Miguel Carvalho Pires

Licenciado em Engenharia Informática

Incremental Compilation and Deployment for OutSystems Platform

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : João Costa Seco, Professor Auxiliar, FCT/UNL

Co-orientador : Lúcio Ferrão, Principal Software Engineer, OutSystems

Júri:

Presidente: Prof. José Augusto Legatheaux Martins

Arguente: Prof. Salvador Pinto Abreu

Vogal: Prof. João Costa Seco

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2014

Incremental Compilation and Deployment for OutSystems Platform

Acknowledgements

I could not carry such hard but rewarding journey until the end if it was not the support and the valuable contribution of some people. I hope I did not forget anyone.

A want to express my sincere gratitude for my supervisors Lucio Ferrão, from OutSystems, and João Costa Seco, from *Faculdade de Ciências e Tecnologia de Lisboa (FCT)*. Thanks for your guidance. Thanks for the patience and the interest with which you helped me to communicate better and to be more critical with my own work. Thanks for your reviewing and critical observations.

I want to thank *Faculdade de Ciências e Tecnologia de Lisboa (FCT)* for giving me the opportunity of work in such intellectually engaging environment that is *OutSystems R&D team*, and for the monetary support.

A very special thanks to Ricardo Soeiro, the team leader of the pipeline team. I thank you for your guidance and valuable support. I thank you for all the insightful discussions we had, which helped me to make sense of the problem I was tackling. Without you this work would not have been possible.

Finally, I want to thank my friends and family.

To my father, who did everything that was at his reach to help me being a better prepared person. To my stepmother and my grandmother, for your support and love.

To my friends, Nuno Costa, Nuno Cruz, Hugo Cabrita, and Daniel Santos. Thank you all, for your companionship and support, for raising my spirits at those moments when things seemed more deary and daunting.

All errors and mistakes in this dissertation are my fault alone.

Abstract

OutSystems Platform is used to develop, deploy, and maintain enterprise web an mobile web applications. Applications are developed through a visual domain specific language, in an integrated development environment, and compiled to a standard stack of web technologies. In the platform's core, there is a compiler and a deployment service that transform the visual model into a running web application.

As applications grow, compilation and deployment times increase as well, impacting the developer's productivity. In the previous model, a full application was the only compilation and deployment unit. When the developer published an application, even if he only changed a very small aspect of it, the application would be fully compiled and deployed.

Our goal is to reduce compilation and deployment times for the most common use case, in which the developer performs small changes to an application before compiling and deploying it. We modified the OutSystems Platform to support a new incremental compilation and deployment model that reuses previous computations as much as possible in order to improve performance.

In our approach, the full application is broken down into smaller compilation and deployment units, increasing what can be cached and reused. We also observed that this finer model would benefit from a parallel execution model. Hereby, we created a task driven Scheduler that executes compilation and deployment tasks in parallel. Our benchmarks show a substantial improvement of the compilation and deployment process times for the aforementioned development scenario.

Keywords: Incremental Deployment, Incremental Compiler, Deployment pipeline, OutSystems, Large Projects

Resumo

A plataforma OutSystems é usada para o desenvolvimento, *deploying* e manutenção de applicações *web* empresariais e móveis. As aplicações são desenvolvidas através de uma linguagem visual de domínio específico, em um ambiente integrado de desenvolvimento, e são compiladas numa pilha convencional de tecnologias *web*. Na plataforma, existe um compilador e um serviço de *deployment* que são responáveis pela transformação do modelo visual numa applicação *web* funcional.

Com o crescimento de uma aplicação, os seus tempo de compilação e *deployment* também aumentam, o que afecta a produtividade do programador. No modelo anterior, a aplicação era a única unidade de compilação e *deployment*. Quando uma aplicação era publicada, ainda que o programador tivesse realizado uma alteração de muito pequena dimensão, a aplicação seria sujeita a um processo completo de compilação e *deployment*.

O nosso objectivo é reduzir os tempos de compilação e *deployment* para o caso de uso mais comum, em que o programador efectua pequenas mudanças numa aplicação antes despoletar a sua compilação e *deployment*. Nós modificámos a plataforma OutSystems para suportar um novo modelo de compilação e *deployment* incremental que reutiliza resultados de publicações antecedentes, de forma a reduzir processamentos redundantes e consequentementemente os tempos de espera.

Na nossa abordagem, a modelo de aplicação é partido em unidades de compilação e *deployment* mais pequenas, aumentando, assim, o que pode ser aproveitado por publicações posteriores. Observou-se, também, que este modelo mais fino benificiaria de um modelo de execução paralelo. Nesse sentido, criou-se uma unidade de execução de tarefas que escalona as tarefas de compilação e *deployment* tirando partido paralelismo. As nossas métricas revelam uma redução substancial dos tempos de compilação e *deployment*, para os cenários acima mencionados.

Palavras-chave: *Deployment* incremental, Compilação Incremental, Deployment Pipeline, OutSystems, Projectos de grande dimensão

List of Figures

Contents

1

Introduction

OutSystems is a company with a single product, the *OutSystems Platform*. The platform is used to develop standard enterprise web applications or mobile web applications that are scalable, easy to maintain and easy to change. The developer designs applications on an integrated development environment, on the top of a proprietary visual domain language. An application is compiled to a web application that runs over a standard web technology stack.

1.1 Motivation

Over the last years, the applications developed with the platform grew in complexity and number. Such growth exposed the compiler and deployment limits, as the compilation and deployment times reached uncomfortable levels. Large applications take a significant amount of time to compile, which affects negatively the developer's productivity. Our goal with this project is to identify the inefficiencies of the compilation process and propose a incremental compilation model that reduces compilation times.

Lets consider a scenario where Dave, a seasoned developer, is working on a supplier management web application. The current task on his backlog is to implement a interface that displays a table that lists the supply contracts celebrated with a given supplier. Requirements dictate that the table must contain a column for the customer's name along the dates in which the contract is valid. In this table, contracts are identified by an integer, that figures in the first column and if it is clicked on, shows a more descriptive view of that contract. Dave implements this interface and the underlying logic, and deploys the application in order to test what he has just changed. Despite the simplicity of these changes, the supplier management application is very large, and the platform takes about

3 minutes to compile and deploy it.

Compilation is an event that disrupts Dave's workflow, since it breaks his cognitive flow, forcing him to temporarily switch his attention from the problem he is working on, to the output produced by the compiler. This leads Dave to postpone the compilation process as much as possible.

1.2 Dissertation Context

This is a proposal for a master dissertation, that is being carried out in the context of *OutSystems Research and Development Team (R&D)*, together with *Faculdade de Ciências e Tecnologia de Lisboa (FCT)*.

OutSystems platform contains an integrated development environment (IDE) that has been developed in the last 13 years, and currently comprises than 1.9 million lines of code.

The platform is used to develop typical enterprise web applications connected to an SQL database. Easy to learn, easy to change, and scalability, are the three core values of the platform. Development is made under an integrated environment, using a visual domain specific language that covers all the aspects of a standard web application, including the data model definition, the business logic, the user interface, and the integration with other systems.

1.3 Problem Identification

In the last years, the applications developed on the top of the platform have become bigger and more complex, and their compilation times increased as well. Reducing compilation time has become a priority. This is not, however, a easy goal, for the process that accomplishes the compilation and deployment of the applications is a complex pipeline that currently has got 320 thousand lines of code.

The pipeline consists in three phases: *Code Generation*, *Compilation*, and *Deployment*. In prior work, the *OutSystems R&D* team optimized some parts of the process to use incremental strategies, achieving substantial gains in its efficiency (about 40% faster). The other phases, however, were not so optimized.

The problem is that the application as a whole is currently the only Deployment Unit. Consequently, even a superficial change on an already deployed application, triggers a full compilation and deployment, that does not reuse work performed in previous runs. Our goal is towards a more granular model where parts of application can be compiled and deployed separately using incremental mechanisms.

1.4 Goals

With this work, we intent to optimize the compilation and deployment process so that developers can see the effects of their application changes as fast as possible, even in large projects. In order to do so, we attack the problem identified in the previous subsection, by decomposing it into the following subgoals:

1. Break down an application into smaller deployment units;
2. Propose and implement an incremental compilation and deployment model;
3. Design an solution that has minimal impact in the existing compiler and deployment code base.

1.5 Document Organization

The rest of the document is structured as follows:

Chapter 2: Before we tackled the problem we have in hands, we had made some research about akin problems and challenges, both in the industrial and the academic context. This chapter is dedicated to the synthesis of our research.

Chapter 3: The purpose of this chapter is to provide all the context that is necessary to understand the problem and the proposed solution. Here, we introduce the platform, we describe the pipeline and we finally identify the main problems with it, guided by metrics, that not only regard the pipeline process, but also the development patterns.

Chapter 4: In the chapter, we describe our proposed model, and justify our choices.

Chapter 5: We detail implementation aspects and describe what was needed to change on the former pipeline implementation in order to leverage the proposed model.

Chapter 6: In order to demonstrate the improvements yielded by our new model, we performed some benchmarks. The chapter is dedicated to the discussion of those measurements.

Chapter 7: We make a retrospective of all the work that was accomplished and we look at the key insights in our implementation.

2

Related Work

In this chapter we describe topics related to out core theme, which is partial and incremental compilation of an application. We first describe and help understand how programming language mechanisms can improve the process of code compilation. We describe some module mechanisms present in programming languages, and argue about the properties they convey into the (partial) compilation of an application.

We also describe how compiler related tools tackle the problem of efficiently compiling fragments of programs, the so called compilation units. We describe and relate our problem to the strategies of differential compilation that have been put to use in widely used tools. We considered the standard UNIX tool *Make*, the *Vesta* the *Eclipse Java Compiler*.

Our research also lead us to more generic computational approaches, namely the results in incremental computation, that inspired the core of our partial compilation model. From this type approaches, we focused on the Umut Acar's *Self-Adjusting* computation model.

2.1 Modules in Programming Languages

In a programming-in-the-large context, good programming and software engineering practices recommend the decoupling of parts of an application, and the distribution of functionality by small and manageable components. It is commonly accepted that the wise modularization of application code, as promoted by software development methodologies, improve maintenance, safety, readability, and flexibility on using third party components.

From early on, it was identified the necessity of optimizing the recompilation process,

by exploiting the capability of separate compilation, leveraged by the modularization facilities provided by the languages. [Car97]. Tools like Make would function upon the basis of the "Conventional Recompilation Rule"[Tic86], which states that a compilation unit must be recompiled whenever:

(1) the compilation unit changes, or

(1) a context changes upon which the compilation unit depends.

However, those conditions are not strong enough to minimize redundant computations. Under this rule, a module that depends on a definition whose signature did not change is unnecessary compiled, because the context it depended on changed.

A more granular model is proposed by Walter F.Tichy and Mark C.Baker[Tic86] that minimizes the set of modules to compile in recompilations. The idea is that the smart compiler computes for every pair of modules (M_a, M_b), where M_a depends on M_b, it is computed a *context* C_{ab} for module M_a that comprises all the free identifiers belonging to M_b. Whenever M_b is modified, the compiler recomputes a *change set* G_b that contains all the declarations whose signature did change relatively to the last version of the modules. The module M_a is only compiled when $C_{ab} \bigcap G_b \neq \emptyset$, i.e. , when it changes or the signature of a definition it depends changes.

C

The C language has a very simple module system. Importing a module consists in inserting the code in the file. Modules in C do not create namespaces, so name clashing occurs whenever two modules contain definitions that have the same name. Programmers typically solve this problem by prefix a definition name with the module's name. Information hiding is possible through a static annotation. A static type is internal to the module where it is defined.

Java

Packages and Classes are the primitives of the Java's module system. A Java project typically comprises a set of packages that aggregate classes in a cohesive and logical way, as defined by the developer.

In Java, a *Compilation Unit* exists under a package, comprises a set of types declarations and declares external types that it imports, possibly from other packages. A type can either be class or a interface. Compilation in Java compiles types of a *Compilation Unit* (commonly a Java file) into `class` files [GJS+13].

Before a class can be instantiated, it has to be loaded, linked an initialized [LYBB13]. Loading a class consists in searching for the `class` file correspondent to the class that is being loaded and from it extracting the `Class` object that will represent that class.

Linking takes a binary form of a class or interface type and combines it into the state of the Virtual Machine. During linking, symbolic references to other classes may be resolved, triggering the Load-Link-Initialize process for each class that is resolved. Alternatively, an Virtual Machine implementation may choose to defer resolution, resolving symbolic references only when they are needed.

Finally, in Initialization, the class's static fields are initialized and its superclass's fields are initialized too.

ML

In ML, there is a difference between open modules and closed modules. A closed module is a module which has no free terms. A module that is not close is opened. A module's signature, beside its exports, enunciates also the signatures of modules that it depend. Before a module can be used in a certain context, it has to be instantiated. Instantiation consists in replacing the free terms required by the module with concrete modules that respect the signatures.

Linking

Linking is the process that glues separate compiled modules, through their interface, into a full application.

Modules may be compiled independently but they have to be glued together somehow; the step that accomplishes this is *Linking*. During compilation, a program written in a source language is translated to a new language, while Linking combines modules, by resolving dependencies and collapsing them into an executable unit [TGS08]. However, as we'll see, linking can also happen during runtime.

Compilation and linking is an extensive subject that is handled differently by different languages. We'll reduce our scope to languages that compile to native code, such as C or OCaml. In languages that compile to machine code, modules are ultimately compiled to libraries, which can be either shared or static and whose representation depends on the underlying Operative System. When a program is linked against static libraries, an executable is created that includes both the code of the program and the library to which it is linked. Shared Libraries, on the other hand, are loaded by the operative system's linker before the program is loaded – alternatively, shared libraries can also be load at runtime through wrappers to linker provided by the system [BWC01].

2.2 Build Automation Tools

The development tools under the category of *Build Automation Tools* share a considerable amount of characteristics with the OutSystems pipeline. Their purpose is to build an application from a set of primitive compilation units; their main feature is to manage the dependencies between the different compilation units, in order to, efficiently orchestrate

the building process; they usually resort to external tools like compilers and databases to implement primitive operations such as code generation, linking, testing, and configuration. We relate our approach, that of a new model for the OutSystems compiler pipeline, with some of the more commonly used tools, and describe how they work.

2.2.1 Make

Make is a Build Automation Tool whose execution is driven by a configuration file, the makefile, where a sequence of rules describe how the different parts of a project are built [Fow90].

The basic rule mechanism is supported by the existence of target and source files. A rule, as seen in the example 1 below, is fired when there is an active dependency to it. By default, the execution of `make` starts with the target `all`.

Example 1.
```
huffman.o:   huffman.c  heap.o
             cc -Wall -std=c99 -o huffman huffman.c heap.o
```

A rule declares a sequence of dependencies (possibly empty) that if are all active trigger the rule. A dependency can be either the head of other rule or a filename. In the case of the filename, it's considered to be active if it changed since the last make's execution. Make does such by using the filesystem's metadata. When a rule is triggered, the designated system command is executed. The rule of example 1 states that target `huffman.o` is recompiled whenever `huffman.c` or `heap.o` become active. Its second line indicates which system commands have to be executed so that the target is generated. This rule language, together with the conventions of targets and sources being files in the filesystem, and using timestamps, results in a very flexible and simple to use compilation tool. Moreover, it permits granular build models that only do what is strictly needed, reusing as much as possible from previous builds. If the application is very monolithic, however, it will not benefit much of the finer build mechanisms that `make` allows.

