and print them out, as shown below.

Table 16: Printing the solutions of the Knapsack problem.

```
for (int i = 0; i < result.size(); i++)
{ Solution solution = result.get(i);
  double[] objectives = solution.getObjectives();
    // negate objectives to return them to their maximized form
  objectives = Vector.negate(objectives);
  System.out.println("Solution " + (i + 1) + ":");
  System.out.println(" Sack 1 Profit: " + objectives[0]);
  System.out.println(" Sack 2 Profit: " + objectives[1]);
  System.out.println(" Binary String: " + solution.getVariable(0));
}
```

3.5. MOEA Framework Γ Repository

We will use the LaunchPad Eclipse plugin for developing the repository. For details about this tool see the Tool Support LaunchPad Eclipse Plugin section.

In the previous section we showed how to use the MOEA framework for solving multi-objective optimization problems, and how to introduce new problems to the framework. All this was accomplished using pure Java language. Now, we will approach the problem in a different way - using combinatory logic synthesis. The process of developing combinators is incremental; the first step is to create as few combinators as possible to generate the desired code, afterwards we continue breaking it down into smaller parts, which presumably are more generic than those in the previous step.

Let's have a look at the Knapsack problem. In the pure Java implementation it has two classes: `Knapsack.java` and `KnapsackExample.java`. The figure below (Figure 33), shows the class diagram for the Knapsack problem.

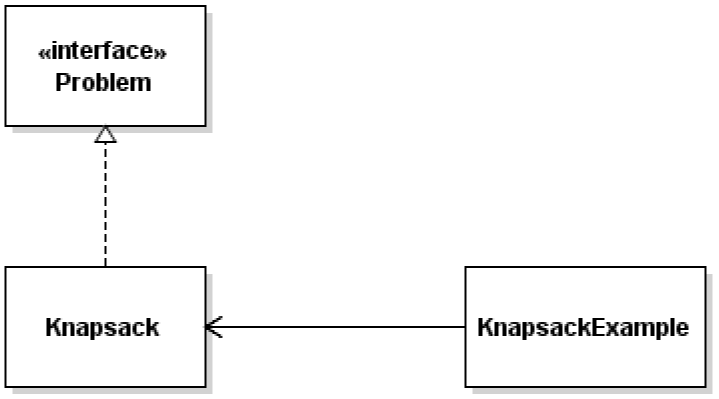


Figure 33: The class diagram for the Knapsack problem.

The `Knapsack` class is the implementation of the Knapsack problem explained in the previous section, see Table 12. The `KnapsackExample` class contains the code for solving the Knapsack problem with MOEA framework and printing the solutions, see Table 15 and Table 16.

Initially, we place the whole implementation code of `Knapsack` into one combinator; we do the same for the `KnapsackExample` class. The initial model for the repository is very simple, it defines three components: *Problem, Knapsack and KnapsackExample (Executor).*

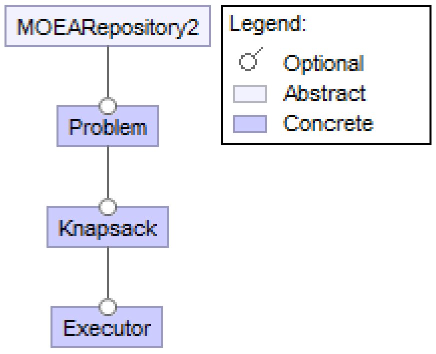


Figure 34: The initial model of the repository for the MOEA framework.

Our first version has only two combinators: **Knapsack.comb** and **Executor.comb**. The LaunchPad plugin defines a macro-language on top of L2, which is more user-friendly and human readable than the L2 language. The table below shows the important parts of the combinators mentioned above, the java code is omitted for the sake of brevity.

Table 17: The combinators for the Knapsack problem defined in LauchPad

```
type KnapsackProblem {
        [alhpa.problemType, problem];
}
implementation KnapsackProblem (knapsack/Knapsack.java) {
    // Java code for the Knapsack class
}
```

```
type Executor {
        [alpha.problemType, executor];
}
implementation Executor (knapsack/KnapsackExample.java) {
    // Java code for the KnapsackExample class
}
```

The table below shows the same combinators defined in pure L2 language.

Table 18: The combinators for the Knapsack problem defined in L2.

```
KnapsackProblem : #[alhpa.problemType, problem]
KnapsackProblem : {box ["=======knapsack/Knapsack.java=======
     // Java code for the Knapsack class
"] }
```

```
Executor : #[alpha.problemType, executor]
Executor : {box ["=======knapsack/KnapsackExample.java=======
      // Java code for the KnapsackExample class
"]}
```

Two more things that we need at this point to make it work are the **.inhab** file and **.type** file; the **.inhab** file contains the target for solving the inhabitation problem, the **.type** file contains the definition of the problem types. The code below, shows our **.inhab** and **.type** files for the Knapsack problem, respectively.

```
target {
  [knapsack, problem];
  [knapsack, executor];
},
problemType ~> knapsack
```

The difference between these two syntaxes (Table 17 and Table 18) doesn't seem so big, but things get more bizarre as we start using the parameters, function tables etc. Then, using the L2 language becomes much more difficult. Now, let's start breaking the code down into more combinators, then the difference becomes more obvious. Note that, in Table 17, when we write the implementation for the **KnapsackProblem** and **Executor** combinators, we are hard-coding the name of the files these classes will be saved to; consequently, whenever we use, let's say the **KnapsackProblem** combinator, the implementation code will be saved to a file named *Knapsack.java*. We do the same with the package name. To parameterize the class/file name and the package name, we create two combinators, **ProblemName** and **PackageName**, respectively. Now, the list of our combinators looks like in the table below.

Table 19: The list of combinators with parameterized problem-name and package-name.

```
define {
        [knapsack, packageName]                PackageName -> TwoKnapsacks;
}
```

```
define {
        [knapsack, problemName]                ProblemName -> Knapsack;
}
```

```
type KnapsackProblem {
        ProblemName                     [alpha.problemType, problemName];
        PackageName                     [alpha.problemType, packageName];
                                        [alhpa.problemType, problem];
}
implementation KnapsackProblem (<PackageName>/<ProblemName>.java) {
   package <PackageName>;
   //the list of imports needed
   public class <ProblemName> implements Problem
   {
     public <ProblemName>(InputStream inputStream) throws IOException
     {
            this(new InputStreamReader(inputStream));
     }
     //The rest of the code for the Knapsack class
   }
}
```

```
type Executor {
        ProblemName                     [alpha.problemType, problemName];
        PackageName                     [alpha.problemType, packageName];
                                        [alpha.problemType, executor];
}
implementation Executor (<PackageName>/KnapsackExample.java) {
        package <PackageName>;
        //Java code for the KnapsackExample class
}
```

The table below shows the **KnapsacProblem** combinator in L2. Now it's very obvious that it is much harder to read the combinator in L2 than it is in the macro language defined and used in LaunchPad. This is true, also due to some hacks used in L2, for example the 7 equal signs (they are used to have the content of a combinator saved into a file).

Table 20: The KnapsackProblem combinator in L2.

KnapsackProblem : { λ**var1**.{**letbox** *ProblemName* = {**var1**} in { λ**var2**.{**letbox** *PackageName* = {**var2**} in { **box** ["=======" *PackageName* "/" *ProblemName* ".java======= package " *PackageName* "=======+" *PackageName* "/" *ProblemName* ".java=======; //imports public class " *ProblemName* "=======+" *PackageName* "/" *ProblemName* ".java======= implements Problem { //Java code } "] }}}}}

This way we go on building our repository of components/combinators, which later will be used in solving other variations of already introduced problems or completely new problems. This is an incremental process, we make one step at a time and verify that it's correct. Every time we create a combinator and prove its correctness, we add it to the repository, consequently enrich our repository with new features.

We continued the process of building up the repository for the MOEA framework, by adding new combinators for problems, algorithms, and breaking down the initial combinators that we had created for the Knapsack problem into smaller and more generic ones. The picture below shows the structure of the model with dozens of combinators.