Complex building may involve diverse tasks such as running different compilers, generating documentation, updating databases, among other activities that we left aside [Baa88]. Make is able to deal with such scenarios, because it is not sensible to the semantics of the tools and files that it manipulates, it just blindly executes a sequence of commands defined by the developer, for each unit that is assumes as changed.

Make has some disadvantages too. Stating the dependencies between compilation units is cumbersome, time consuming, and error prone. Also, `make` is not aware of the semantics of files and tools that it manipulates, therefore rules cannot considerate units finer than files. Nonetheless, it is heavily supported in the UNIX environment and its conventions, and has inspired a broad range of modern tools such as Rake, Vesta or Ant.

2.2.2 Vesta

Vesta is a software configuration management tool (SCM) targeted at the development of very large software projects[HLMY99]. This tool merges Version Control with Automatic

Building. Vesta is a complete solution that supports many aspects of the development of big projects. Vesta is an extensive tool and we only describe here the automatic building aspect where there is a significant intersection with the scope of our work.

Diagram 2.1 shows the parts of Vesta's architecture that are relevant to us. One important design decision in Vesta is that all sources are immutable, that is, every time a file is edited, a new version is created while the old one is kept.

Vesta, as well as Make, is not sensible to the semantics of the compilation units that it manipulates. Versions of sources and tools are immutable, what allows *Repeatable Builds*: any version can be rebuilt at any time in future. Building is driven by *System Models*, which are descriptions that express how parts of the project are built and how to combine those parts into a final unit; it is a more sophisticated makefile counterpart. When a tool is spawned, a cache entry is created in *Function Cache Server*, that maps the name of the tool, along with the arguments with which it was called, to the set of references that point to the artifacts that were generated. We should recall that everything is immutable in Vesta, therefore we can be sure that the files that are referenced don't change, in any circumstance.

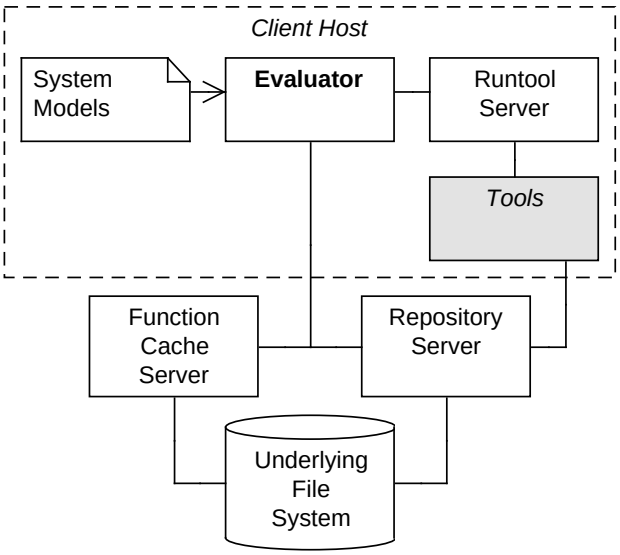


Figure 2.1: *Vesta's architecture*

A *System Model* describes how a certain application is built, and it is interpreted by the *Evaluator*, that communicates with other components in order to accomplish what is expressed in the system model. Tools are requested by the Evaluator to the *Run Tool* server, that spawns them inside an encapsulated process. Processes are encapsulated by Vesta so that file accesses to disk by those tools can be captured and dependencies subsequently inferred.

2.3 Eclipse Java Compiler

The Eclipse Java Compiler is an incremental compiler that compiles only what changed relatively to the previous compilation. The rational is that a modification of the source of the program should contribute proportionally to compilation time relatively to the extension of such modification. Naturally, a compiler that follows this model has to cache results for each unit that it compiles. This technique exploits the fact that typically between successive compilations there is a considerable amount of redundant work, unless the program was radically changed.

Eclipse JDT, a set of development tools shipped with eclipse, contains an incremental compiler, the Eclipse Compiler for *Java (ECJ)*. ECJ compiler takes the idea further: it is able to run valid fragments of source code even when the whole file doesn't compile, as long the invalid excerpt is not reachable from the fragment that is to be ran.

ECJ is based on the incremental compiler of *VirtualAge* for *JAVA* an integrated development environment for *JAVA* developed by *IBM*, but that was discontinued.

We are dealing with a compiler that has been designed and adapted to support incremental compilation, due to this being a promising path towards a faster compilation; it is, thus, of our interest to understand how other compilers achieve incremental compilation and, hopefully, adapt some of their ideas to our work.

2.4 Incremental Computation

So far, we've been analysing how some tools approach the problem of orchestrating complex build processes efficiently. The tools that we've studied were designed to a specific use case, however, it is notable that they share some characteristics: the use of dependency graphs to infer a minimal set of units to be compiled or built, and the caching of resources and their subsequent reuse. The computation model that we present follow, it is the more generalist of the models and therefore can be applied to far wider range of problems, although, we'll also see that this model articulates exactly the aforementioned notions but in a more generic form.

An incremental program aims to reduce its execution time by avoiding computations that don't depend on the changes of its input[*]. The less sensible a program is to small changes of its input, more benefits this technique brings to its running times. Two notable examples are Stylesheets and compilers [Aca09].A change of a cell in a Stylesheet shouldn't lead to the re-computation of cells whose expression doesn't have the changed cell as operand. Concerning the subject of our study, the Compiler, small changes to independent modules or isolated functions shouldn't provoke the recompilation of modules or functions that not depend on the affected units, provided that the interface remains unaltered[SA93][Tic86].

2.4.1 Self-Adjusting Computation

Self-Adjusting computation is an incremental computation model that was introduced by Umut Acar, as the theme of his dissertation for Phd, in 2005[Aca05]. An *adaptive program* minimizes what is recomputed in response to small changes of its input - relatively to the preceding execution. As an adaptive program executes, dependencies between data are captured into a *dependency graph*, which is used, in further executions, to infer what needs to be recomputed. This is the most generalist model that we've discussed so far and can be applied to a wide range of problems.

In this model, the smallest changeable unit is the *Mutable Reference*. It can be either a memory cell or an expression that uses a value that is computed from another mutable reference. Mutable References and their dependencies form a *Dynamic Dependency Graph*, which drives *changes propagation*. *Changes Propagation* is the mechanism by which changes are propagated through the graph, triggering, along its path, re-evaluation of expressions that depend on changed data and subsequently marking them as changed too.

A functional program can easily be transformed into an adaptive program, by adapting it to use a set of primitives: *mod*, *read*, *write*; and a set of meta-primitives: *init*, *change* and *propagate*[ABH01]. Any powerful enough underlying type system can enforce the correct use of those primitives [Car02]; for example, forcing the expression of a mod or a read to terminate with a write (soon we'll understand why and how). Example 2 exemplifies an instantiation of this model as an Ocaml's library.

Example 2.

```
module SelfAdjusting :
  sig
    type a' mod
    type a' dest
    type changeable

    val mod: ('a * a' -> bool) ->
         (a' dest -> changeable) ->
          a' mod
    val read: a' mod * (a' -> changeable) -> changeable
    val write: a' dest * a' -> changeable

    val init: unit -> unit
    val change: a' mod * a' -> unit
    val propagate: unit -> unit
  end
```

Types are opaque and they enforce to some extent a correct use of the library. Mutable references have type (a' mod). Write can only be applied to (a' dest) values, with obligates writes to be call inside mod and read expressions, that is, a write is made under the context of a mutable reference expression. These primitives are just functions and can be implemented in any language that supports functions as values.

Mod creates a *mutable reference*. Its first argument, whose signature is ('a * 'a -> bool), it is a comparison function that defines a conservative equality class between elements of generic type 'a; its role is testing if the reference's value, after an explicit change, was effectively changed, in other words, if the new value is really different from the previous – this avoids triggering unnecessary changes propagation. Along with that function, it also receives an initializer function that initializes the mutable reference with a value.

Read reads a value from a mutable reference, its first argument, and applies it to an expression passed as second argument. This expression has return type "changeable", suggesting that it should terminate with a *write*: unless the value of the mutable reference is ignored, an expression that reads that value becomes dependent upon the mutable reference that it refers.

Write writes a value to a mutable reference and commits a dependency between the node that is read and the node that is written. Writes only appear in the context of read expressions or mod expression.

Dependencies: They arise from the use of reads, writes and mods. As the program is evaluated, a dynamic dependency graph is constructed, as those primitives are called. An edge is added whenever a write is committed in the context of a mod or read expression. The edge's source node is the mutable reference that is read, and its incidence is the mutable reference that is written. Edges are labeled with *time spans* (t_0, t_1), where both t_i are *time stamps*; t_0 is assigned before read's expression is evaluated, and t after write expression is committed. Any *totally ordered infinite* set T defined on relation $\leq_T$ is a valid candidate to *time stamp*'s domain – It's not specified a concrete structure. We say that edge e_1 is contained in e_2 if $TS(e_1)$ is within $TS(e_2)$.

```
let x = mut (==)
  (fun m -> write(m, 2))
let y = mut (==)
  (fun m -> write(m, 3))
let z = mut (==) (fun m ->
  read y (fun valFromX ->
    read z (fun valFromY ->
      let w = valFromX + valFromY in
      write(m, w))))
```

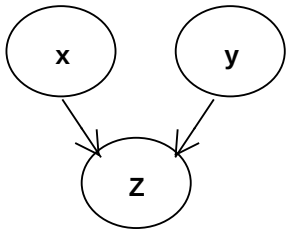


Figure 2.2: *A functional self-adjusting program and the respective dynamic dependency graph*

Example 3.

Changes propagation: A mutable expression's value is changed by calling the meta-primitive *change*, and propagations are triggered by *propagate*. During propagation, expressions that depend on changed mutable references are re-evaluated and the dependency graph is updated: dependencies may become obsolete and new dependencies may

emerge, consequence of the conditional expressions that may entail distinct call trees that depend on the input. When a certain mutable expression is recomputed, all edges that are within that expression's time span become obsolete and subsequently are removed from the graph.

In 2007, Ancar generalizes this mechanism to imperative programming, by extending the model with a new concept: traces. A trace is a sequence of reads and writes which has as target certain mutable reference, which imply a memorized value [AAB08]. Traces are comparable to multi-version mechanism in a database or persistent data structures. Basically, instead of memorizing the value of an expression, it stores the log of writes and reads that target that expression.

3

OutSystems Context

Our description of the platform is focused on the components that have a role in the publication process. As our ultimate goal is to improve the development experience, it becomes necessary to comprehend the developer's workflow as well, hence we also briefly describe what developing with the *OutSystems Platform* consists in. Finally, we provide an in-depth description of the *pipeline*, the process that compiles and deploys an application developed with the platform into a typical Web application.

An application is deployed to either one of two currently supported stacks: .NET or JAVA. Under the context of this work, the differences between the two are not significant, so we just focus on the .NET one. In the stack we used for this thesis, data is stored on MICROSOFT SQL SERVER DATABASE, server logic is leveraged by ASP.NET FRAMEWORK (using the C# programming language), and the application is hosted by INTERNET INFORMATION SERVER(IIS).

3.1 The OutSystems Platform

The *OutSytems Platfom* is an high-productivity tool used to develop Web Applications and Enterprise Web Applications. The platform offers an Integrated Development Environment, the *Service Studio*, where the developer develops, maintains and triggers the compilation and the deployment of the applications he works on. In figure 3.1 it is shown how it is to work with *Service Studio* during a typical development period. All the development is made through a *Visual Domain Specific Language* that provides graphical metaphors with which the developer defines the data model, composes user interfaces, and programs business logic. Those metaphors are the *OutSystems* language elements.

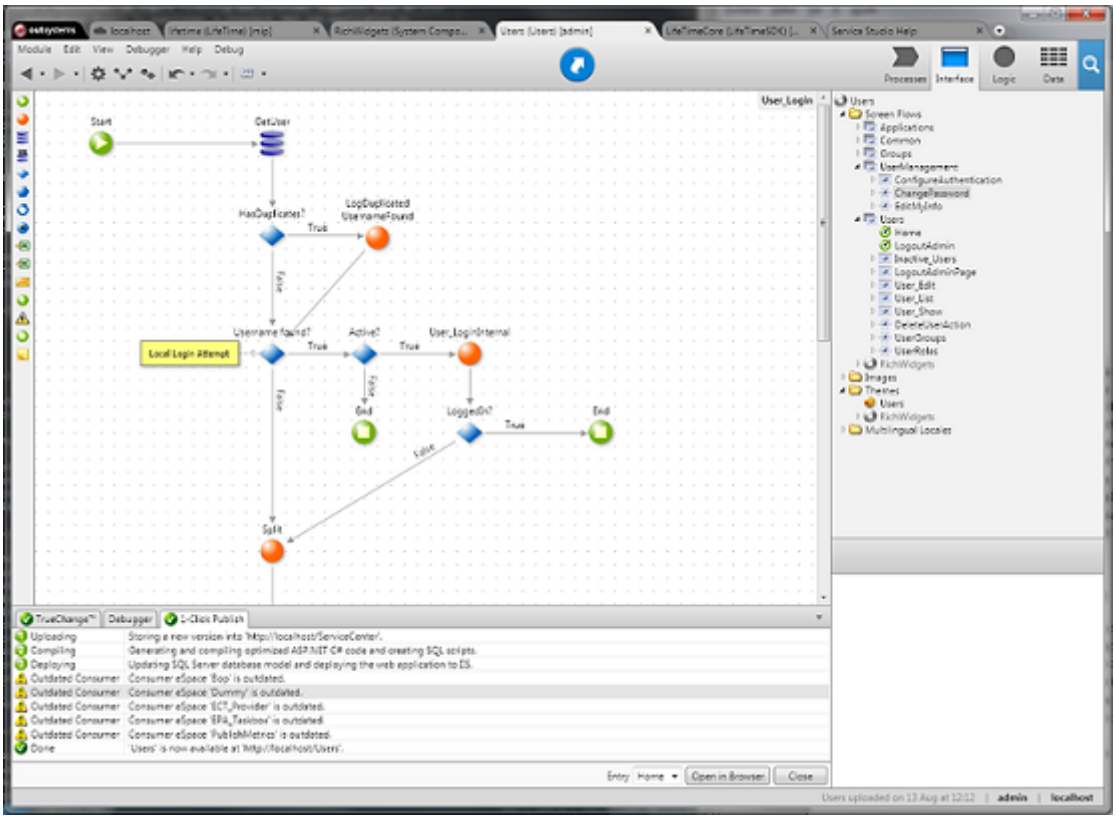


Figure 3.1: *A typical development session on Service Studio*

Despite the simplicity of developing with the *OutSytems Platform*, its language is actually very rich and extensive. Due to its dimension, it would be too overwhelming to focus on the whole language, therefore we chose to prioritize a subset of its elements, under the criterion that the ones that are most frequently changed have more relevance to the compilation times.

3.1.1 The Language Elements

The *OutSystems Platform* provides a proprietary *Visual Domain Specific Language* that allows the developer to work on all aspects of an application. The language aggregates a set concepts and metaphors that abstract the development of a application from the implementation details. To narrow the scope, we focus just a subset of those elements, justifying our choice with the developing metrics that are given in section 3.2. The elements are: *Espace*, *Action*, *Entity*, *WebScreen*, *WebBlock*, *Stylesheet*, *Structure*, *Image*, *and* Javascript.

Espace

An *Espace* may be both a running deployable application and a module. All the elements we further describe are contained in it. As a module, an *Espace* may export a set of elements which may be used by other *Espaces*. An *Espace* that imports an element is called

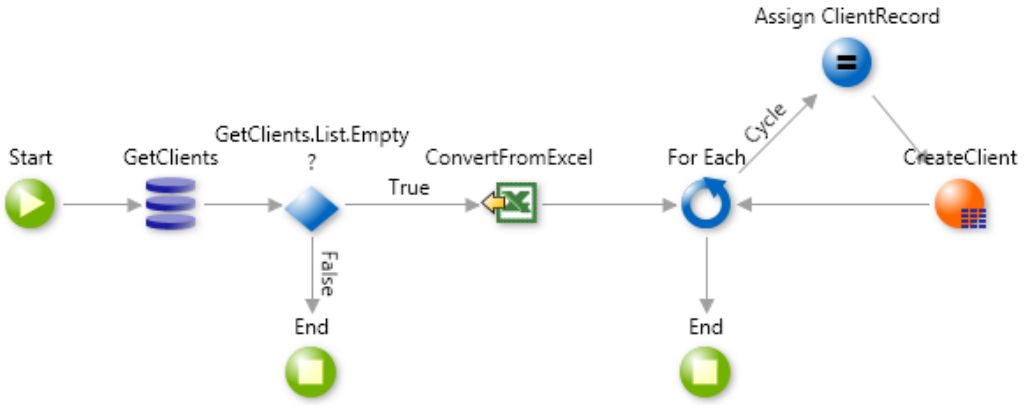


Figure 3.2: *The definition of an action*

a *Consumer*, whereas the one that provides the element is a *Producer*. Modules are used to aggregate related functionality wrapped in a pluggable interface so other systems can reuse it, which makes them an fundamental building block for more complex systems.

Currently, the Espace is only *deployment unit*.

Action

Actions are used to encode business logic, through the composition of visual elements, instead of the traditional programming languages that are text-based. Visually, an action resembles a graph, where the nodes are the action elements, and the control flow arrows are the edges.

An *Action* may be invoked from two different contexts: when some event on a screen is triggered: for instance, when a screen is loaded or when a button in a *WebScreen* is clicked on; or they may appear somewhere in the middle of some other action, as an action element itself.

Identified by a name, an *Action* defines an interface and an implementation. The interface specifies the action's inputs and outputs. Inputs are values passed to the action at its invocation. Outputs are values that the action produces and that can be used by action elements on the context where the action was called. Values can be entity instances or basic types such as text, integers, dates, etc.

Developers define actions by connecting action elements using arrows that drive the control flow. An action element is the basic building block, that may be a control structure, such as an *if* or *foreach*, action calls, queries to the database, among others.

As an example, consider the action shown in Figure 3.2. The goal of the action is to seed a database with data that is loaded from an Excel file. The execution flow always departs from element *Start* and ceases at an *End* element. When the action terminates, the execution flow continues in the context where the action was called from. In our example, when this action is triggered, an SQL query is executed that selects all clients from the

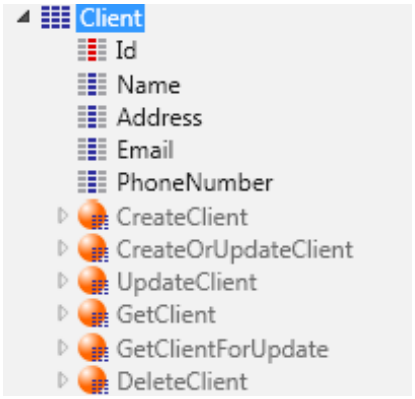


Figure 3.3: *Entity's attributes and actions*

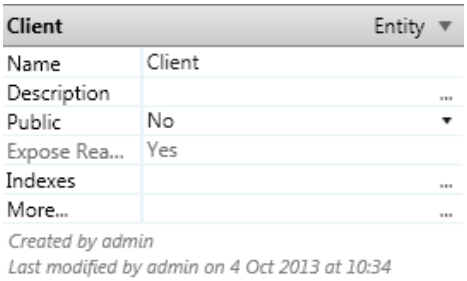


Figure 3.4: *Entity's meta-information*

database (a query element is represented by a stack of three purple cylinders). Then, it is followed by an IF element (whose icon is a losang) that checks if the list returned by the query is empty; if it is not, the action ends, otherwise, the execution continues: the Excel file is loaded. Each record in the file is iterated and inserted in the database. The orange element, labeled as "CreateClient", is an action call to one of the default actions that are automatically created for each *Entity*.

Entity

An *Entity* abstracts and encapsulates access to a database's table. It is described by a list of attributes, that correspond to database columns, and meta-data. For each defined entity, there is a set of *Actions* that perform basic CRUD (Create, Read, Update, Delete) operations over entity instances.

Web Screen

Web Screens are elements used to define dynamic web pages. Associated to a *Web Screen* there are variables, widgets and *actions*. The scope of screen local variables include the screen actions and the screen definition. Widgets are UI components that define an interface, which includes typical items like "input boxes", "buttons" or "links".

Web Block

A *Web Block* is a reusable web screen component that is used to build modular interfaces. Just like the *Web Screen*, they are composed by Web Widgets, however, they are not web pages and they do not have an autonomous existence: they either exist inside a *Web-Screen* or other *Web Block*. A *Web Block* depends on the parent component in which it is contained, which can be a *Web Screen* or a *Web Block*.

Contrary to *Web Screens*, *Web Blocks* are exportable, which means that the developer can define *Web Blocks* and share them between *Espaces*. They are a modular approach to interfaces. *Web Blocks* can also have logic associated to them by providing *Actions* that

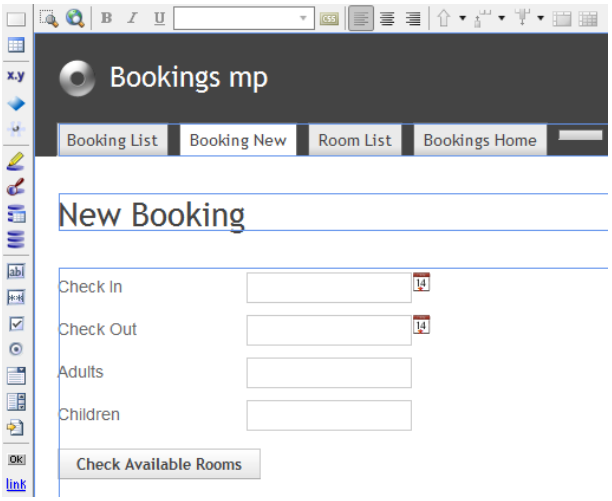


Figure 3.5: *Developer iterating a Web Screen in Service Studio*

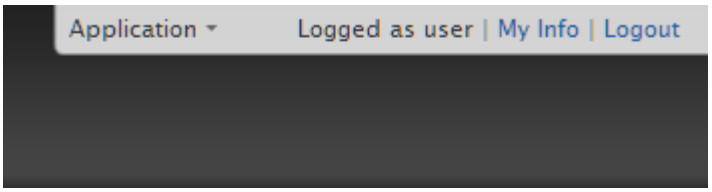


Figure 3.6: *A Web Block that modularizes the user context panel*

allow their manipulation.

Stylesheet

Cascading Style Sheets as defined by *W3C*. The following elements can have a `CSS` associated them: *Web Screens*, *Web Blocks*, *Themes*. A `CSS` can be global or local. A global `CSS` affects all UI elements of the application, while a local `CSS` affects particular elements, such as a *Web Screen* or *Web Block*.

Structure

Structures are containers that are used to store and manipulate data in memory, during an action execution, for example. A *Structure* instance is similar to an entity instance in the sense that both are composed by a set of attributes, however, contrary to the entity counterpart, a *Structure* instance is ephemeral as it only exists in memory.

Image

An *Image* is a resource. The supported file types are `png`, `jpg` and `gif`. *Images* can have three types: *static*, *external*, and *database*. *Static* images are included in the *Application Model*; database images are stored in the database, whereas external images are stored somewhere outside of the application.

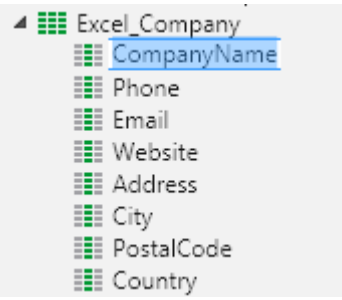


Figure 3.7: *A Structure*

Javascript

A *Javascript* is a Javascript snippet written by the developer. Typically, it is used when the developer wants to implement complex client logic that could not be implement uniquely through the facilities offered by the visual language. *Javascripts* are encoded in the application model in raw.

Other Elements

We did not consider all the *OutSystems* DSL since that would make the problem too extensive for a dissertation context. Moreover, the elements that we chose cover most of the developers workflow, as proven at the section about the *platform usage patterns*.

3.2 Developer Workflow

Understanding the user work-flow lets us to appreciate better the impact of publication times on the development experience. From previously collected metrics about the development patterns, we identify the model elements' subset that are most often changed between publications. This metrics tells what we should prioritize in order to maximize the impact on perceived publication times and consequently on developer's experience.

3.2.1 Change-Publish-Validate cycle

The Figure 3.8 illustrates the typical developer's interactive workflow, where the developer changes the application model using *Service Studio*, publishes using the development environment, and validates the results by testing the deployed application. This cyclic process goes on during development and maintenaince phases, which are basically the whole application's lifetime.

In the *OutSystems Platform*, editing and validation of the application model is performed using *Service Studio*, while code translation and optimization is the job of the, so called, *Compiler Service*. During a development session, *Service Studio* constantly validates the modification that are applied to the model, and alerts the user with error and warning messages in realtime, as shown by Figure 3.9. An *Action* call that does not agree

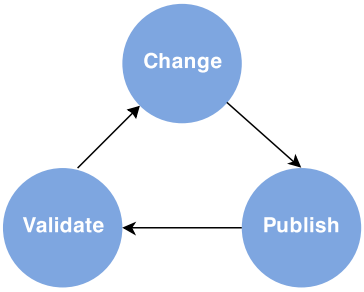


Figure 3.8: *Developer's Workflow*

TrueChange™ | Debugger

Invalid Data Type	Incompatible data types in = operator ('Computer Identifier', 'Entity Reference (Integer)').
Invalid Data Type	Incompatible data types in <> operator ('Computer Identifier', 'Entity Reference (Integer)').
Invalid Data Type	'Computer Identifier' data type required instead of 'Computer Identifier'.
Invalid Data Type	Incompatible data types in = operator ('Computer Identifier', 'Computer Identifier').
Invalid Data Type	Incompatible data types in = operator ('Computer Identifier', 'Computer Identifier').
Invalid Data Type	'Computer Identifier' data type required instead of 'Computer Identifier'.
Invalid Data Type	Incompatible data types in <> operator ('Computer Identifier', 'Entity Reference (Integer)').
Invalid Entity	Circular dependency in Computer identifier.
Invalid Entity	Computer identifier attribute data type must be set to 'Integer' or 'Text'.

Figure 3.9: *ServiceStudio notifying the user to errors in the model*

with the callee's interface, or a web link that refers to a *Web Screen* that has been deleted, are some examples of errors that may occur. When there are no more validation errors, the developer is free to trigger the publication from the *Service Studio*.

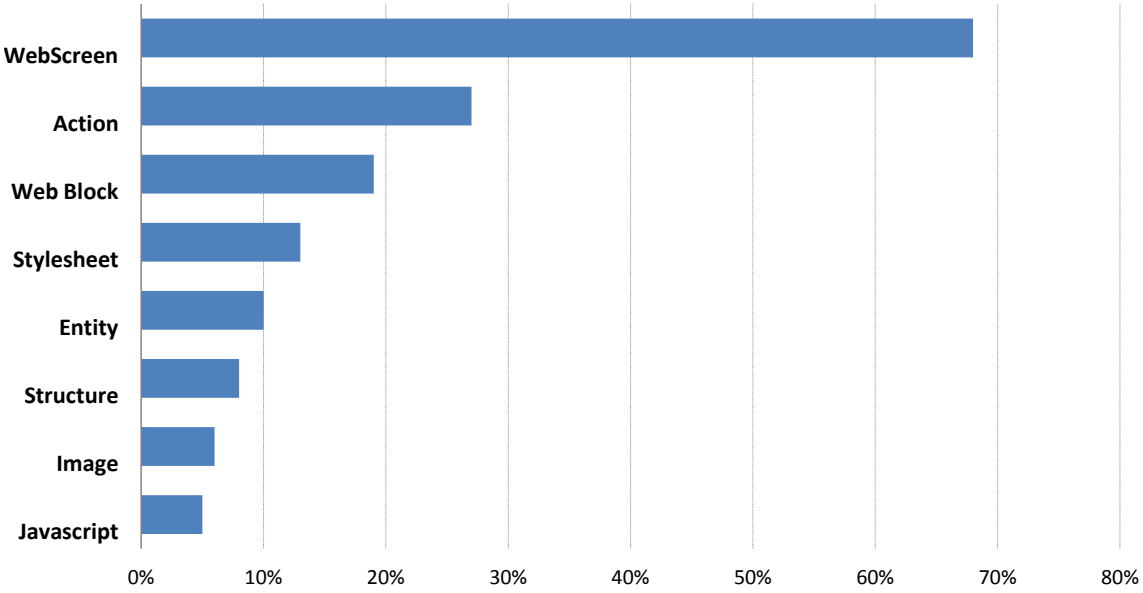


Figure 3.10: *Top elements most changed between consecutive versions*

3.2.2 Platform Usage Patterns

In order to improve the developer's experience we need to know which are the actual usage patterns of the platform. We now show some metrics, previously collected by the *OutSystems* team, for a typical set of projects, and obtained by analysing which are the most changed elements, and hence that are most often compiled.

These results account for 4715 publication operations and 15 different projects. From this data, we obtained the probabilities of each element being changed between successive publications, and present it in figure 3.10. The results reveal that the most frequently changed elements are in the UI components instances, such as *Web Screen*, *Web Block*, *Stylesheets*, and *Javascript*. These results are not surprising since the UI elements are the ones that require the largest amount of fine-tuning, given their relevance to application user's adoption. It is worth noting that in more than half of publications, a least one *Web Screen* is changed.