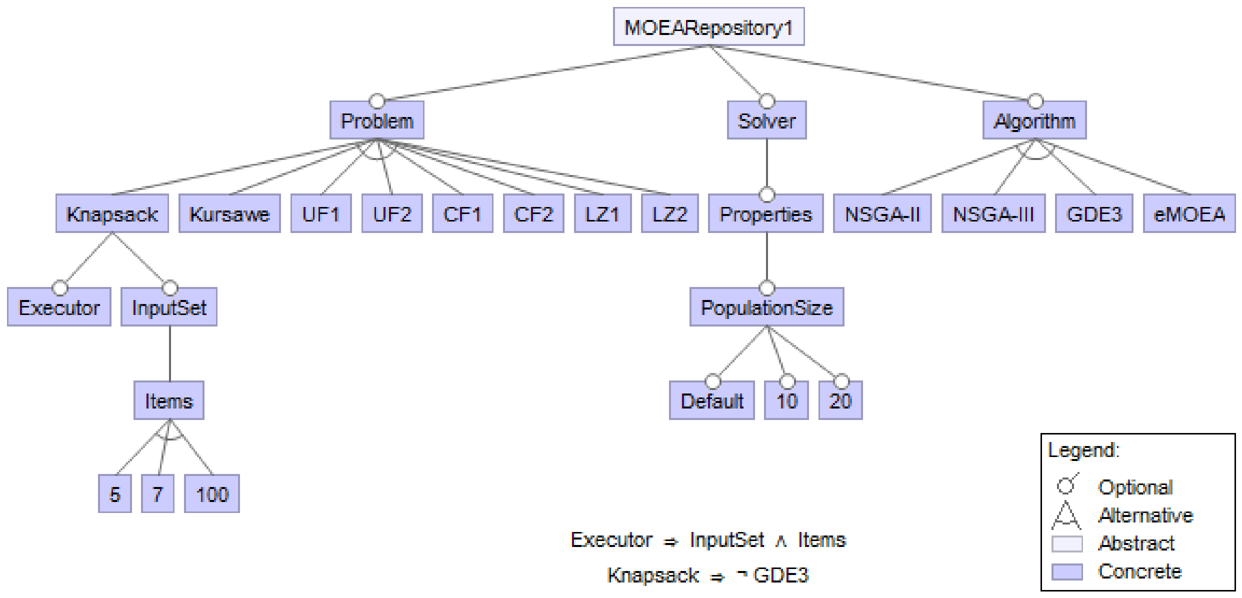


Figure 35: The model for the MOEA repository

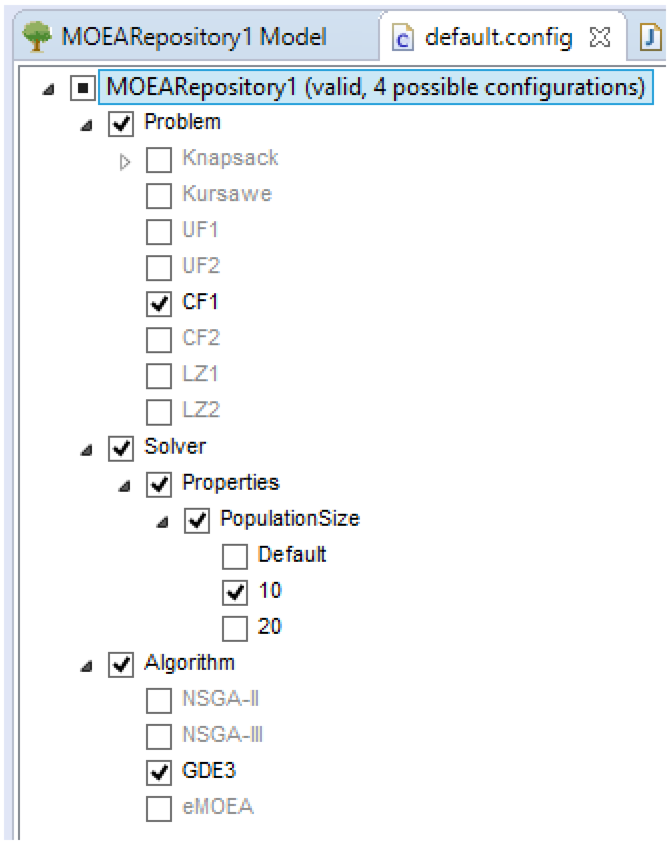
The three main branches of the repository are: Problem, Solver and Algorithm. Under the Problem component we put all the problems that we encode into the framework, and those that are already provided by the framework. The Solver component is used to make the necessary configuration for solving a problem, and create the executable file. The Algorithm component contains the algorithms provided by the framework, which can be chosen over the configuration process in the Solver component. Figure 36 shows a possible configuration.

FeatureIDE allows us to have restrictions on the model of the repository, for example feature selection under the Problem component is limited to one, which means we cannot select two problems, let's say UF1 and LZ2. We can have different kinds of restrictions in the model. There are two other restrictions in this model; shown in the middle bottom in Figure 35.

Restriction 1: $Executor \Rightarrow InputSet \wedge Items$ (read: Executor *implies* InputSet *and* Items). This forces to select the component IputSet and Items once the Executor is selected.

Restriction 2: $Knapsack \Rightarrow \neg GDE3$ (read: Knapsack *implies not* GDE3). This forces to make the selection of GDE3 algorithm disabled if the Knapsack problem is selected.

Restrictions may be necessary to prevent problems arising from different sources: design decisions, violation of domain rules, lack of expressibility etc.


MOEARepository1 Model | default.config

- [■] MOEARepository1 (valid, 4 possible configurations)
  - [✓] Problem
    - [ ] Knapsack
    - [ ] Kursawe
    - [ ] UF1
    - [ ] UF2
    - [✓] CF1
    - [ ] CF2
    - [ ] LZ1
    - [ ] LZ2
  - [✓] Solver
    - [✓] Properties
      - [✓] PopulationSize
        - [ ] Default
        - [✓] 10
        - [ ] 20
  - [✓] Algorithm
    - [ ] NSGA-II
    - [ ] NSGA-III
    - [✓] GDE3
    - [ ] eMOEA

Figure 36: The config file for solving the CF1 problem using the GDE3 algorithm.

Design Decisions – Consider the restriction 1 mentioned above. The Executor combinator is used to solve the Knapsack problem and it reads the input data from a file. We chose to model the input set as a separate combinator from the Executor, therefore whenever we need to solve a Knapsack problem we need an input set, hence the restriction to avoid problems that arise due to missing input data.

Domain rules – The MOEA framework offers many algorithms for solving different optimization problems, but not all optimization problems can be solved with all algorithms. For example, the Knapsack problem can be solved with NSGA-II but cannot be solved with the GDE3 algorithm; if we try to use the GDE3 algorithm to solve this problem we get the error message "***unsupported decision variable type***". To prevent this scenario from happening, we put the restriction 2 mentioned above.

Lack of expressibility - We'll have a closer look at the Solver component of our model presented in Figure 35. It is used to create a class with a main method, where the multi-objective optimization problems can be solved (as described in Section 3.4). It configures the Executor class by setting the required parameters and then runs it to provide us with the solution(s).

Table 21: Solver.comb - The Solver combinator

```
type Solver {
        PackageName         [alpha.problemType, packageName];
        ProblemName         [alpha.problemType, problemName];
        AlgorithmName       [alpha.algoType, algorithm];
        SolverConfig        [alpha.problemType, alpha.algoType, config];
                            [alhpa.problemType, alpha.algoType, problem];
}
implementation Solver
(<PackageName>/<ProblemName>_<AlgorithmName>Solver.java) {
package <PackageName>;
import org.moeaframework.Executor;
import org.moeaframework.core.NondominatedPopulation;
import org.moeaframework.core.Solution;
public class <ProblemName>_<AlgorithmName>Solver
{
    public static void main(String[] args)
    {
        <SolverConfig>
        // display the results
        System.out.format("Objective1  Objective2%n");
        for (Solution solution : result)
        {
            System.out.format("%.4f      %.4f%n",
                    solution.getObjective(0),
                    solution.getObjective(1));
        }
    }
}}
```

The first four lines are the parameters for the Solver combinator. Each parameter is characterized by its name and its intersection type.