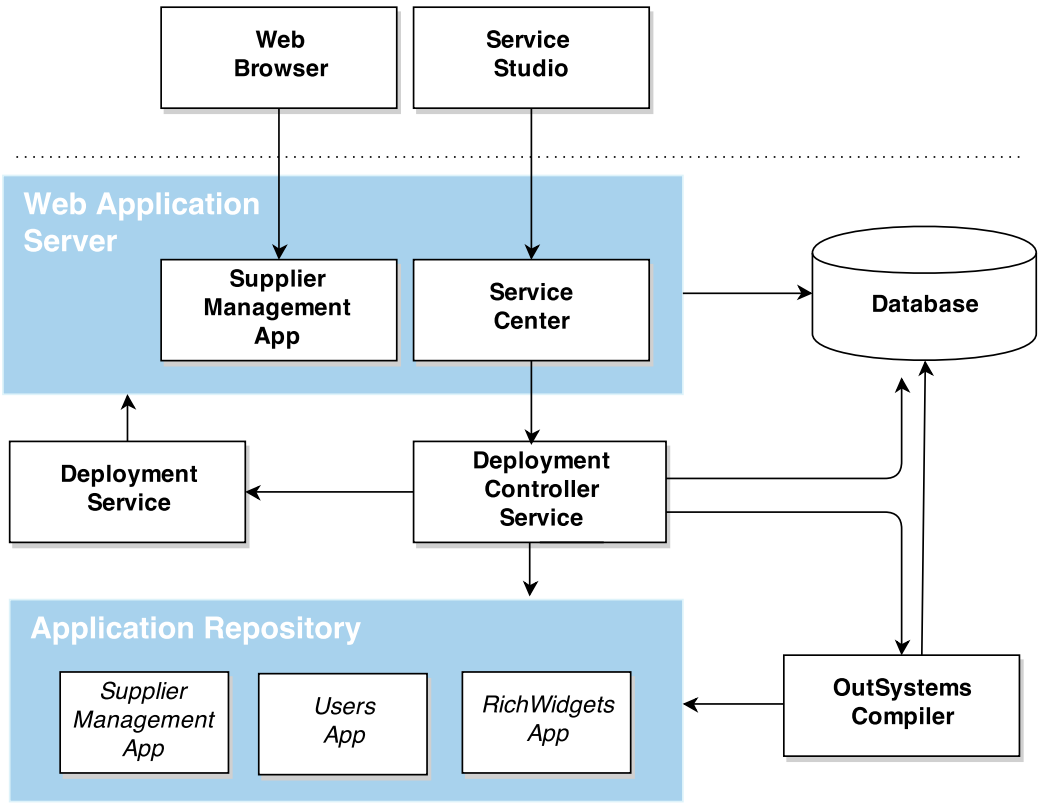


Figure 3.11: *OutSystems Platform Server's architecture*

3.3 Platform Architecture

The *OutSystems Platform* has two major components: the *Service Studio*, the integrated development environment where the developer creates and develops applications, and the *Platform Server*, where those applications are compiled and deployed. Both the compilation and deployment are aggregated in a single action called the *Publication*, which is performed on the *Platform Server* side.

Inside *Platform Server*, there are smaller components, that assume different responsibilities in the publication, and cooperate to achieve an application's publication. Figure 3.11 details both the components and the interfaces that bind them. The *Service Center* acts as a facade between *Service Studio* and the remaining components of *Platform Server*. For the particular case of the *Publication*, the *Service Center* communicates just with the *Service Center*, which orchestrates most of the publication process. Figure 3.14 is a sequence diagram that explains the control and data flow between components as the publication unfolds, to help the reader in the description we are about to make.

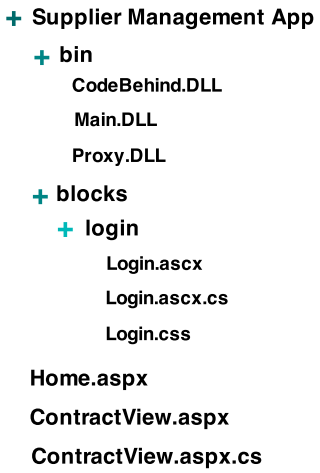


Figure 3.12: *An example of the structure of a deployed application.*

3.3.1 Publication Overview

The *publication* of a publication is a process that consists in transforming the *Application Model* into a standard ASP.NET application and deploying it to the application server. Typically, the ASP.NET application has a structure akin to the one that is shown in figure 3.12. The result of publication comprises code in different languages and file formats. It includes: `ASPX` files and `ASCX` files to define the web pages of the application, *Stylesheets* and *Javascript* scripts to define the client's behaviour, `DLL` assemblies that contain the application logic, and SQL scripts to define changes to the meta-model and migrate data and database schema.

These files are generated from *Compilation Units*, which are the model elements that are transformed in files of some sort. Examples of *Compilation Units* are the *WebScreen* and the *Action*. Other important concept is the *Deployment Unit*. A *Deployment Unit* is a model element that can be compiled and deployed independently. Currently, only the Espace is a *Deployment Unit*.

An Espace is compiled into three assemblies: `Main`, `CodeBehind`, and `Proxy`. Model elements that may be consumed by a *Consumer Espace* are compiled into the `Main` assembly (which are the majority), whereas `CodeBehind` receives everything else that is private to an *Espace* (in this case, only the WebScreens). The *Proxy* assembly acts as layer between a *Consumer* and a *Producer*, by which the former consumes the elements exported by the latter. Further on, we will not care about the *Proxy*'s role, because it is very specific and out of the context of this work.

Figure 3.13 shows the three phases that a publication goes through: *Code Generation*, *Compilation*, and *Deployment*. *Publication* is triggered in the *Service Studio*. It begins with a publication request message carrying the *Application Model* being sent to the *Service*

Center. The *Service Center* drives the *Deployment Controller Service* throughout the process, dispatching the publication phases as the feedback it receives from the *Deployment Controller Service* is positive.

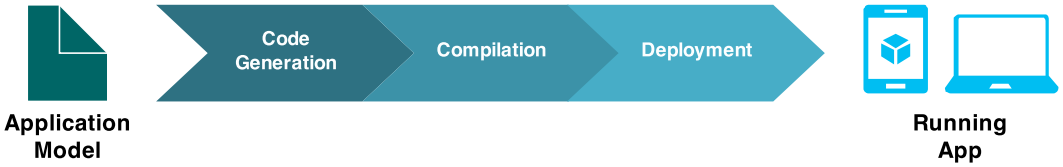


Figure 3.13: *Publication's phases*

The *Code Generation* phase begins; the *Deployment Controller Service* delegates the generation of sources to the *OutSystems Compiler*. Associate to each model element that is a *Compilation Unit*, there is a set of transformation processes that generate the files. The *OutSystems* compiler handles the application model and recursively treats all model elements, executing all applicable transformations. The files generated in this phase are stored in the *Applications Repository*. The *Application Repository* is where applications' code is compiled and stored to be deployed.

After the compiler finishes translating the model, the *Deployment Controller Service* invokes the C# compiler to compile the source files into the set of assemblies mentioned above. The compiler groups the files among the assemblies they belong to. The first assembly that is compiled is the `Main`, followed by the compilation of the `CodeBehind`, which is then linked against the `Main`. These assemblies are also stored in the *Application Repository*.

Generated files also include database scripts that update the database schema and data so that it conforms with the new data model. Scripts are executed at publication time, thus updating the data-model in the database as well as the application's meta-data in the database.

The *Deployment Controller Service* acknowledges the *Service Center* of the termination of the first two phases of the publication process, which then triggers the deployment through the *Deployment Controller Service*.

The *Deployment Service* deploys the application to the *Application Server*. Recall that the application was stored in *Application Repository*, and that the *Deployment Service* requests the generated application to the *Deployment Controller Service*, which produces an archive containing all the deployable files. The last step of the publication process is taken by the *Deployment Service*, that makes the *Application server (IIS)* aware of a new application version.

The *Service Center* gives feedback to the developer in *Service Studio* about a new version running in the attached server, or about any kind of error in the publication process.

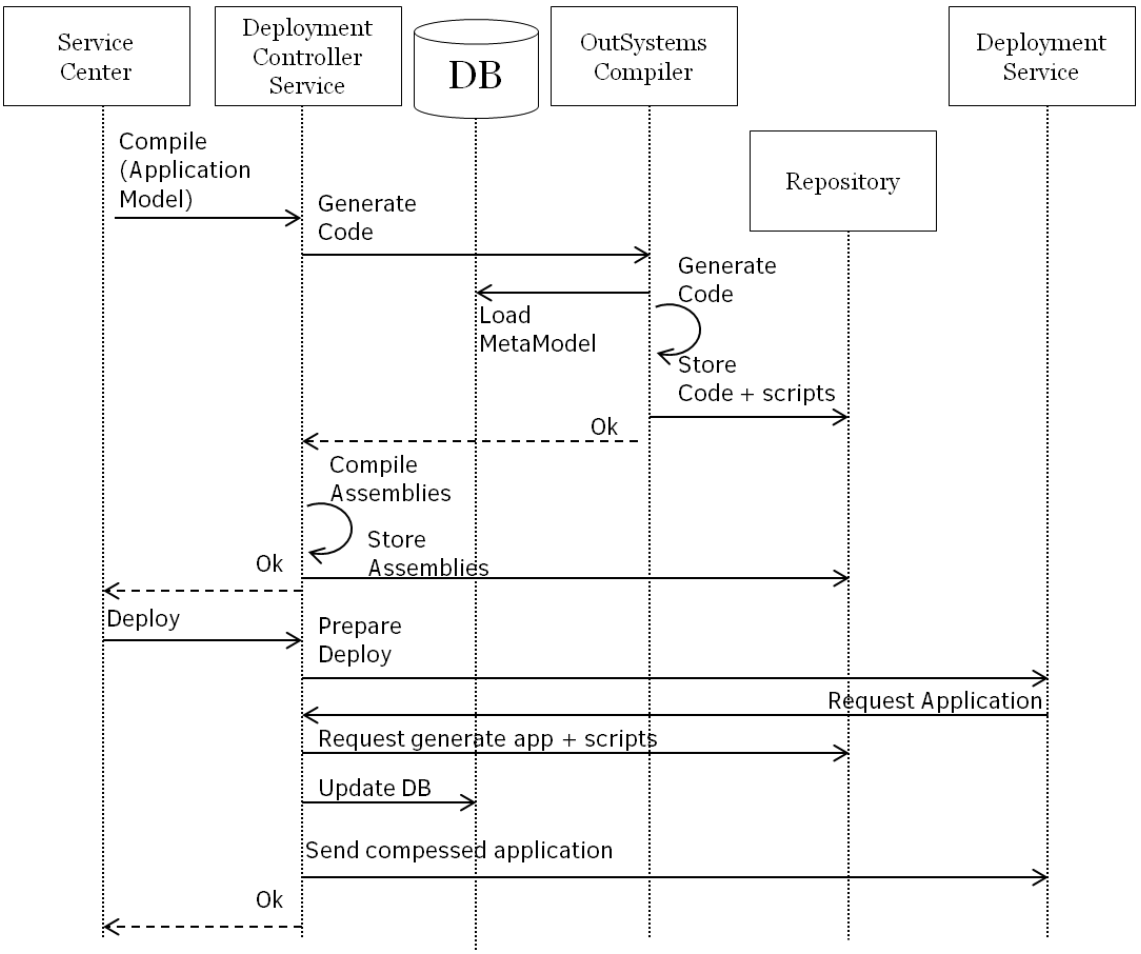


Figure 3.14: *Publication's Protocol*

3.3.2 Compiler Pipeline per Model Element

The description of the compiler pipeline that we gave above does not consider the whole detail of the smaller processes performed over each particular kind of elements. In this section, we complete the description of the pipeline with the details of the compilation operations on individual model elements. All these descriptions should be understood in the context of the general compiler pipeline described at subsection 3.3.1.

Appendix **A** shows a comprehensive graphical explanation of the pipeline.

Espace pipeline

Each *Compilation Unit* contained in a *Espace* is translated to a set of files inside the *Application Repository*. From this set of generated files, C# source files are compiled by the C# compiler into either the `Main` assembly or the `CodeBehind` assembly, depending on whether that element is exportable or not. During *Deployment*, the deployment service copies the *application repository* to the *application server*, `SQL` scripts are executed, and the server is signaled that a new version of the application is available and running.

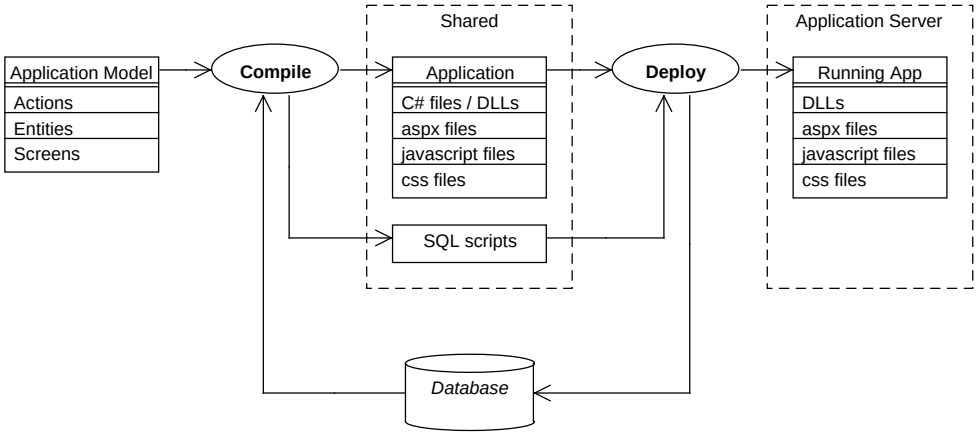


Figure 3.15: *Overall diagram of pipeline*

Action pipeline

Actions are directly transformed into `C#` code. A `cs` file is created for each *Action*, which are compiled together into the MAIN assembly, in the case of user-defined actions that can be used by other *ESpaces*, or into the `CodeBehind` assembly, in the case of *Web Screen* actions.

WebScreen pipeline

During the *Code Generation* phase, two files are created: one `aspx.cs` and one `aspx`, following the structure of a typical ASPX.NET application. The former contains visual structure of the screen, that is, markup with common ASPX metadata that, among other information, identifies the file as an ASPX page. The latter contains the server `C#` code of the *Actions* bound to that *WebScreen*.

In *Compilation Phase*, the `aspx.cs`, along with all the other files of the same type, are compiled into the `Code Behind` assembly. The `aspx` is deployed, but the `aspx.cs` is not, for it was already compiled into the assembly.

WebBlock pipeline

For a *WebBlock*, the compiler generates an `ascx` and an `ascx.cs`. As it is with *WebScreens* `aspx`, the `ascx` is the HTML document that represents the component; in ASP.NET, these files represent *User Control* elements: reusable user defined blocks that are integrated in broader components. The `ascx.cs` contains the backbend logic for the block and it is compiled into the `Main` assembly; recall that *WebBlocks* are exportable, contrarily to *WebScreens*.

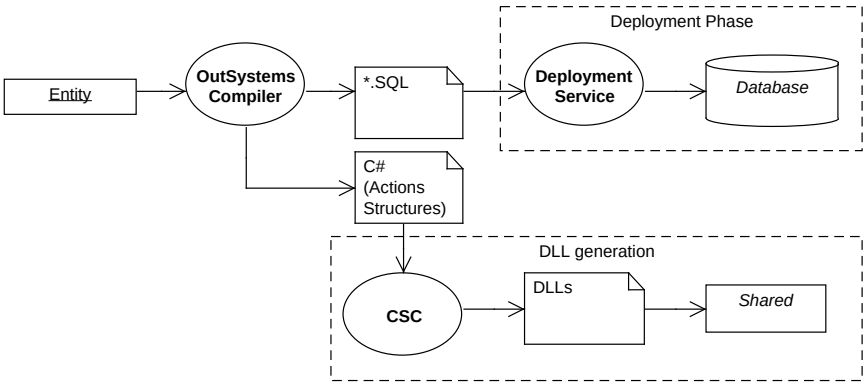


Figure 3.16: *Entity pipeline*

Entity pipeline

During the*Code Generation Phase*, the *OutSystems Compiler* takes an entity definition in the *Application Model* and generates `SQL` scripts containing all the operations needed to update the database so it complies with the new metamodel. To create those scripts, the *OutSystems Compiler* inspects the metamodel on the database and identifies the minimum sequence of `SQL` operations that have to be executed so the metamodel on the server becomes coherent with the new one. In addition, `C#` code is also created to implement the set of actions that are implicitly defined to manipulate instances of entities.