PackageName	**[alpha.problemType, packageName]**;
ProblemName	**[alpha.problemType, problemName]**;
AlgorithmName	**[alpha.algoType, algorithm]**;
SolverConfig	**[alpha.problemType, alpha.algoType, config]**;

The first row defines the parameter named *PackageName*, which is of the type $alpha.problemType \cap packageName$. In the same way the three other parameters are defined. After the parameter declaration, comes the intersection type of the combinator, which in this case is $alpha.problemType \cap alpha.algoType \cap problem$. The result of the Solver combinator, after it has been evaluated, is set to be saved to a file. This is done by specifying the filename inside parentheses in the implementation part of the combinator.

implementation Solver (<*PackageName*>/<*ProblemName*>_<*AlgorithmName*>Solver.java)

After the combinator is completely evaluated, which means all the parameters are replaced with their corresponding implementation code, the result is saved to a file named exactly as the evaluated string inside the parentheses after the '/' symbol. The string before the '/' symbol specifies the package (folder) where the file is saved. As we can see from the combinator above (Table 21), combinators can have as parameters other combinators, and those combinators may have as parameters other combinators and so on. While trying to evaluate the combinators, namely finding inhabitants, if more than one solution exists, then the first one found will be returned.

The first three parameters, *PackageName, ProblemName and AlgorithmName*, are very simple combinators, indeed the simplest we can create. Table 22 shows the code for these three combinators. Here we use a shorter form of defining and implementing a combinator, which is made possible by means of the **define** keyword (see LaunchPad for more details).

Table 22: PackageName, ProblemName and AlgorithmName combinators

define {
[alpha.problemType, packageName] *PackageName* -> solver;
}

define {
[uf1, problemName] *ProblemName* -> UF1;
}

define {
[nsgaiii, algorithm] *AlgorithmName* -> NSGAIII;
}

The first row defines a combinator which is nothing more than the string "solver", in the same way the second and the third define combinators with the string values "UF1" and "NSGAIII", respectively. Deciding which problem to solve using which algorithm is a matter of changing a string, for example *ProblemName -> CF1, AlgorithName -> NSGAII.* Furthermore, the FeatureIDE tool makes it even easier through the feature selection tool shown in Figure 36.

A little bit more involved is the fourth parameter, *SolverConfig*, which defines a combinator that among other parameters it expects the *ProblemName* and *AlgorithmName* parameters. Table 23 shows the *SolverConfig, Properties* and *PopulationSize* combinators.

Table 23: SolverConfig.comb, Porperties.comb, PopulationSize.comb

type SolverConfig {
ProblemName **[alpha.problemType, problemName]**;
AlgorithmName **[alpha.algoType, algorithm];**

```
        Properties                  [alpha.problemType, properties];
                                    [alpha.problemType, alpha.algoType, config];
}
implementation SolverConfig {
      NondominatedPopulation result = new Executor()
                    .withProblem("<ProblemName>")
                    .withAlgorithm("<AlgorithmName>")
                    .withMaxEvaluations(10000)
                    <Properties>
                    .distributeOnAllCores()
                    .run();
}
```

Now, let's suppose we want generate to solver classes that solve the built-in problem UF1 using two different algorithms, let's say, NSGA-II and GDE3. Based on the **Solver** combinator, after running the tool, we should have two classes named:

1. `solver.UF1_NSGAIISolver.java`, and
2. `solver.UF1_GDE3Solver.java`

and this is exactly what is produced. But, we should also have the corresponding code in each class, shown in Table 23, set properly to solve the UF1 problem with both NSGA-II in the first class, and GDE3 in the second class. Unfortunately, both classes contain the same code as shown below:

```
NondominatedPopulation result = new Executor()
              .withProblem("UF1")
              .withAlgorithm("NSGAII")
              .withMaxEvaluations(10000)
              .withProperty("populationSize",10)
              .distributeOnAllCores().run();
```

What we need here is a way of passing the same *ProblemName* and *AlgorithmName* parameters to both **Sovler** and **SolverConfig** combinators. A possible notation, when using the *SolverConfig* parameter inside the **Solver** combinator, would be:

<SolverConfig <ProblemName, AlgorithmName>>

Since the tool does not support such a functionality, we set the restriction on choosing the algorithm, so what we can choose only one algorithm. This way we avoid the undesired scenario described above.

A possible hack to make it work

Consider these two combinators, **Solver** and **SolverConfig**, shown in Table 21 and Table 23, respectively; and the configuration in which two algorithms are selected. They both use the *ProblemName* and *AlgorithmName* parameters, and yet when evaluated, Solver produces two solutions whereas SolverConfig only one. The latter case happens because both of the selected algorithms satisfy the type condition, they are both of the same type as the *AlgorithmName* parameter is, namely [alpha.algoType, algorithm], and the tool that solves the inhabitation problem returns only one solution. Nevertheless, there's a way around it, and that's precisely what the Solver combinator does: force the intermediate solutions to be dumped into a file.

We can change the **SolverConfig** combinator so that it will be saved into a file, and also the **Solver** combinator so that it uses the class created by the SolverConfig combinator. The table below shows the changes made to the combinators from the previous version.

Table 24: Modified combinators: SolverConfig, Solver

```
type SolverConfig {
        PackageName          [alpha.problemType, packageName];
        ProblemName          [alpha.problemType, problemName];
        AlgorithmName        [selected, algorithm];
        Properties           [alpha.problemType, properties];
                             [alpha.problemType, selected, config];
}
implementation SolverConfig
(<PackageName>/<ProblemName>_<AlgorithmName>Config.java) {
package <PackageName>;
import org.moeaframework.Executor;
import org.moeaframework.core.NondominatedPopulation;
public class <ProblemName>_<AlgorithmName>Config {
  public static NondominatedPopulation execute() {
      NondominatedPopulation result = new Executor()
                 .withProblem("<ProblemName>")
                 .withAlgorithm("<AlgorithmName>")
                 .withMaxEvaluations(10000)
                 <Properties>
                 .distributeOnAllCores().run();
      return result;
 }}
```

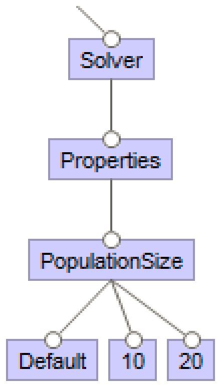
```
type Solver {
        PackageName          [alpha.problemType, packageName];
        ProblemName          [alpha.problemType, problemName];
        AlgorithmName        [selected, algorithm];
        SolverConfig         [alpha.problemType, selected, config];
                             [alhpa.problemType, selected, problem];
}
implementation Solver
```

```
(<PackageName>/<ProblemName>_<AlgorithmName>Solver.java) {
package <PackageName>;
import org.moeaframework.core.NondominatedPopulation;
import org.moeaframework.core.Solution;
public class <ProblemName>_<AlgorithmName>Solver{
   public static void main(String[] args){
     NondominatedPopulation result =
                    <ProblemName>_<AlgorithmName>Config.execute();
      System.out.format("Objective1  Objective2%n");
      for (Solution solution : result){
         System.out.format("%.4f      %.4f%n",
                              solution.getObjective(0),
                              solution.getObjective(1));
      }
   }
}}
```

This way, the SolverConfig combinator creates two classes which configure the Executor for solving a problem using two different algorithms. Then, the Solver combinator invokes the `execute()` method of these classes to solve the problem and print the solutions.

Default values

Recall from the MOEA Framework section that if we don't set parameters on an **Executor** object, it uses default values as they're set in the framework. One of the parameters that we use is 'populationSize'; if we want to change its default value we have to explicitly set it using the `withProperty` method, like in the code snippet below.



```
NondominatedPopulation result = new Executor()
          .withProblem("UF1")
          .withAlgorithm("NSGAII")
          .withMaxEvaluations(10000)
          .withProperty("populationSize", 10)
         .distributeOnAllCores().run();
```

In our model, shown in Figure 35 (a part of that picture is shown here on the left), we have created combinators for the population size. Note that we have added a combinator named 'Default', which is basically an empty combinator, as shown below.

type PopulationSize {
  [alpha.problemType, populationSize];
}
implementation PopulationSize {}

When we select the 'Default' feature (instead of 10 or 20) the above code will not contain the line '.withProperty("populationSize", ...)', which means the **Executor** object will have the default value for the 'populationSize' parameter. The *PopulationSize* combinator is a parameter for the *Properties* combinator. If we don't select a *PopulationSize*, the *Properties* combinator won't receive an empty value for this parameter, but it won't be generated at all. So the only way to pass an empty value, is by creating an empty combinator. Otherwise, we can set any population value that we want, for example 10. The code snippet below shows a combinator which sets the 'populationSize' to be 10.

```
type PopulationSize {
   [alpha.problemType, populationSize];
}
implementation PopulationSize {
.withProperty("populationSize",10) }
```

Types and subtypes

Consider the following combinators:

```
type Properties {
        Population                 [alpha.problemType, properties, populationSize];
        SbxRate                    [alpha.problemType, properties, sbxRate];
                                   [alpha.problemType, properties];
}

implementation Properties {<Population>
  <SbxRate>}
```

```
type PopulationSize {
        [alpha.problemType, properties, populationSize];
}

implementation PopulationSize {
.withProperty("populationSize",10)}
```

```
type SbxRate {
        [alpha.problemType, properties, sbxRate];
}

implementation SbxRate {
.withProperty("sbx.rate",0.8)}
```

```
type TestComb{
        Properties           [alpha.problemType, properties];
                             [alpha.problemType, selected, config];
}
implementation TestComb (Test.java)
{
  import org.moeaframework.Executor;
    import org.moeaframework.core.NondominatedPopulation;
    public class Test
   {   public static NondominatedPopulation execute() {
      NondominatedPopulation result = new Executor()
                 .withProblem("UF1")
                 .withAlgorithm("NSGAII")
                 .withMaxEvaluations(10000)
                 <Properties>
                 .distributeOnAllCores()
                 .run();
      return result;
        }
   }
}
```

To keep it brief we are using a simplified version of the SolverConfig combinator, which we are naming **TestComb**.

TestComb expects a Properties parameter, Properties expects two other parameters: PopulationSize and SbxRate.

PopulationSize is replaced by "**.withProperty("populationSize",10)**"; SbxRate is replaced by "**.withProperty("sbx.rate",0.8)**"; Properties concatenates these two; and finally TestComb uses Properties to configure the parameters for the Executor object.

It's easy to get fooled and expect the TestComb combinator to produce the following code:

Table 25: this is what we want

```
…
NondominatedPopulation result = new Executor()
                 .withProblem("UF1")
                 .withAlgorithm("NSGAII")
                 .withMaxEvaluations(10000)
                 .withProperty("populationSize", 10)
                 .withProperty("sbx.rate", 0.8)
                 .distributeOnAllCores()
                 .run();
…
```

As a matter of fact, it produces this code (Table 26):

Table 26: this is what we get

```
…
NondominatedPopulation result = new Executor()
                .withProblem("UF1")
                .withAlgorithm("NSGAII")
                .withMaxEvaluations(10000)
                .withProperty("sbx.rate", 0.8)
                .distributeOnAllCores()
                .run();
…
```

Where is the PopulationSize combinator? Is seems like a defect in the tool, but this is because of the type - subtype relationship inferred from intersection types. Let's have a look at the intersection types of all these combinators. The table below shows the intersection type of each combinator and its name.

Combinator	**Intersection type**
Properties	[alpha.problemType, properties]
PopulationSize	[alpha.problemType, properties, populationSize]
SbxRate	[alpha.problemType, properties, sbxRate]

As we can see from the table above the PopulationSize and SbxRate combinators are subtypes of the Properties combinator, because they contain the intersection type of Properties, which is $alpha.problemType \cap properties$, plus their own distinctive type *populationSize* and *sbxRate*, respectively.

The type definition of the TestComb combinator is:

```
type TestComb{
        Properties        [alpha.problemType, properties];
                          [alpha.problemType, selected, config];
}
```

It expects a parameter named *Properties* of the type $alpha.problemType \cap properties$. The name of the parameter is not important - in this case it's *Properties*, but it can be anything - the intersection type is what defines which combinators fit the bill. In this case there are three: Properties, PopulationSize and SbxRate; because the inhabitation problem is nondeterministic, it returns the first one found, and it just happens to be the SbxRate combinator.

To get the desired code generated, all we have to do is change the intersection type of the PopulationSize and SbxRate combinators, so that they are no more subtypes of the Properties combinator, and modify the parameters of the Properties combinator to match with the intersection type of PopulationSize and SbxRate. The table below shows a possible solution.

Combiantor	**Intersection type**
PopulationSize	[alpha.problemType, populationSize]
SbxRate	[alpha.problemType, sbxRate]

type Properties {
Population
SbxRate
 	
}
	
	[alpha.problemType, populationSize];
	[alpha.problemType, sbxRate];
	[alpha.problemType, properties];

Now, only the Properties combinator will fit the bill for TestComb's parameter, then from Properties, PopulationSize and SbxRate get evaluated, thus producing the desired code shown in Table 25.

Conclusion

After several iterations, we have developed a baseline of components that can be used in introducing new problems to the framework and extending it. The picture below (Figure 37) shows the model of the repository at this point.

Using the repository it's very easy now to solve a new problem, let's say UF4 (which is not in the repository). All we have to do is add a new feature to the model, name it UF4, and create a combinator of the intersection type [alpha.problemName, problemName]. After we have added this combinator, namely the string which represents a built-in problem in the MOEA framework, we can configure the Executor by choosing the features we want, we don't have to write anymore code in this case, and all the necessary code will be automatically generated.

Introducing a new problem to the framework requires to write code, but only the code that is specific to that particular problem, the boilerplate code is not necessary to be written or copy pasted from other classes, it can be encoded in the combinators and just reused any time we need. Moreover, we have encoded even the order of the steps, necessary to solve a problem, so that someone who wants to solve a problem does not need to worry about how to configure the Executor, whether the order of method calls is correct or not, that part is being taken care of in the repository.

This is very helpful in the case of high variability, like the MOEA framework. We have to write the code only for that part that varies from the other version of the same problem, the rest of the code can be generated using the previously created combinators.