At the *Compilation Phase*, the `C#` source files are compiled into the `Main` assembly. Next, at the *Deployment Phase*, *Deployment Controller Service* executes the `SQL` scripts updating the database.

Structure pipeline

Structures are translated to `C#` source code that define their representation in memory, as well as operations that permit their manipulation in programmatic contexts, such as in a *Action*. The produced source files are compiled into the `Main` assembly, because they can be exported by a producer *Espace*.

Stylesheets, Images, and Javascript

These elements are simply extracted from the application model and deployed along with all the other generated files.

3.4 Differential Code Generation

The *OutSystems Compiler* supports two compilation modes: *Integral Compilation* and *Differential Compilation*. It runs in *Integral Compilation* mode when it has to re-compile the whole *application model*, typically on the first time an application is published, or when

a differential publication was aborted by some reason. The *Differential Compilation* is an optimization introduced in the compiler previous to this work, and that targets only the *Code Generation* phase. The *OutSystems Compiler* runs in this mode for publications that occur after an integral publication. With this mode, only sources provided by the modified model elements are regenerated. OutSystems internal benchmarks show that the Differential Compilation is 40% faster than the Integral counterpart.

The Differential is sustained above three principles:

1. Cache Invalidation
2. Merge
3. Cache Update

The *OutSystems Compiler* keeps a table in the filesystem that maps *Model Elements* to the files that they generated in previous publications, the Cache. Before a publication starts, a *Cache Invalidation* has to be triggered, because there are possibly parts of the cache that cannot be reused, for they no longer apply due to their elements had been changed or deleted. The Compiler identifies the model elements that did change by comparing their signatures. In addition, there are some rules that have to be executed in order to enforce constraints on model elements.

The *Merge* adds to the reused model elements the new model elements. At the end of the publication, the cache is updated with the new model elements and the files that they generate.

3.5 Analysis of Publication Times

Now that we have a more complete notion of how applications are published, it is time to see how much takes to publish a typical medium size application, as well as how much time is spent on each phase. This will allow us to understand which are the phases less efficient and assay the effect of differential mode on the publication times.

Figure 3.17 shows those metrics for both the full publication and differential publication of *Lifetime*, an *OutSystems* application that is used to manage the life cycle of deployed applications.

We are not interested in the Misc Steps times, as it regards steps that do not fall under the scope of this work. Figure 3.17 shows that the full publication takes roughly 38 seconds to compile, whereas differential publication takes 29 seconds. Despite slight oscillations, the difference in times is very small for all the phases but the *Code Generation* phase. Recall that in prior work to this project, the *Code Generation* phase has been optimized to use differential compilation strategies, whose gains are not subtle, for it has an improvement of 40% in compilation times.

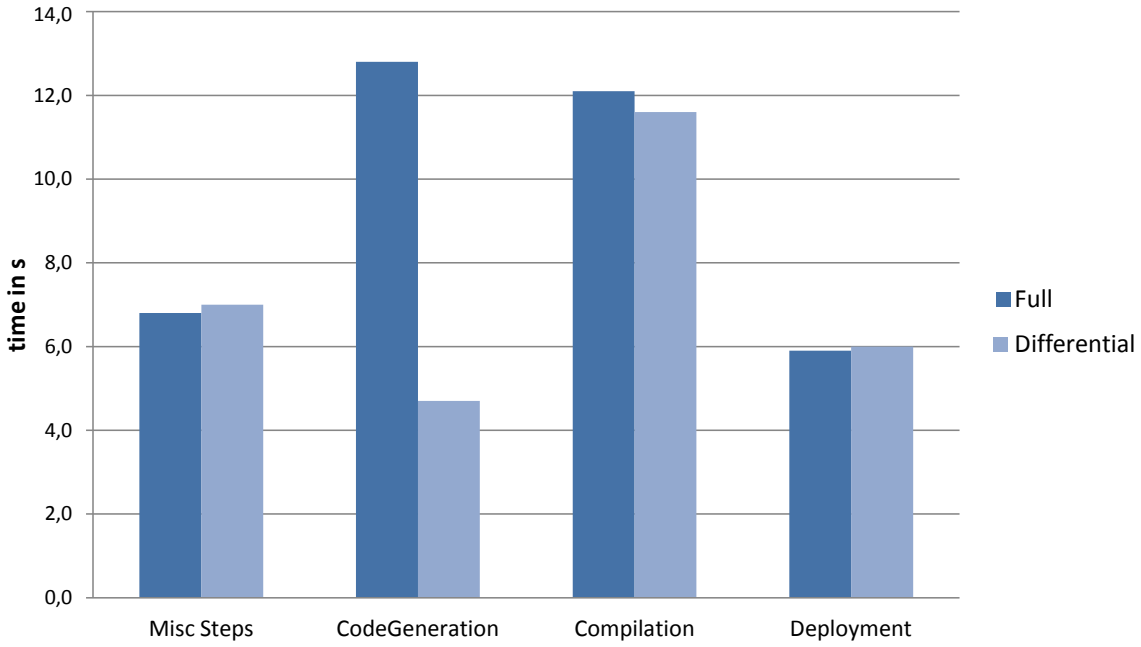


Figure 3.17: *Time spent on each phase*

The *Compilation* and *Deployment* phases are the current bottlenecks of the publication, so they are now subject of our attention. To justify why the times for those two phases are high, we must recall that in the *Compilation*, two large assemblies are compiled for every publication, while in the *Deployment* the compiled application is fully deployed to the *Application Server*. These are the key observations that will drive our proposal.

Note that from the observations presented above we conclude that the publication time is always bounded by the time it takes to compile those two assemblies plus the time it takes to deploy the complete application. This lower bound, which we denote by L, is the minimum time a developer has to wait, independently of the number of elements he has changed after the last time he fired a publication. Ideally, the constant L would not exist; instead, publication times would depend primarily on the number of model elements changed by the developer.

3.6 Dependencies

There are many types of dependencies: two *Web Screen*s bound by http link, a nested *Action* call, a *Web Block* that is contained inside other UI component, among others. Refer to example 4 for a common type of dependency.

Example 4. Consider an Espace *BookStore*, in which we have a Web Screen *Frontpage* and Web Screen *Personal Area*. The *Frontpage* model contains a link that targets *Personal Area*, which is served through *HTTPS*. When *Frontpage* is translated to an html page, the link to *Personal Area* has to be rendered to a valid html link tag with *https* as schema. In order to do so, *Personal Area*'s model propriety *https* has to be consulted.

Matrix 3.18 shows all the dependencies that exist between the elements of the subset we are focusing on. These dependencies are the reason why the `Main` assembly is linked against the `CodeBehind`: the WebScreen, for instance, depends on Entity, but they belong to different assemblies.

Recall that *Service Studio* validates the application model in real time as this is being changed. When an element's interface changes, the Service Studio uses the dependencies graph to find all the elements that depend on it, so it can tell the developer about what become unsound.

	WebScreen	WebBlock	Action	Structure	Entity	Javascript	Stylesheet	Image
WebScreen		✓	✓			✓	✓	✓
WebBlock		✓	✓			✓	✓	✓
Action			✓	✓	✓			
Entity			✓	✓	✓			
Structure					✓			
Javascript								
Stylesheet								
Image								

Figure 3.18: Model Dependencies Matrix

4

Approach

The *Code Generation Phase* of the *OutSystems compiler* is optimized to use an incremental strategy, by caching results for future reuse. All other phases of the compilation process are executed from scratch on each publication triggered by the developer. In the *Compilation Phase*, the assemblies `Proxy`, `Main`, and `Code Behind` are compiled, and in the *Deployment Phase* the *Deployment Controller* does not distinguish between new and untouched components, which causes the deploying of the whole application. This is mainly due to granularity of the assemblies being generated, since any (partial) change will cause that at least one of these "big" assemblies to be modified. In chapter 2, we concluded that the *Compilation Step* is the main bottleneck of the entire publication process, as it accounts for 39% of the total publication time.

The approach presented in this chapter should allow compile times to be somehow proportional to the expectations a developer has about the impact its changes have in the application model. For instance, changing the background color of a Web Screen should have a publication time close to zero. We propose to increase the granularity of compilation units, so that a change on a model element has a smaller impact on the compiled code, fits into a smaller assembly, which is faster to compile than the ones generated in the present model. Typically, the number elements changed by developers between publications is small. Hence, our approach is that of a increased compilation granularity, using thinner assemblies.

We present the notion of *Assembly Distribution*, that defines a systematic distribution of model elements' code by assemblies, and that can be parameterized to obtain different a compilation granularity. This mechanism is static in the previous model.

The distribution into assemblies is constrained on static code dependencies. The concrete publication process is described set of tasks, where each *Task* is a logical execution

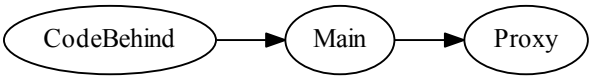


Figure 4.1: *Initial distribution and linking relationships*

unit that produces data, and consumes data produced by other tasks, their predecessors or dependencies. *Dependencies* enforce an execution (partial) order in which tasks ought to be executed.

The graph of tasks is defined by the dependencies, and called *Task Graph*, is built at publication time and is executed by a user level parallel *Scheduler*.

A task defines one operation, from a set of three available types: source code *generation*, *compilation* of generated code units, and *deployment* of compiled code units.

4.1 Refinement of the Deployment Units

With finer modularization, a change on a model element has less impact on the recompilation of an application. Ideally, only the parts that changed or that depend on changed parts are compiled. This is the idea is exploited by tools such as *Make* or *Incremental Compilers*, that allow efficient build strategies which reuse as much as possible from bast builds. In the context of this work, we do not care about modules' cohesion, that is, our approach to the modularization of the application has as aim the publication's efficiency, and not so much if modules are "logical", as the publication is transparent process and the developer is not aware of what applications are compiled into.

Until now, applications were compiled into just three assemblies: `Main`, `CodeBehind`, and `Proxy`. Both `CodeBehind` and `Main` were very dense, for the former contained the code from *Web Screens* and *Web Services*, while the later contained code for everything else. Figure 4.1 depicts those assemblies and the way they are linked with the previous model, from which we departed.

With this model, nothing could reused from past compilations, leading to redundant processing and inefficient executions. A single change would entail the compilation of the whole application. This inefficiency would ultimately entail publication that took longer than what the developer expected. By increasing the number of modules we aim for efficient a *incremental publication mechanism*.

4.1.1 Assembly Distribution

We begin by introducing a new notion. A *Assembly Distribution* is a publication's parameter that states how model elements are distributed by assemblies. More concretely, an *Assembly Distribution* defines a set of assemblies A, which is possibly unbounded, and a

function Γ that maps model elements into assemblies in A. For convenience, we assume that model elements belong to a set M. For instance, the previous model is described by the distribution in which:

$$A = \{\textit{Main}, \textit{CodeBehind}\} \text{ and } \Gamma(o) = \begin{cases} CodeBehind & o \in \text{WebScreens}^M \\ Main & \text{otherwise} \end{cases}$$

We do not considerate the `Proxy` in assembly distributions because as we said in subsection 3.3.1 this assembly assumes a special role that is to act as an interface between a *Producer Espace* and its *Consumer*. From now on, we just assume that all assemblies link against he proxy.

Moreover, a code level dependency between x and y is expressed by $a \rightarrow b$, while linkage between assemblies $a, b \in A$ is denoted by $a \hookrightarrow b$. Recall that in table 3.18 are represented all the code dependencies for the elements that we are focusing.

Assembly Distributions are constrained by the *code level dependencies* between the model elements. Recall that model elements, prior to being compiled into assemblies, are transformed into source code, more specifically, they are transformed into classes that may depend on other classes generated from other elements. Figure 4.2 shows code level dependencies for the model elements that fall under the scope of this work. Refer to section 3.6 for a more in depth discussion about this matter. We do not consider *Javascript* scripts nor *Stylesheets* for they have no dependencies.

For two assemblies a and b, if a has an element t_1 such that $t_1 \rightarrow t_2$, and if t_2 belongs to b, then a must link against b. So, for two dependent elements, either they fall into the same assembly, or the assembly the dependent element is in has to be linked against the assembly where its dependency lives in. Moreover, elements should not be distributed in such way that there are cyclic dependencies between assemblies, otherwise compilation is not attainable.

$$\textbf{if } a \rightarrow b \textbf{ then } \Gamma(a) = \Gamma(b) \textbf{ or } \Gamma(a) \hookrightarrow \Gamma(b)$$

In chapter 4, we will present the iterative process that we undertook in order to find an adequate distribution, as well as the chosen one. The problem is stated as follows: Find an *Assembly Distribution* D, that is, a set A and a function Γ that reduces the compilation times for differential compilations.

We anticipate already that one more factor has to be taken into account, the overhead of calling the framework's compiler. While it is true that compiling smaller modules improves publication time, this strategy can lead to a inverse effect when number of the modules to compile is too large.

The first compilation is particularly critical: since there is nothing that could be reused, all assemblies will have to be compiled. With a more modular distribution, it will take sensibly as much time as the less modular model, because in both all the sources files are

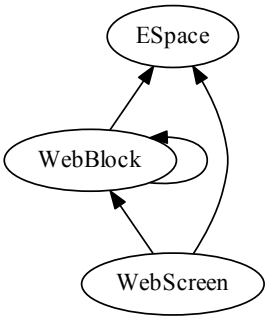


Figure 4.2: *Code Level Dependencies Hierarchy*

compiled, but now there is a new toll, the increased number of calls to the C# compiler.

Thereof, a more granular distribution entails a trade-off between decreased *differential compilation* times and increased *full compilation times*. The challenge in finding a distribution arises is in the balancing between the times for the two publication modes. On one hand, if the times of a first publication are too high, the developer may create a negative first impression about the platform. On the other hand, a *Full Publication* is triggered less frequently, so a even if its times increase, the impact is amortized throughout development.

Testing the distributions is thus necessary to avail more concretely their impact.