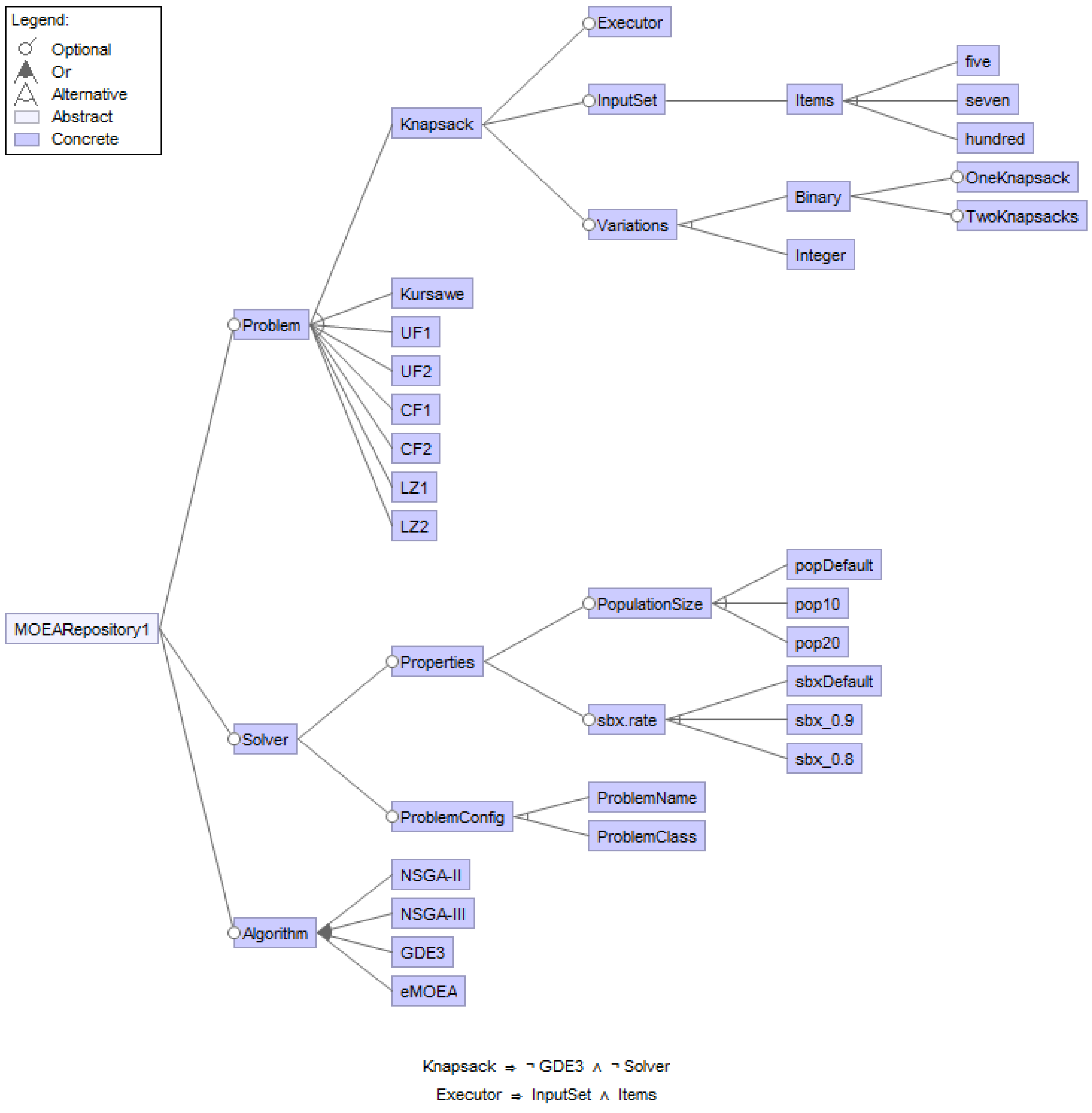


Figure 37: The final model of the MOEA framework Γ repository

4. Related Work

Throughout this thesis, there are two major problems that we are dealing with: extending a framework and generating members of a software product line. More specifically, we try to convert a framework extension problem into a configuration problem, which is the core activity in developing products of a product line family. This chapter consists of two sections, which describe the work that is related to the problems we explore in this thesis.

4.1. Product line literature on configuration

The approach to developing a software product line rather than products as separate software, is desirable because it maximizes the reuse of software components, which are common for virtually all members of the product line family. Developing a product, then, becomes more of a configuration than development task; only a small portion of the code which is specific for that product needs to be written. However, composing a product from core assets [30], components that form the basis for a software product line, is not a straightforward task (for details on benefits and costs see [31]). In large product lines, managing the components (core assets) is a challenge in its own. Moreover, just as any software system changes over time, product lines do as well, rendering the maintenance of components even harder.

Components in a software product line are generic, and usually depend on each other. The relationship between components is complex, since they are designed to support many products in the product line family and not only a single product. Selecting components, requires knowledge about the relationships and dependencies between them, and is error prone as it's possible to create invalid configurations by not choosing necessary components or choosing those that conflict each-other. Methodologies and tools that support the maintenance of components are necessary to facilitate or make possible the development of product line members.

Krebs et al [32] introduce a methodology which combines the research areas of software product families and model-based configuration in order to fill the gap in between. "*This methodology is based on a configuration model that represents functionality and variability provided by the product family*" [32].

White et al [33], on their paper published in 2008, report on three contributions towards debugging configurations of feature models: "*(1) a technique for transforming a flawed feature model configuration into a Constraint Satisfaction Problem (CSP) and show how a constraint solver can derive the minimal set of feature selection changes to fix an invalid configuration, (2) how this diagnosis CSP can automatically resolve conflicts between configuration participant decisions, and (3) experiment results that evaluate this technique*". They claim that this technique scales to models with several thousand features.

Kroon [34], in his thesis, proposes a layered approach to configuration management. It is an approach that allows step-by-step adoption of product lines and is capable of handling distinct phases of Software Product Line Engineering. He also offers a preliminary tool that works with the proposed layered approach.

Salinesi et al [35] propose an approach that combines configuration and recommendation techniques to help the process of selecting features, and they call it interactive configuration. It aims at providing the customers with the necessary information in real time about features which may be: desirable, possible or unattainable according to their choices.

The papers presented above comprise just a small portion of the whole research effort being put on product line configuration. During our literature research we targeted only some papers on software product line configuration, whereas product line configuration, in general, is a much broader field, including: software, car industry, electronics etc.

4.2. Documentation of OO frameworks

Domain knowledge is encoded into a framework, and the design is abstract, because it's not a complete software application; it is meant to be extended by implementing extra classes to complete the application. Usually the flexibility provided by a framework is not all needed by the application being developed, since applications are much more specific than frameworks. Therefore, documentation is essential to explain the behavior of a framework. It must provide the necessary information for a programmer to start using the framework.

We have conducted a research on OO framework documentation on three different repository hosting services: **SourceForge, GitHub** and **Google Code.** In this section, we describe the process of researching, and we report findings related to OO framework documentation. While doing the research, we looked specifically for the documentation entities listed below:

a. tutorial
b. design documents written by humans
c. generated document (Doxygen, Javadoc etc.)
d. video(s)
e. screenshot(s)
f. wiki(s)
g. text files only
h. code snippets
i. the code itself only

We wanted to see if there's any correlation between any sort of documentation and the popularity or usage of a framework. Basically, we want to find out what is it that makes a framework live long and catch on.

4.2.1. SourceForge (www.sourceforge.com)

SourceForge provides download statistics for each project. We took note of *total downloads* and *last week's downloads* to find out the usage and popularity of a framework.

Search keyword: "framework", date: Tuesday, January 27, 2015

Results: 93 pages, 25 results per page, total = 93 x 25 = 2325

The result list was sorted by relevance and frameworks were picked based on the number of downloads in the prior week (relative to the search date provided above). We picked a framework for review if it was downloaded at least once over the prior week. Out of 2325 frameworks, we selected 30 and categorized them based on their domain.

Here's the list of chosen frameworks and the type of documentation they provide. The boxes that contain 'y' show that that type of documentation is provided by framework developers.

Table 27: The list of the selected frameworks from sourceforge.

										Downloads		
Framework	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**Total**	**Last week**	**Domain**
Hibernate	y	y	y		y	y		y		9,127,992	10,675	Database
BIRT Report Designer	y	y		y	y					6,970	342	Reporting & Data Visualization
Code::Blocks	y	y			y	y		y		11,430,642	70,281	IDE
Liferay Portal	y	y	y		y	y		y		13,470,610	50,096	Business & Enterprise
Win32++	y	y	y		y			y		51,299	67	Development
Spring Framework	y	y	y	y	y	y		y		3,845,066	457	Database, Enterprise
SW Test Automation Framework	y	y			y			y		791,042	1,335	Testing Automation
C Unit Testing		y	y		y			y		185,814	445	Testing
Hcon Security Testing	y	y			y					56,886	574	Testing
CppCMS C++ Web	y	y	y					y		54,509	583	Web Development

MOEA	y	y	y		y			y		16,393	138	Artificial Intelligence, Mathematics
WebSploit									y	78,218	249	Networking/Web
Logging Framework For C++			y			y		y		161,099	414	Logging
openLCA Framework	y	y		y	y					33,640	92	Simulation
Logging Framework For C							y			92,310	338	Logging
MLT Multimedia	y	y	y	y				y		86,144	136	Multimedia
TreeFrog	y	y	y	y	y			y		10,284	59	Web Development
Java Neural Network Framework Neuroph	y	y	y		y					169,123	702	Machine Learning, Simulation
OWLNext: C++ Application Framework	y	y	y		y	y		y		24,300	57	GUI Widget
Marvin Image Processing	y	y	y	y	y			y		32,103	124	Image Processing
Genode OS Framework	y	y			y			y		13,562	43	OS Security
EPICS Qt	y	y			y			y		5,544	41	Development, Scientific, Engineering
DMF: Distributed Multiplatform Framework	y	y			y					11,498	291	Modeling
Evolutility	y	y			y			y		55,991	80	Development, Database
TRAK Enterprise Architecture Framework		y			y	y				6,051	34	Enterprise
ProM - Framework for Process Mining	y				y					30,721	49	Enterprise
Abbot Java GUI Testing Framework	y	y	y		y			y		112,345	52	Testing
PL/SQL Starter Framework									y	6,452	7	Database
Moqui	y	y	y					y		3,833	6	Enterprise
SAFS	y	y	y	y	y			y		74,600	42	Testing Automation

4.2.2. Github (www.github.com)

Unlike **SourceForge**, which provides download statistics over time, **GitHub** doesn't; it provides statistics about: **pull**, **fork** and **star**.