4.2 Task Oriented Model

Two assemblies can be compiled in any order as long as they do not depend on each other, which permits their parallelization. Parallel programming is hard, hence it demands abstractions that mitigate complexity and that are easier to us to reason about. Finding a suitable abstraction is the next goal. We observed that it is tractable to decompose the sequential publication model into a set of tasks with narrower responsibilities. We noted as well that the operations where the CPU would spent greater time intervals idle were:

1. Generation of source files;
2. Compilation of assemblies;
3. Introspection of the database.

Because many of those tasks existed already implicitly in the code, the notion of graph of tasks seem a quite natural way of expressing the publication's logic. The *Task* is the main concept in our new architecture. A *Task* is an logical execution unit that accomplishes some goal. It may depend on artifacts produce by other tasks: its precedences. From its precedence's perspective, the task is a continuation. *Task* and their precedences

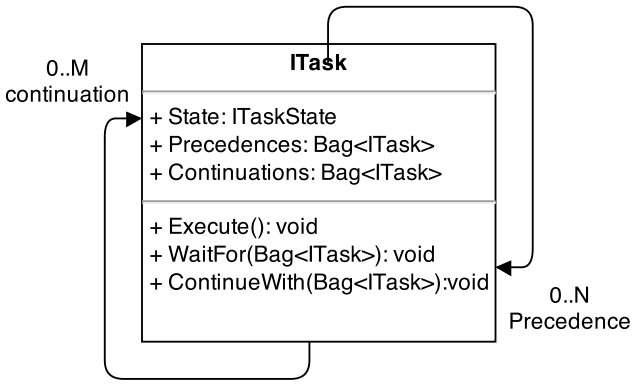


Figure 4.3: *Task's Class Diagram*

form a graph: the *Task Graph*. Any execution model shall respect the semantics of dependencies between tasks, i.e, a task is not allowed to execute until after all of its dependencies have terminated.

During its lifetime, a task goes throughout five states: *Instantiated (I)*, *Ready (W)*, *Running (R)*, *Finished (F)*, and *Error (E)*. A task always starts in the *Instantiated* state, and while it is in that state, it cannot execute. When all dependencies have terminated, the task is in the *Waiting* state, that is, it's allowed to run. It changes to the *Running* state when its execution is triggered (supposing it was allowed to do so). Once a task successfully terminates the job which was delegate to, it commutes to the *Finished* state. The *Error* state is reserved for situations in which an anomaly occurred during the tasks' execution.

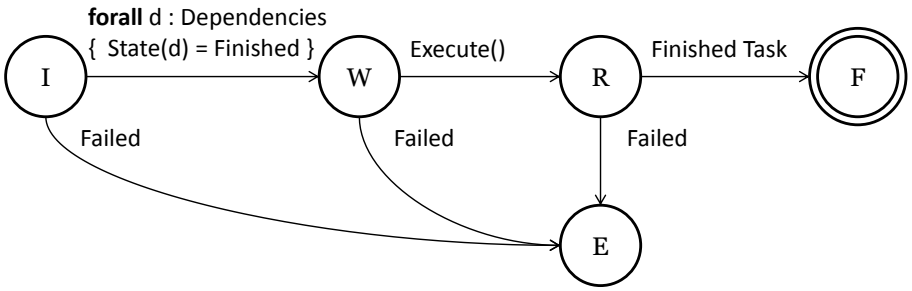


Figure 4.4: *Task's States*

Since some patterns are repeated throughout the code, we deemed that specializing the general concept of task into more specific tasks that could abstract those patterns, would bring more flexibility to the model. For instance, the compilation of an assembly consists in the same sequence of steps for whatever set of sources we compile. A call to the compiler is parameterized by a number of sources to compile, an assembly's output name, and a set of assemblies which it links to. The publication comprises different tasks that fall in one of three categories: *Generation*, *Compilation* and *Deployment*, which a task may be specialized into. Generation tasks compile one or more model elements into source files; Compilation tasks compiles sources files into assemblies, and the

Deployment tasks transport Deployment Units within remote nodes.

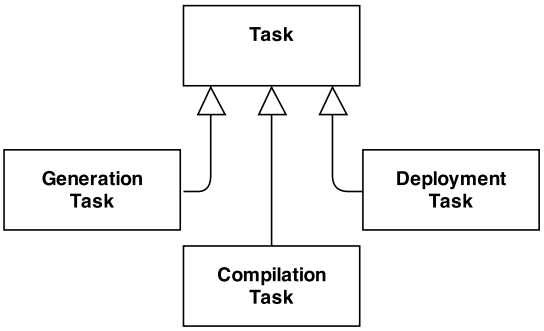


Figure 4.5: *Task's Class Diagram*

4.2.1 Incremental Deployment Model

As an *Espace* grows, more are the files the *Espace* is compiled into, and therefore more is the I/O between between the *Compiler Service* and the *Deployment Service*, which is exacerbated when the *Compiler Service* and the *Deployment Service* are distributed. Once again, we set out to apply the ideas about incrementally with which we tackle the problem of assemblies compilation.

Figure 4.7 gives a glimpse of the protocol between the *Deployment Controller Service* and the *Deployment Service*. *Deployment Tasks* delegate the file transportation to the *Dispatcher*, that then decides when it should dispatch the file to the *Deployment Service*. The *Dispatcher* should also be responsible for batching requests when the load is heavier. The file cache is used to infer if a file should be updated or created on the front end, and that information accompanies the request made by the *Dispatcher*, so the *Deploy Service* knows what to do with the file. The files to delete are found by examining the meta information that is used for the differential code generation.

4.2.2 Building the Task Graph

So far, we have talked about tasks but we have not yet made clear who and when they are created; ditto for they dependencies. Both may be created statically and dynamically. *Compilation Tasks* are created dynamically as they depend on the *Assembly Distribution Policy* that is currently being enforced. For the rest, they are specified by the platform's programmer, as we will now go to describe.

Recall that the application model is hierarchical, that is, broader elements aggregate smaller ones, and so on. Only a subset of those elements need to provide tasks, usually the top level elements. We defined an interface *Task Provider* with which we tag the elements that provide tasks. These tasks are defined statically in the model, contrary to deployment ones.

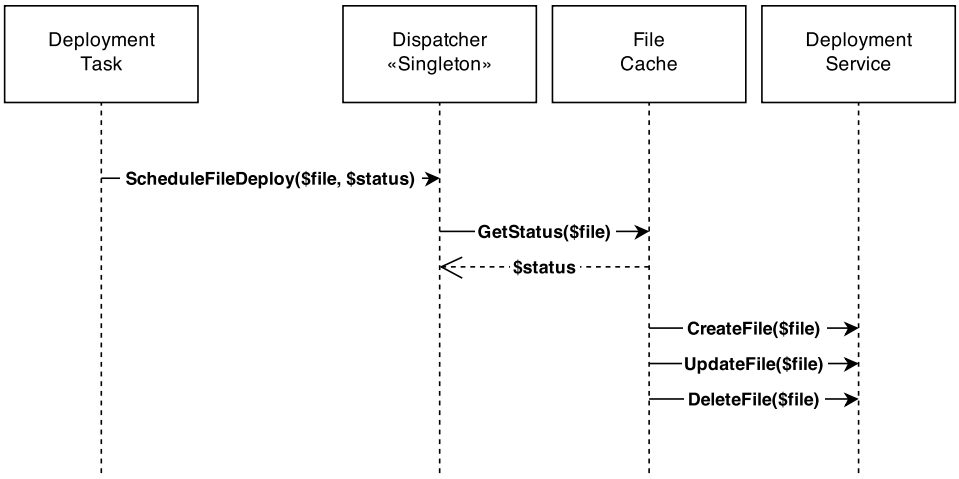


Figure 4.6: *Deployment Protocol*

The *Task Graph* is the model that defines all the tasks that have to be executed for the impending publication, and implicitly defines the relative order in which they are executed through the their dependencies. The *Task Graph Orchestrator* is who creates the task *Task Graph*. It accomplishes that goal by using the *Application Model*, to find which tasks need to be executed, and the *Assembly Distribution Policy*, to find which are the assemblies to be generated so that it creates a compilation task for each one of them.

The *Task Graph* creation is a process that comprises two steps. They are:

- Task Harvesting
- Dependencies Definition

In *Task Harvesting*, the *orchestrator* picks from the model all the *Task Provider* that are set to be compiled. For each one of those, it extracts their tasks and includes them into the set of task G_{tasks}. Then, the *Distribution Policy* is used to find the assembly where that element belongs. It is created the Compilation Task if it not exists and then it's associated to it that element's compilation tasks.

Before a publication is started, we have to infer which tasks to execute, we have to build a *Task Graph*. We defined a new annotation *Task Provider*. A *Task Provider* is an element which have tasks associated to: if a task provider is set as modified, the tasks it provides need to be executed for the imminent publication. We dubbed this step of *Task Harvesting*: from the model, we look for all the modified Task Providers, and then we ask them for the tasks to execute. The tasks provided by the Task Provider might regard not only the provider itself, but also its descendants.

The Compilation Tasks are a special case. These tasks are not provided by the task

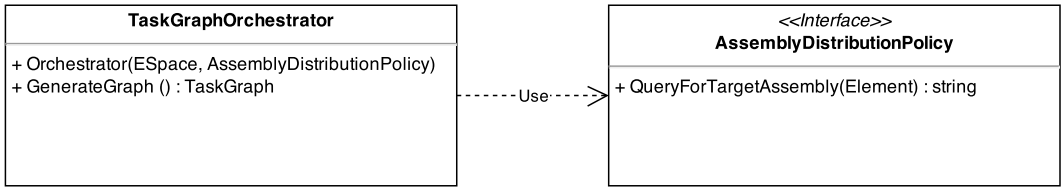


Figure 4.7: *Relationship between Task Graph Orchestrator and Assembly Distribution Policy*

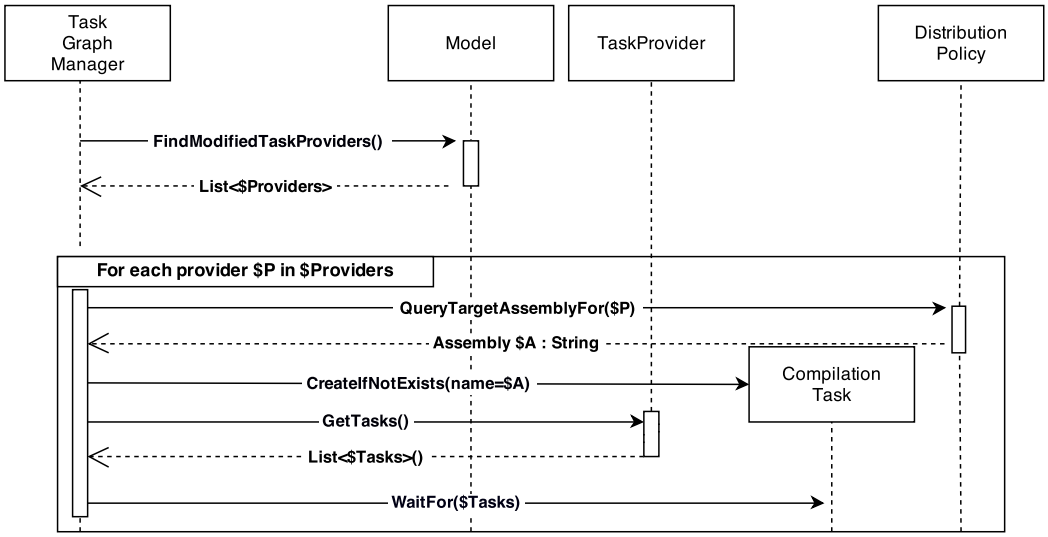


Figure 4.8: *Assembly distribution*

providers, instead they are created by a *Assembly Distributor*. The Distributor is parameterized by an *Assembly Distribution Policy*, which defines which assemblies are created and map each compilation unit to the respective assembly. The distributor, driven by the Policy, distributes the Tasks providers for the Compilation Tasks, and each Compilation Task becomes dependent of the Compilation Tasks provided by the Provider.

Essentially, an *AssemblyDistributionPolicy* is a *strategy* that dictates in which assembly each type belongs to. This notion allows for more sophisticated strategies, that could use, for instance, statistical information about the developer's patterns in order to generate optimal distribution strategies.

4.3 The Execution Model

We have seen that parallelism was not a premise underlying the previous compiler's architecture. Multi-core architectures, which are now pervasive, makes parallelism very desirable, because it improve significantly the efficiency of the publication model. Parallelization is not suitable for every problem, though, and thus it is important to ascertain

if our problem benefits from this strategy. Applications that rely heavily on I/O are improved in a parallel context, because I/O is slow and results in a suspension of the execution, in which the application could be doing progress on other front of its execution.

The Execution model follows from a *Observer-Notifier* pattern and it comprises a scheduler and set of workers (threads). This is depicted by diagram 4.10. Each task assumes the role of a notifier, whereas the scheduler assumes the role of the Observer. This pattern allow us to keep orchestration logic separated from other aspects, such as logging, by having one observer that is a scheduler and other observer that is a logger. The Worker notifies each of its Observers of two events: when it starts executing a task *(onTaskExecution)*, and when it finishes the execution of the task *(onTaskEndExecution)*.

Both the workers and the scheduler execute an event-loop, being asleep in the periods in which they have no work to do. Communication is achieved by asynchronous message passing – each worker waits on a queue with its messages. Every time a workers begins or finishes working on a task, it notifies each one of its observers. The scheduler wakes whenever is notified of a task termination. On doing so, it updates the state of the ongoing execution, and then dispatches any task that might have become ready due to the termination of the task that triggered the event. The scheduler dispatches a task by assigning it to a free worker. When the Scheduler cannot dispatch a task because there is no free workers to whom delegate the task to, the task is kept in the waiting queue until a worker becomes free.

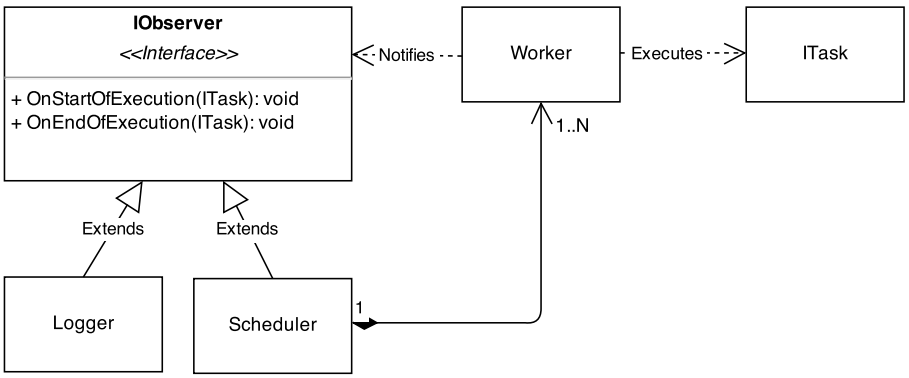


Figure 4.9: *Scheduler's Class Diagram*

The process keeps living until all the tasks have been executed. If the task graph has no cycles and if no task ends up in an infinite loop, we have guarantees of progress and thus that the process eventually terminates. It is easy to prove this claim: if a task always finishes, every time a worker finishes its task, it can begin working on enqueued task. If the scheduler only assigns to workers tasks that have its dependencies satisfied, the workers will execute the tasks in topological order. Since task graph does not have cycles, we prove that the algorithm at some point terminates.

Summarizing, the new publication process executes as follows:

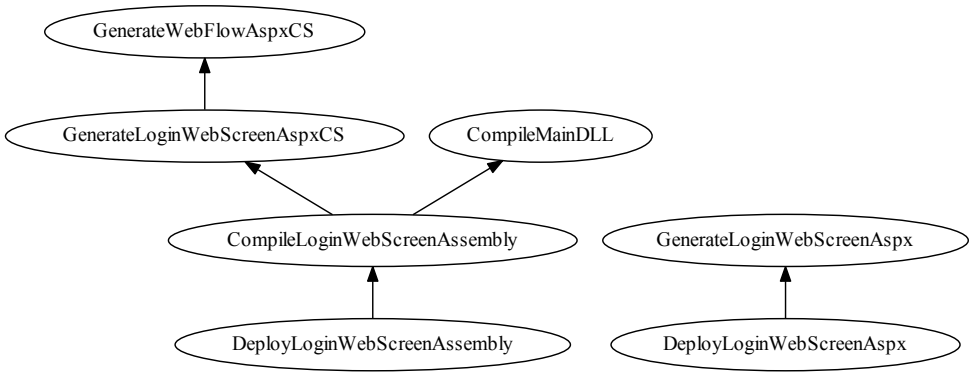


Figure 4.10: *An Instance of task graph*

1. **Task Harvesting:** Look at application model; gather the tasks to execute from tasks providers set to be compiled;
2. **Assembly Distribution:** Create a compilation task for each assembly to be generated and bind to it the tasks harvested in the previous step;
3. **Deployment Tasks Creation:** Create a deployment tasks for each deployment unit that is to be generated;
4. **Task Graph's execution:** Start the scheduler and pass to it the Task Graph; wait for its termination.

5

Implementation

In chapter 3, we presented a new publication model without making considerations about its implementation. With this chapter, we aim to make the bridge between the model and its concrete implementation, resolving open question such as the adoption of an *Assembly Distribution*. Moreover, we also describe the difficulties we had as well as the processes we adopted throughout development.

5.1 Architecture

Our work involved changing the *Platform Server* components, with a particular focus on the *Deployment Controller Service* and the *OutSystems Compiler*. Our model obliges that the *Code Generation* and *Compilation* (and *Deployment* from the dispatcher side) logic to be centralized in a single component. We deemed moving such responsibility into the *OutSystems Compiler* as the most straightforward alternative of all, since the Code Generation's logic is much larger and tangled.

Figure 5.1 gives an overview of the compilation process. Note that a big part of the process is performed by the *OutSystems Compiler*. It uses the *Task Graph Orchestrator* and the *Scheduler* to respectively create and execute the Task Graph. The job of the *Deployment Service* did not change, it sills update the application on the *Application Server*.

5.2 Refinement of the Deployment Units

The rational behind sparser assemblies is that it minimizes the sources to reprocess in the *Compilation Phase*, between consecutive differential publications. For instance, if each element's class was compiled into its own assembly, a change on a model element, as long

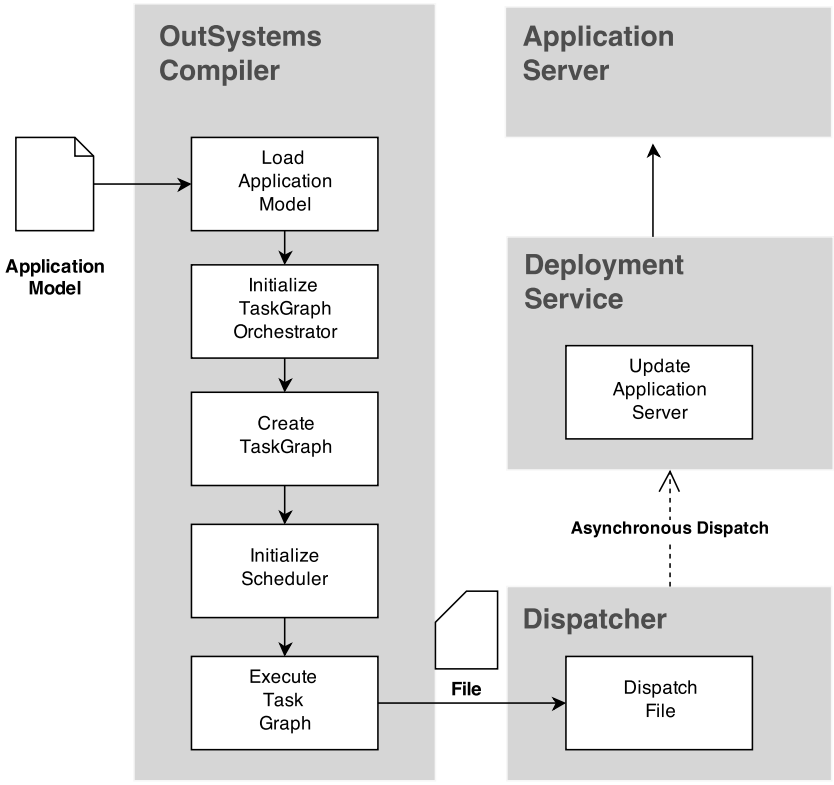


Figure 5.1: *The New Publication Model*

as it does not change its signature, would entail the recompilation of a single class; two model elements, would entail the compilation of two classes, and so on. While less classes are recompiled, the calls to the C# increase, tolling an overhead that did not exist with the previous model. It is not easy, however, to forecast the impact of calling the framework compiler based on number of times that it is called. That is, we cannot assume a constant c and state that overhead of the compiler depends linearly on that constant. Thereof, this is a decision that cannot stand solely on theoretical grounds, but we have to assume an empirical stance and actually test the distributions with real medium sized projects, so we could avail their real impact.

5.2.1 Finding The Right Distribution

It is not easy to anticipate the impact of calling the *C# Compiler*, so finding an efficient distribution is something that has to be done both theoretically and empirically. Picking an *Assembly Distribution* was an iterative process where we tried and validated different distributions until a balanced one was found. In the fist approach we tried, which we called *distribution* $UI^{1:1}$, each *WebScreen* and *WebBlock* was mapped, 1:1, into its own assembly. Using the notation introduced in chapter 3, the distribution is defined as follows:

$$A_{UI^{1:1}} = \{Main\} \cup \{Name(x) : \text{x is WebBlock}\} \cup \{Name(x) : \text{x is WebScreen}\}$$

and

$$\Gamma(o) = \begin{cases} N & \text{for WebBlock with name N} \\ N & \text{for WebScreen with name N} \\ Main & \text{otherwise} \end{cases}$$

Before we venture in further considerations on this distribution, we have to show that the *conditions for linking* are satisfied, otherwise the compilation of certain *ESpaces* may not be attainable, namely if they preconfigure the situations described at chapter 3. Remember that $a \rightarrow b$ means that there is a code level dependency from model element a onto b. First, we have to remember that code level dependencies for *WebScreens* and *WebBlocks* are the following:

$$WebScreen \rightarrow WebBlock \textbf{ and } WebBlock \rightarrow WebBlock$$

The aforementioned dependencies may compromise the *Conditions for Linking*. Observe that neither *WebBlocks* nor *WebScreens* depend on *WebScreens*, hence it is impossible to close a cycle in which there is a *WebScreen*.

On other hand, *WebBlocks* may depend on other *WebBlocks*, and so *a priori* cyclic dependencies could exist. The development environment, however, validates that we do not find cyclic dependencies between Web Blocks, so there is no need to concern about those dependencies. Figure 5.2 exemplifies the dependencies for a subset of elements of an *ESpace*.

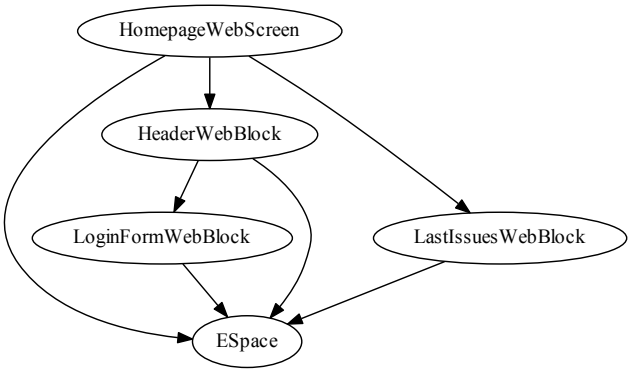


Figure 5.2: *Assemblies Dependency Graph*

With the confirmation that $UI^{1:1}$ is a valid distribution, we may now look at the its impact on the efficiency. Distribution $UI^{1:1}$ reduces the number of elements to recompile

at differential compilations. On the other hand, this distribution becomes fairly inefficient when the number of changed elements reaches a certain threshold: at some point, compiling a large set of smaller assemblies takes longer than compiling a single albeit larger assembly, due to a greater number of calls to the framework's compiler. Indeed, our metrics revealed an augment of 32% in compilation times for integral compilations of large projects. Even though we already expected a deterioration of the times, we were not expecting such a significant impact.

Distribution $UI^{1:1}$ was an edge case where any Web Block and Web Screen are compiled into its own assembly. If there is no overhead in calling C# compiler, this distribution would be efficient, because it allowed to a greater reuse of past publication work. However, as we have seen, this model is not satisfactory due to a keen increment on the integral compilation times, consequence of the overhead of calling the compiler. We proceed to propose and test more distributions.

Aiming for the sweet spot between a finer granularity and a low deterioration on integral publication times, we proposed another distribution. Lets call it *distribution* $Web^{1:1}_{block}/Web^{N:M}_{screen}$. With $Web^{1:1}_{block}/Web^{N:M}_{screen}$, WebBlocks are compiled into individual assemblies as well. This distribution differs from $UI^{1:1}$ in how WebFlows and WebScreens are compiled. A WebFlow is compiled into a separated assembly but it shares that assembly with all its WebScreens too. A topological sort is still performed to drive the compilation of WebBlocks as it was with distribution A.

This strategy reduces the number of assemblies to compile and keeps a reasonable degree of modularity. The problem we had with distribution $UI^{1:1}$ is less prominent in *distribution* $Web^{1:1}_{block}/Web^{N:M}_{screen}$, because we have less assemblies. Finally, using the notation introduced in chapter 3, this distribution is defined as:

$$A_{Web^{1:1}_{block}/Web^{N:M}_{screen}} = \{Main\} \cup \{Name(x) : \text{x is WebFlow}\} \cup \{Name(x) : \text{x is WebBlock}\}$$

$$\Gamma_B(o) = \begin{cases} N & \text{for WebBlock with name } N \\ F & o \text{ is WebFlow } F \\ F & o \text{ is WebScreen and it belongs to WebFlow } F \\ Main & \text{otherwise} \end{cases}$$

5.3 Construction of the Task Graph

In chapter 2, we made reference to *Transformation Rules*, which were used by the *OutSystems Compiler* to transform the *Application Model* into source files. We extended those rules to generate *Compilation Tasks* that accomplish that generation, instead of actively transforming the model. That is, the transformation operations are now deferred and assume the shape of *Code Generation Tasks*, which are extracted from the model at the beginning

of a publication, through the transformation rules application, and executed latter by the *Scheduler*.

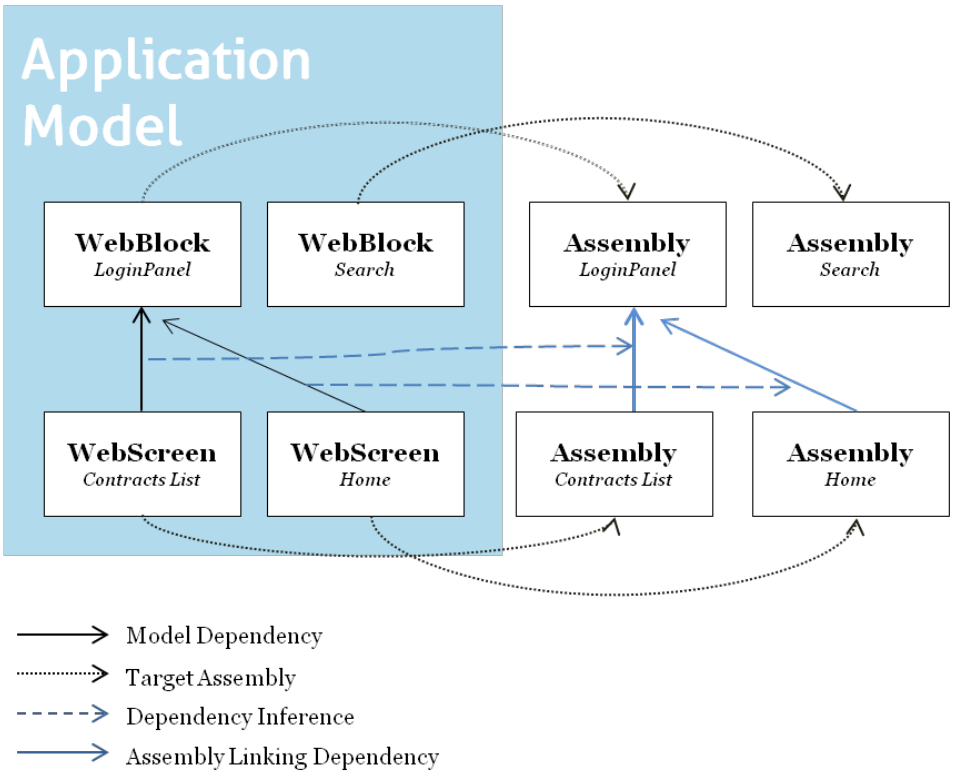


Figure 5.3: *Compilation Task Inference for an application model fragment*

Following the *Compilation Tasks* creation, the model is subjected to a second analysis, by the *Task Orchestrator*, in which the *Compilation Tasks* and its dependencies are created. There is one *Compilation Task* for each assembly entailed from the *Assembly Distribution* in use. To create the dependencies, we have first to find the dependencies between model elements. The *Application Model* is already rich enough to provide us that information. The *Service Studio*, the Platform's IDE, uses a structure called the *Referrers Graph* to find which elements need to be updated due to changes on the interfaces of its dependencies. As figure 5.3 shows, the dependencies between *Compilation Tasks* are inferred from the dependencies between model elements, for those that bind elements that belong to different assemblies.

5.4 Task Graph Persistence

It would be wasteful to recompute parts of Task Graph for every run of a publication. Hence, it could be stored for a posterior use, so we do not have to waste time on redundant computations. We particularly interested in preserving the information about the *Compilation Tasks*, because computing those is expensive. So a *Compilation Task* compiles a assembly, it needs to know all the model elements that belong to it, so it can ask for

their sources, which implies that task has to load them from the application model. Once again, that would not be coherent with our incremental model, because it would load elements that were of not use for the actual publication but because of the sources they generate.

Our answer to this problem was devising a mechanism that accomplished the persistence of the *Compilation Tasks* through a metadata file which is stored in the filesystem. On the beginning of the *Task Graph Construction*, we load from that file the *Compilation Tasks* created in previous publications, which is further updated with the new information gathered from current publication, which may include modification, deletion or creation of *Compilation Tasks*.

The file stores an entry for each compilation tasks. For each entry, it lists the sources files that Task consumes and the names of the assemblies it depends on.

5.5 Task-Driven Model

Parallelism was barely used in the previous architecture, except for operations over the database, that were costly and in which most of the thread's lifetime consisted in waiting for a response from the database. There were no implemented abstractions that could easily support a parallel execution model throughout all the publication process.