Pull request: "*Pull requests let you tell others about changes you've pushed to a repository on GitHub.*" [36]

Fork: "*A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.*" [37]

Star: "*Starring a repository allows you to keep track of projects that you find interesting, even if you aren't associated with the project. When you star a repository, you're actually performing two distinct actions:*

- Creating a bookmark for easier access

- Showing appreciation to the repository maintainer for their work" [38]

Combining the data from **pull**, **fork** and **star**, we can tell how much a framework is being used, how much it has gained popularity, how much people are contributing to further development of the framework and the like.

Search keyword: "object oriented framework", date: Friday, January 30, 2015

Results: 323 repositories

The result list is sorted by "most stars". Initially, we picked top 10 results. After that, we applied the "Java" filter, and we picked top 10 java frameworks, out of 17. Finally, we applied the "JavaScript" filter, and we picked top 5 frameworks (excluding those that were picked up during the first round). Out of 323 frameworks, we selected 25 for review. Here's the list of the selected frameworks.

Table 28: The list of the selected frameworks from Github.

Framework	a	b	c	d	e	f	g	h	i	star	fork	pull	Domain
oocss	y	y			y	y		y		4,462	616	22	Web Development
micromvc							y			618	130	4	Web Development
Simple.Web	y				y	y		y		200	62	4	Web Development
Objective-Chain		y						y		199	8	2	Development
rails_script	y	y						y		128	7	0	Web Development
polymode	y				y			y		111	13	1	Development

skull							y			96	6	0	OS
onphp-framework									y	72	45	16	Web Development
xp-framework	y	y	y			y		y		36	28	9	Development
Jive Selenium Pages Framework			y					y		3	2	0	Testing
aGOOF									y	2	0	0	Gaming
Flow3 Netbeans Plugin									y	1	0	0	Development
Anasy									y	1	0	0	Syntax Analysis
jauk									y	1	0	0	Parsing
oo-webdriver									y	1	1	0	Testing
tell-me									y	1	0	0	Development
meteor									y	1	1	0	Data Management
groops									y	1	0	0	Simulation
tangram									y	1	0	0	Web Development
Joov									y	0	0	0	VXML
UIZE JavaScript Framework	y	y			y			y		41	13	0	Web Development
easejs	y	y						y		35	3	0	Web Development
MuLego UI			y							34	5	0	GUI Widget
Simple Game Framework									y	24	2	0	Gaming
Romano									y	18	0	0	Gaming

4.2.3. Google Code (https://code.google.com)

Search keyword: “object oriented framework”, date: Saturday, January 31, 2015

Results: 584 frameworks

Many projects have been moved to GitHub and are not being maintained any longer in Google Code. We selected top 20 frameworks which have not been moved to GitHub or any other host. Google Code does not provide any download statistics or other usage indicators, except how many people have starred a project. Thus we could only record the number of people that have starred a project. The concept of "starring" in Google Code is same as in GitHub (for details have a look at the **GitHub** section).

Table 29: The list of the selected frameworks from Google Code.

Framework	a	b	c	d	e	f	g	h	i	star	Domain
mdanalysis	y	y			y	y		y		75	Simulation
mid3d						y				311	Mobile Development
php-reader		y				y		y		114	Web Development
TibiaAPI						y		y		52	Development
PHP Form Builder Class	y				y			y		261	Web Development
Lua for Windows	y	y						y		471	OS
OSCATS						y				17	Testing
mybatis.NET	y	y						y		113	Database
GAPI						y		y		552	Web Development
ease-bat-php									y	2	Web Development
axiospace									y	2	Web Development
Nuwani IRC Platform						y				21	Web Development
emo-framework	y	y		y	y	y		y		77	Gaming
make-it-easy						y		y		83	Testing
tamy	y	y				y		y		3	Gaming
Jease	y	y			y			y		18	Web Development
WebSite-PHP	y	y	y	y	y	y		y		3	Web Development
ieUnit	y			y				y		14	Testing
TangramCOM	y							y		1	Networking
QuickDB	y	y	y	y	y	y		y		26	Database

It's obvious from the tables presented above that frameworks, which are considered to be successful, provide code snippets along with user generated design documents and other forms of documentation, i.e. screenshots, videos, APIs etc. We define a framework as successful, if it has been downloaded recently by a considerable number of users, Hibernate for instance - over 10k downloads during the last week of the research. In GitHub or Google Code, the number of downloads would be equivalent to "star". This is a good indication that the framework is being used and/or extended, which makes it successful in terms of usability.

After having gathered considerable information on OO framework documentation, we shortlisted three frameworks, which had enough documentation, tutorials and help, to conduct our second case study. Our three candidates for the second case study are: Hibernate, MOEA and Marvin Image Processing framework.

- **Hibernate**

Hibernate is an Object/Relational Mapper tool. It's very popular among Java applications and implements the Java Persistence API. Hibernate ORM enables developers to more easily write applications whose data outlives the application process. As an Object/Relational Mapping (ORM) framework, Hibernate is concerned with data persistence as it applies to relational databases (via JDBC). [39]

- **MOEA** (Multi Objective Evolutionary Algorithms)

MOEA is an open source java framework for developing and experimenting with multi-objective evolutionary algorithms, and other general purpose multi-objective optimization algorithms. It provides many ready-to-use algorithms, and offers the possibility of defining new problems which then can be solved using these algorithms. In addition, it provides the tools necessary to design, develop, execute and statistically test optimization algorithms. The documentation is very thorough and provides complete examples of defining new problems and solving them using the MOSA's algorithms. [27]

- **Marvin Image Processing**

Marvin is a framework that provides features for image and frame manipulation, image analysis, filtering and multi-threaded image processing. All features are provided as plugins, which are run at runtime using reflection. The framework can be extended by developing new plugins, and it provides documentation and examples on how to develop a plugin. [40]

Among these three candidates, MOEA seems to best fit the bill for our purpose. It has a high variability, where different problems can be encoded into the framework, each problem can have many variations. In these kinds of domains, combinators show their power of reusability. Therefore, we decided to experiment with the MOEA framework.

5. EVALUATION

This project is exploratory in nature and so the first contribution was a proof of concept towards using an available Combinatory Logic tool [21] to generate non-trivial software applications. We have developed a repository of combinators of the KS product line and demonstrated the ability to generate a couple of variations.

To ensure that this work is applicable to other software frameworks, after having searched the Internet for open source OO frameworks and characterized them by the mechanisms and artifacts they provide to explain how one should extend the framework, we have selected one for which we have developed a Γ repository for generating extensions in that framework. This part of the overall effort demonstrates that our approach can work on multiple frameworks.

And finally, we present a set of basic metrics which are supposed to help us evaluate Γ repositories.

5.1. EVALUATION OF KS AND MOEA Γ REPOSITORIES

By developing the Γ repositories for KS and MOEA frameworks, we have demonstrated how a framework extension problem can be converted into a configuration problem. Using this approach we make it possible for a framework designer to encode the abstractions necessary for extending the framework in question and make a complete reuse of the code which is repeated across different variations.

To better explain the power of abstraction encoding and code reuse, we will use a combinator from the KS framework repository (see Figure 38). For the sake of brevity, we are using a very simple combinator to show how a very basic requirement, as a result of a design decision during the framework development, can be encoded in a combinator. Referring to Figure 38, **DeckController** is responsible for handling a mouse-press action on the deck. Obviously, to be able to handle a mouse-press on the deck, we need to provide a class which extends the `SolitaireReleaseAdapter` class. Besides providing the code for the `mousePressed(MouseEvent m)` method, there are other things we need to take care of, otherwise nothing will work; and they are: call the constructor of the superclass, and after the mouse press action had been handled (whatever needs to happen after clicking on the deck) call the `refreshWidgets()` method of the `Solitaire` class. There is no way of enforcing this in Java. The only way to convey this information to framework extenders, is by providing documentation and perhaps code snippets that do a similar thing.

```
type DeckController {
        NameParameter        [alpha.gameType, namerule];
        DeckPressed          [alpha.gameType, deckPressed];
                             [alpha.gameType, deckController];
}
implementation DeckController (<NameParameter>/DeckController.java) {
package <NameParameter>;
import java.awt.event.MouseEvent;
import ks.common.view.*;
import ks.common.model.*;
import ks.common.controller.*;
public class DeckController extends SolitaireReleasedAdapter {
      protected <NameParameter> theGame;
      public DeckController(<NameParameter> theGame) {
            super(theGame);
            this.theGame = theGame;
      }
      // Deal cards
      public void mousePressed(MouseEvent me) {
            Move m;
            // Action on press
            <DeckPressed>
            //have solitaire game refresh widgets that were affected
            theGame.refreshWidgets();
}}}
```

Figure 38: Example of reuse and abstraction encoding using combinator

The **DeckController** combinator makes sure that all the required actions are taken, and lets the programmer focus on developing the **DeckPressed** combinator – the code that handles a mouse-press action on the deck. Another advantage that comes along is the reusability. Note that, we don't have to retype the `refreshWidgets()` line (or the other lines of Java code) whenever we develop a new Solitaire variation, they're encoded in this combinator and get generated by the CLS tool. In a real life application the framework itself evolves, and a very common scenario is when other steps are necessary to be taken, let's say we have to call a method before <**DeckPressed**>, and this happens after we have developed dozens of variations. In a pure object-oriented implementation we have to modify each variation[2], and update the documentation so that other programmers who develop new variations follow the modified necessary steps. Whereas using the CLS approach, all we need to do is modify the **DeckController** combinator and rerun the tool.

[2] One could use the Aspect Oriented Programming (AOP) to make the changes in all the variations, but in AOP it is much more difficult to predict and control the effects of modifications, it is not a type-safe system, they may affect the wrong parts.

When introducing Combinatory Logic Synthesis in Section 2.1, we mentioned that L2-combinators are higher-order polymorphic function acting on L1-arguments. Explaining this concept is best done by an example. Figure 39 shows a combinator that will synthesize various move classes that involve cards moving from a source stack to a destination stack.

```
type Move {
        Designate     [alpha.gameType, stack, movename];
        Helper        [alpha.gameType, stack, helper];
        Valid         [alpha.gameType, stack, valid];
        Do            [alpha.gameType, stack, do];
        Undo          [alpha.gameType, stack, undo];
        NameRule      [alpha.gameType, namerule];
                      [alpha.gameType, stack, stackFrom, stackTo];
}
implementation Move (<NameRule>/<Designate>.java) {
        package <NameRule>;
        import ks.common.model.*;
        import ks.common.games.*;
        public class <Designate> extends Move {
        Stack source;
        Stack destination;
        public <Designate>(Stack from, Stack to) {
              super();
              this.source = from;
              this.destination = to;
        }
        public boolean undo(Solitaire game) {
              <Undo>
              return true;
        }
        public boolean doMove(Solitaire game) {
              if (!valid (game)) { return false; }
              <Do>
              return true;
        }
        public boolean valid(Solitaire game) {
              <Valid>
              return false;
        }
}
```

Figure 39: Solitaire polymorphic combinator

This combinator synthesizes a class named **Designate** (a string value bound to an input parameter) and stores the generated code in a package **NameRule**. When exercised multiple times during the inhabitation algorithm, this combinator will create Java classes whose structure properly embodies the logic of a move. The specific logic synthesized by the **Do** and **Undo** input parameters will be inserted in their proper location in the class file. Thus instead of relying on native inheritance as supported by Java, this combinator will generate any number of classes in polymorphic fashion, each one guaranteed to be correct if properly specified.

These examples also explain why we feel CLS is well-suited for software product lines. Logic programming approaches seek to synthesize a program from a single specification [41], typically using stepwise refinement to ensure correctness with each transformation. It seems hard to explain how to conduct this process individually for each product line member; alternatively, it seems hard to explain how one could share or reuse results from each stepwise refinement across multiple product line members. By contrast, the repository development process outlined in this thesis starts by constructing a simple repository of combinators by identifying the fundamental steps necessary to produce the code (as expressed in tutorials or sample software artifacts). Iteratively over time, the repository incrementally adds the individual combinators identified as the different product line members are migrated and synthesized.

5.2. Metrics for Combinators

Based on the common practices of object-oriented metrics [42], we have defined several metrics that help us evaluate the quality of a project developed by means of combinators. By doing so, we have tried to evaluate as many aspects as possible of a Γ repository, for example the coupling factor between combinators, reusability of combinators etc.

Table 30: List of the metrics for combinators

Name	Description
NC	Number of combinators in a repository
NM	Number of unique intersection types
APTC	Average number of parameters per combinator (number of all parameters/NC)
LNFT	Number of function tables
LND	Number of 'defines' (simple combinators whose implementation contains just a String value).

All the metrics listed in the table above are L2 metrics, which means they don't take into account the implementation part. Since the implementation of a combinator, namely L1, can be any language (object-oriented, procedural etc.) or a configuration/properties file, it's very difficult, not to say impossible, to come up with generic metrics that would evaluate the combinators at the implementation level regardless of the L1 language.

Table 31 lists the values of all these metrics for our case study repositories: KombatSolitaire and MOEA.

Table 31: Values of metrics for the KS and MOEA repositories

Metric	**Value (KombatSolitaire)**	**Value (MOEA)**
NC	92	54
NM	142	33
APTC	3	1
LNFT	13	1
LND	25	27

According to [42], coupling is a measure of interdependence of two classes. For example, class **A** and **B** are coupled if a method declared in class **A** calls a method declared in class **B** or vice-versa. In our context, two combinators are considered to be coupled if one uses the other as an input parameter. Observe that two combinators, even if they don't have parameters at all, may be coupled at the implementation level. However, this is beyond the scope of the metrics defined so far, so we won't consider it as a metric for combinators, but rather point it out as a topic for future work.

The APTC metric's intension is to measure the coupling factor in a repository. It basically represents the average number of parameters per combinator in a repository. The larger the APTC's value is, the more coupled, combinators in a repository are considered, since the more parameters a combinator has the more dependent on other combinators it is.

From the data on Table 31 we can conclude that the MOEA framework has a smaller coupling factor than KombatSolitaire, which is desired (for details see [42]).

The number of combinators (NC) metric is similar to the LOC (Lines of Code) metric in object-oriented metric system. "*If a comparison is made between projects with identical functionality, those projects with fewer lines of code have superior design and require less maintenance*" [42]. Thus, the value for LOC is desired to be as low as possible. In the case of NC it is slightly different. We can compare two different repositories for the same framework. The one with a larger NC is considered to have a better design, since smaller combinators (consequently larger overall NC) are likely to be reused more easily than larger ones.

Currently, our metric system is very basic and defines a set of very simple metrics. More advanced metrics are described under the Conclusion and Future Work section.

6. Conclusion and Future Work

We have given a comprehensive report on using CLS as an alternative approach to designing and extending object oriented frameworks. We have presented and described the tools which help us realize our goals, and described in detail two case studies that help us evaluate this approach – emphasizing advantages and challenges that come along with it. However, there are many questions remaining to be answered in the future, and some of them are listed below. We conclude this thesis project with some remarks on the future work, which fall into two categories: *tool support* and *theory*.

Tool support

Currently there is no support for navigation from a feature in the model to the corresponding combinator(s) or vice-versa. The only way to find the combinators associated with a feature, is by manually finding the folder with the same name as the feature, then listing the combinators inside it. This task will be difficult to carry out in the case of large repositories; a large repository will be one that has over a thousand combinators or even more. Thus, an easy navigation will tremendously improve the usability and contribute to the success of the overall approach.

Besides navigation, the continued evaluation of the LaunchPad macro-language is key to improving the way combinators are written, detecting syntax and specifically semantic errors, enriching the language with new expressions etc.

The L2 language that we use in this project, is one of many possible higher level languages that could be used in a type-safe system. It would be interesting exploring other possible L2 languages and finding advantages and disadvantages of one over the other.

We have successfully demonstrated applying the CLS principles to a number of case studies in this thesis. Naturally the next question is to evaluate whether other programmers will have similar success. Upon the completion of CS 3733 in May 2015, Professor Heineman will coordinate a number of students in using LaunchPad to build solitaire variations in the same way that the FourteenOut variation was constructed. This experience will provide valuable feedback as we continue to evaluate the widespread applicability of the technique.

Theory

Up until now, all the case studies we have experimented with, use Java as a target (L1) language. It would be of a great interest in the future to experiment with other languages as well, which would potentially reveal the strengths and weaknesses of the tool and the overall approach, and consequently contribute to developing a more robust system.

Combinatory Logic Synthesis (CLS) is an approach to a much bigger picture than what we use it for in this thesis, and it is continuously being researched on. Thus, it is very is crucial to have the advances on CLS mapped onto LaunchPad, for a more complete overview of the idea of synthesizing code using CLS. In addition, the L2 language will likely change in future releases of the InhabConsoleClient tool chain, and LaunchPad will adjust accordingly.

Evaluation is a very important aspect in developing qualitative software, and we have defined several metrics to address this issue. Nevertheless, there are still many remaining evaluation questions, which require metrics that capture the behavior of combinators and the relationship between them, and give more meaningful answers that help better evaluate the project. Such metrics would take into consideration not only the definition but the implementation part of combinators too. Another metric that would be interesting to consider in the future, is similar to the *Depth of Inheritance Tree* [42] metric defined for object-oriented programming. It basically defines the depth of subtyping of the intersection types. Let's say we have the intersection types defined as below:

1. [a, b]
2. [a, b, c]
3. [a, b, c, d]

The intersection type (3) is a subtype of (2), and (2) is a subtype of (1), therefore (3) is a subtype of (1), too. In this case the depth of subtyping is 2, since from (3) we can go two levels up in the tree of intersection types.

References

[1] C. W. Krueger, "Software Reuse," *ACM Computing Surveys (CSUR),* vol. 24, no. 2, pp. 131-183, 1992.

[2] J. Bosch, C. Szyperski and W. Weck, "Component Oriented Programming," in *Object-Oriented Technology. ECOOP 2003 Workshop Reader*, Darmstadt, Springer, 2004, pp. 34-49.

[3] G. T. Heineman and W. T. Councill, Component-based software engineering, Vasteras: Springer, 2001.

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns: elements of reusable object-oriented software, Pearson Education, 1994.

[5] G. Butler, "Object Oriented Frameworks," in *15th European Conference on Object-Oriented Programming, Tutorial*, Budapest, 2001.

[6] "Software Product Lines," Software Engineering Institute, Carnegie Mellon University, [Online]. Available: http://www.sei.cmu.edu/productlines/.

[7] M. E. Fayad, D. C. Schmidt and R. E. Johnson, Building Application Frameworks: Object-Oriented Foundations of Framework Design, New York: John Wiley & Sons, Inc., 1999.

[8] S. F. Allen, M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo and E. Moran, "Innovations in computational type theory using Nuprl," *Journal of Applied Logic,* vol. 4, no. 4, pp. 428-469, 2006.

[9] P. Saxena, N. Menezes, P. Cocchini and D. A. Kirkpatrick, "The scaling challenge: can correct-by-construction design help?," in *Proceedings of the 2003 international symposium on Physical design*, Monterey, 2003.

[10] B. Dion, "Correct-By-Construction Methods for the Development of Safety-Critical Applications," *SAE Technical paper,* 2004.

[11] R. L. Constable, "Robert Constable on correct-by-construction programming," Machine Intelligence Research Institute (MIRI), [Online]. Available: https://intelligence.org/2014/03/02/bob-constable/.

[12] B. Düdder, G. T. Heineman and J. Rehof, "LaunchPad: A Synthesis Framework for Feature-rich Applications," *Unpublished work,* 2014.

[13] J. Rehof, "Towards Combinatory Logic Synthesis," in *1st International Workshop on Behavioural Types, BEAT*, Rome, 2013.

[14] G. T. Heineman, "An Instance-Oriented Approach to Constructing Product Lines from Layers," Worcester Polytechnic Institute, Worcester, 2005.

[15] D. Batory, J. N. Sarvela and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE transactions on Software Engineering,* vol. 30, no. 6, pp. 355-371, 2004.

[16] J. Rehof and M. Y. Vardi, "Design and Synthesis from Components (Dagstuhl Seminar 14232)," *Dagstuhl Reports,* vol. 4, no. 6, pp. 29-47, 2014.

[17] J. R. Hindley and J. P. Seldin, Lambda-calculus and Combinators: an Introduction, 2nd ed., vol. 13, Cambridge: Cambridge University Press, 2008.

[18] H. Barendregt, M. Coppo and M. Dezani-Ciancaglini, "A Filter Lambda Model and the Completeness of Type Assignment," *The journal of symbolic logic,* vol. 48, no. 04, pp. 931-940, 1983.

[19] B. Düdder, J. Rehof and M. Martens, "Staged Composition Synthesis," in *Programming Languages and Systems*, Grenoble, Springer, 2014, pp. 67-86.

[20] R. Davies and F. Pfenning, "A Modal Analysis of Staged Computation," *Journal of the ACM (JACM),* vol. 48, no. 3, pp. 555-604, 2001.

[21] J. Bessai, A. Dudenhefner, B. Düdder, M. Martens and J. Rehof, "Combinatory logic synthesizer," in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, Corfu, Springer, 2014, pp. 26-40.

[22] B. Düdder, "Automatic Synthesis of Component & Connector Software Architectures with Bounded Combinatory Logic. PhD Diss," Dortmund, 2014.

[23] C. Kastner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Weilgorz and S. Apel, "FeatureIDE: A tool framework for feature-oriented software development," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, Vancouver, 2009.

[24] G. T. Heineman, A. Hoxha, B. Düdder and J. Rehof, "Migrating Object-Oriented Frameworks to Enable Synthesis of Product Line Members," *ACM. To appear,* 2015.

[25] T. Leich, S. Apel, L. Marnitz and G. Saake, "Tool Support for Feature-Oriented Software Development: FeatureIDE: An Eclipse-Based Approach," in *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, San Diego, 2005.

[26] "FeatureIDE," University of Magdeburg, [Online]. Available: http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/.

[27] "MOEA (Multi Objective Evolutionary Algorithms)," 2010. [Online]. Available: http://www.moeaframework.org/index.html.

[28] "Knapsack Problem - Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Knapsack_problem.

[29] "Systems Optimization," Institut TIK - Zurich, [Online]. Available: http://www.tik.ee.ethz.ch/sop/download/supplementary/testProblemSuite/.

[30] "A note on terminology," Software Engineering Institute, Carnegie Mellon, [Online]. Available: http://www.sei.cmu.edu/productlines/frame_report/terminology.htm.

[31] "Benefits and costs of a product line," Software Engineering Institute, Carnegie Mellon, [Online]. Available: http://www.sei.cmu.edu/productlines/frame_report/benefits.costs.htm.

[32] T. Krebs, K. Wolter and L. Hotz, "Model-based Configuration Support for Product Derivation in Software Product Families," *Mass Customization, Concepts-Tools-Realization, GITO-Verlag,* pp. 279-292, 2005.

[33] J. White, D. C. Schmidt, D. Benavides, P. Trinidad and A. Ruiz-Cortes, "Automated diagnosis of product-line configuration errors in feature models," in *Software Product Line Conference, 2008. SPLC'08. 12th International*, Limerick, 2008.

[34] E. Kroon, "Layered configuration management for software product lines. MS Thesis," University of Twente, 2009.

[35] C. Salinesi, R. Triki and R. Mazo, "Combining configuration and recommendation to define an interactive product line configuration approach," *arXiv preprint arXiv:1206.2520,* 2012.

[36] "GitHub - help - pull request," [Online]. Available: https://help.github.com/articles/using-pull-requests/.

[37] "GitHub - Help - fork," [Online]. Available: https://help.github.com/articles/fork-a-repo/.

[38] "GitHub - Help - star," [Online]. Available: https://help.github.com/articles/about-stars/.

[39] "Hibernate," 2001. [Online]. Available: http://hibernate.org/.

[40] "Marvin Image Processing Framework," 2008. [Online]. Available: http://marvinproject.sourceforge.net/en/index.html.

[41] Y. Deville and K.-K. Lau, "Logic program synthesis," *The Journal of Logic Programming,* vol. 19, no. 20, pp. 321-350, 1994.

[42] "An Introduction to Object-Oriented Metrics," [Online]. Available: http://agile.csc.ncsu.edu/SEMaterials/OOMetrics.htm.