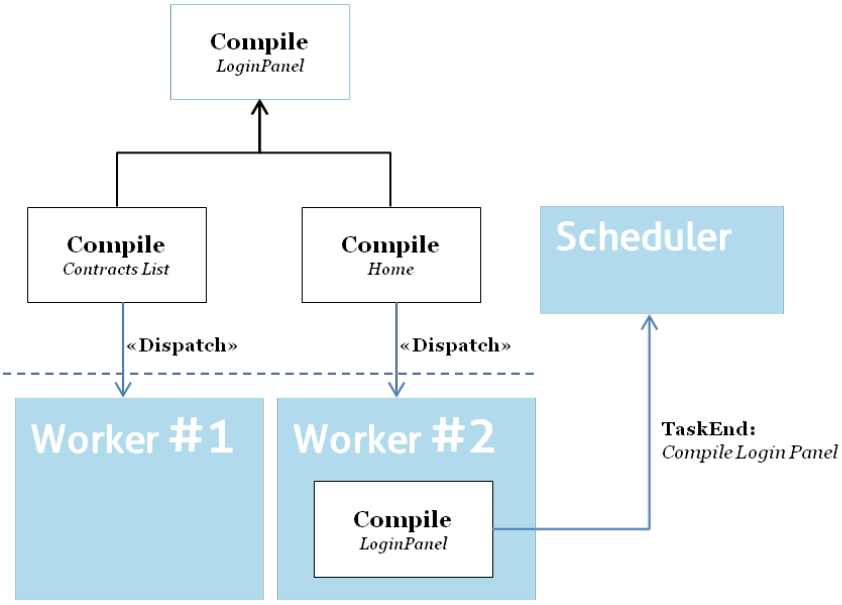


Figure 5.4: *Scheduler*

We developed a small framework that provides primitives that permit the creation of tasks and their dependencies, following the semantics described in chapter 4. With this framework, we reimplemented the publication process, by scattering responsibilities among a set of tasks.

Practically all the code that concerned the code generation from model elements was reused, yet the execution of that code is delegated to the *Code Generation* tasks. When *Code*

Generation task is created, it is passed to it a pointer to the generation method of a model element, at the definition of the Tasks property of its Task Provider. A Code Generation task necessarily has to be a singleton, because it can be referenced by other tasks, and all that tasks need to share the same pointer. This would not be problem if the state of tasks were tracked by an external component; that way, we could simply rely on the identity of the task to query the component for its state, and we could let fall the condition that all the references have to point to the same memory address.

The Scheduler executes the task graph. It is parameterized by the number of threads to spawn. Its interface is quite simple: it has a method *Start* which receives a graph of tasks to be executed.

6

Metrics and Validation

Whether our new publication model is an improvement over the previous one has to be demonstrated through proper metrics. We undertook a benchmark phase whose results and their analysis motivate the chapter we are about to enter.

We simulated 3 typical development scenarios that fully capture the developer's experience. In each scenario, the application was published and the results tracked. After we gathered the metrics, we treated the data and analysed the results.

A test application was published using both the old and the new publication model. For the new model, we tested different configurations having as parameters the *Assembly Distribution Policy* and the number of threads. The distributions are the ones we introduced in chapter 5. Furthermore, each test for each scenario was replicated 10 times to reduce variance. The numbers that are presented next are the averages of those 10 repetitions.

One could argue that the average is not an reasonable metric, because the distribution that rules publication times may not be normal, due to factors that generally are highly variable, such as I/O and the machine's load. That observation does not apply to our context, since all tests were performed under a controlled setting, in which we do not need to compete for the database and the workload besides of the compiler was reduced to a minimum.

6.1 Test Environment

Our tests had as subject the *Lifetime*, an internal *OutSystems* application developed with the platform that is used to track various aspects of other applications' life cycle. The *Lifetime* is considered to be a medium size application that currently comprises:

- 40 *WebScreens*;
- 47 *Web Blocks*;
- 13 *Entities*;
- 103 *Actions*

The tests were performed on a machine with an *Intel Xeon E5 2.26GHz* and 16GB of RAM, running *Windows 7 64bits*.

6.2 Development Scenarios

We have identified 3 main development scenarios that capture the whole development experience with the platform:

- **Full** The application is compiled for the first time (testing integral computation);
- **UI**: The developer changes a small set of WebBlocks and a WebScreens (testing differential case for the most favorable case);
- **Generic**: The developer changes a set of includes other elements besides WebBlocks and WebScreens (testing the differential case for a generalized scenario)

Naturally, for any of the cases, the set of changed elements is the same for any simulation of that scenario.

6.3 Results

We analyse each scenario separately. As you are about to see, our model performed better at all the scenarios but the full.

6.3.1 Full Scenario

As it can be observed in Figure 6.3, the new publication model performs worser than the former model, for whatever parameterization that is used. Still, from all configurations, the *Distribution* $UI^{1:1}$ with 2 threads is the one that achieves times nearer of the ones of the former model, while *Distribution*$W^{1:1}_{block}W^{N:M}_{screen}$ in general yielded the worst results.

We have already noticed in chapter 4 the reason for this scenario's times getting worse with a more granular model. With a more fine grained model, an application is compiled into more assemblies than it was in the previous model. Consequently, in the case of a full publication, where there are no cached results yet and so all assemblies have to compiled, it ends doing more processing that it previously did, which has impact on its times. As such, *Distribution* $UI^{1:1}$, which is very granular, as each UI element is compiled into its own assembly, had the worst times.

Model	Time
Former	37.6s
Dist $UI^{1:1}$	68.3s
Dist $UI^{1:1}$ 2 threads	57.5s
Dist $UI^{1:1}$ 3 threads	53.5s
Dist $UI^{1:1}$ 4 threads	50.2s
Dist $W^{1:1}_{block}W^{N:M}_{screen}$	48.7s
Dist $W^{1:1}_{block}W^{N:M}_{screen}$ 2 threads	46.0s
Dist $W^{1:1}_{block}W^{N:M}_{screen}$ 3 threads	46.4s
Dist $W^{1:1}_{block}W^{N:M}_{screen}$ 4 threads	50.0s

Figure 6.1: Times for Full Publication Scenario

It should also be noted that without parallelism times have a very steep increment, of about 81%. Therefore, it is recognized the efficiency of parallelism in this context, since it attenuated the impact of the calls to the C# compiler, shrinking the augment of 81% in times to 24%,

This deterioration of the times have low impact on the development experience, though, because the full publication is a rare event. It is only expected to occur once: when an application is compiled for the first time.

6.3.2 UI

In this scenario, our finer grained model pays off. For both distributions, it is yielded a keen improvement of 38% of publication times. This is the kind of scenario that we aimed to optimize, due to it being so frequent throughout an application development. It is also the one that supposedly would benefit more from a fine grained model.

Times had a large decrease because now only a few set of small assemblies is compile, instead of the three assemblies compiled previously. Figure 3.17 shows that for *Lifetime*, 12s were required for the compilation of the assemblies. Indeed, if we look at the data shown in figure 6.2, we see a cut of nearly 12s on those times.

Model	Time
Former	27.5s
Dist $UI^{1:1}$	18.1s
Dist $UI^{1:1}$ 2 threads	17.6s
Dist $UI^{1:1}$ 3 threads	16.2s
Dist $UI^{1:1}$ 4 threads	19.2s
Dist $W^{1:1}_{block}W^{N:M}_{screen}$	19.0s
Dist $W^{1:1}_{block}W^{N:M}_{screen}$ 2 threads	17.1s
Dist $W^{1:1}_{block}W^{N:M}_{screen}$ 3 threads	16.8s
Dist $W^{1:1}_{block}W^{N:M}_{screen}$ 4 threads	19.5s

Figure 6.2: Times for UI Publication Scenario

6.3.3 Generic

Model	Time
Former	27.5s
Dist $UI^{1:1}$	25.0s
Dist $UI^{1:1}$ 2 threads	23.1s
Dist $UI^{1:1}$ 3 threads	21.2s
Dist $UI^{1:1}$ 4 threads	21.8s
Dist $W^{1:1}_{block}W^{N:M}_{screen}$	29.0s
Dist $W^{1:1}_{block}W^{N:M}_{screen}$ 2 threads	24.9s
Dist $W^{1:1}_{block}W^{N:M}_{screen}$ 3 threads	26.6s
Dist $W^{1:1}_{block}W^{N:M}_{screen}$ 4 threads	27.1s

Figure 6.3: Times for Full Publication Scenario

For this scenario, our implementation improved the publication times too, even though the reductions are not as high as the ones of the UI scenario. Recall that in this scenario we also changed elements that belong to he `Main` assembly, which triggered its compilation. However, the `Main` assembly is now much smaller, so its compilation has a smaller contribution to the compilation times.

6.4 Remarks

Our metrics reveal that the publication times for the *UI* ($\tilde{4}1\%$) and the *Generic* ($\tilde{2}4\%$) case improved with our new model, while a *Full* compilation become slower (about 22%). A fair trade off, if we consider that the full publication is a rare event but the other two scenarios are not. With more experimentation and fine tunning we may find distributions that achieve better compromises, but the actual results are already a sheer demonstration of our point: a more granular compilation model can reduce significantly the publication times.

Note that we did not implement the incremental deployment model envisioned in our model, and so the deployment stills being the very inefficient step we described before In the chapter 3, figure 3.17 showed that the *Deployment Phase* took 6s to be completed. We strongly believe that a incremental deployment model can have as such substantial gains on publication times. Indeed, we made a rough prototype that demonstrated that this time slice could be reduce to 2s, in a publication under the same circumstances.

Finally, it shows that parallelism is very effective at fastening up the publication process, but using more threads might, sometimes, have the inverse effect. It is something that is hard to avail since it depends on the machine, on the network conditions, on the workload borne by the machine on that moment. In our tests, where we had total control on those factors, we observed that using 2-3 threads times yielded to the best possible publication times.

7

Conclusion

Do you remember Dave, the developer who was working on a supplier management application? He was used to fast deployment times, for he formerly developed in Ruby On Rails, whose pipeline is way less complex than the OutSystems pipeline. Notwithstanding, he recognizes that the development with OutSystems platform is faster and less prone to errors, which is a decisive advantage in an enterprise context. Nonetheless, this story evinces that slow build times may disturb the developer. Waiting a long time span every time he wishes to test the changes he made to an application decreases his productivity and satisfaction with the usage of the platform. Certainly, Dave would be happy to know that he can get more productive with platform as result of these improvements.

Ideally, the programmer would receive feedback as soon as he performed an change to the application. This is so important that OutSystems has a team whose ultimate goal is to have nearly instant publication times. With this work, we explored a path that we believed would yield considerable improvements on the publication's times, based on the measurements and observations about the previous publication pipeline, with the potential of become the basis of future work by the OutSystems R&D.

Some insightful ideas can be drawn from this work. We saw that through smart strategies that store and reuse work from previous publications we can improve significantly the efficiency of computational processes. Underlying these strategies, there are two main ideas: *Cache Invalidation* and *Changes Propagation*. *Cache invalidation* is the process that finds what needs to be re-processed because it cannot be reused from previous runs. In our context, cache invalidation was used to identify which tasks are re executed for the imminent publication. *Changes propagation* finds which elements of the cache need to be invalidated based on a set of modified elements that act as seeds, and the dependencies between them.

On more lesson is that sometimes design decisions impose trade-offs which we need to judge either as fair or not and thus if we are willing to accept their implications. Such dilemma was illustrated by the assembly distribution, in which though a deterioration of the full publication times is inevitable, the times of the differential publication improved significantly.

Finally, this project was opportunity to apply good engineering practices. The deficiencies that we mentioned were not known from beginning, or at least they had not been properly identified and characterized. Their identification was the culmination of a preliminary analysis of the publication model, which the first semester was dedicated to. After identifying the main inefficiencies, the posterior step was proposing solutions that could be coherently integrated into the model, and were realistic for a dissertation context and that accounted for backward compatibility

Some aspects were subject to several iterations until a good compromise could be found. A good example of such is the distribution model, where we tried and measured different configurations until we found the most balanced one.

7.1 Future Work

Our work was a step towards a model where model elements such as *Web Screens*, *Entities*, and *Actions* are both compilation and deployment units. In such model, a single model element could be subjected to the pipeline, instead of what happened in the previous model, where the *Espace* was the only deployment unit.

In this section, we describe the insights we had both after and during the development of this project. Due to the time constraints, the ideas we present here were not implemented, but nevertheless they are promising extensions to this work that could be undertaken in future iterations.

7.1.1 Differential Deployment

At the *Deployment Step*, the last step in the publication, all the generated files are compressed by the compiler into a single file and dispatched to the *Application Server*. For large applications, this final compressed artifact will be big as well, and its size will have impact on the network I/O time.

Ideally, only the files that were modified should be dispatched to the server. The *Deployment Controller Service* would tell to the front-end which files it must create, update or delete, through an extension to the actual deployment's protocol. Despite the simplicity of this idea, it would demand centralizing the orchestration of publication in a *Deployment Controller Service*, which is indeed the most correct approach to this problem, but it would demand a considerable amount of time.

7.1.2 Dynamic Assembly Distribution

The *Assembly Distribution* is a critical step in our model, for it can significantly improve the times for differential publications, but at the same time carries some challenges as we described at chapter 4. We already remarked that learning from the developer's patterns may be an interesting way of inferring optimal distribution models.

Every publication, the system would collect metrics about the elements that changed, infer development patterns and it would cluster model elements driven by that data. In one simply approach, the most changed elements would be segregated into a specific assembly. If the system inferred that most of the times in which the developer changes the element A would also change the element B, it would change the dynamic distribution so the both elements are compiled in one assembly.

7.1.3 Workload Balancing

The compiler service might be running in the cloud. In such environment, the resources usage is much more voluble and hereby the system should be sensible to those fluctuations in order to not make decisions blindly that could compromise performance. The scheduler, for example, should take the current workload into account when deciding on how many workers to span.

Other strategies that aim to manage the workload by the workers should also be considered. For example, characterizing each task as I/O bound or CPU bound could be used in order to schedule tasks in a smarter way. The scheduler would pair I/O bound tasks with CPU bound tasks in order to increase the number of tasks being executed concurrently.

7.1.4 Alternative Concurrency Models

Our execution model uses native threads as the underlying model concurrency model. While they are efficient for parallelizing tasks that are CPU bound, they are wasteful for I/O bound ones. Moreover, they are expensive and their use is liable to race conditions and deadlocks. Non pre-emptive/cooperative alternative thread models should be subject of future considerations, for they solve some of those problems and more suitable for heavy I/O scenarios.

Bibliography

[AAB08] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. *SIGPLAN Not.*, 43(1):309–322, January 2008.

[ABH01] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *IN PROCEEDINGS OF THE 29TH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES*, pages 247–259. ACM Press, 2001.

[Aca05] Umut A Acar. *Self-adjusting computation*. PhD thesis, Citeseer, 2005.

[Aca09] Umut A. Acar. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, pages 1–6, New York, NY, USA, 2009. ACM.

[Baa88] Erik H. Baalbergen. Design and implementation of parallel make. *COMPUTING SYSTEMS*, 1:135–158, 1988.

[BWC01] David M. Beazley, Brian D. Ward, and Ian R. Cooke. The inside story on shared libraries and dynamic loading. *Computing in Science and Engg.*, 3(5):90–97, September 2001.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 266–277, New York, NY, USA, 1997. ACM.

[Car02] Magnus Carlsson. Monads for incremental computing. *SIGPLAN Not.*, 37(9):26–35, September 2002.

[Fow90] Glenn Fowler. A case for make. *Software—Practice and Experience*, 20:30–46, 1990.

[GJS+13] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[HLMY99] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. The vesta approach to software configuration management, 1999.

[LYBB13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

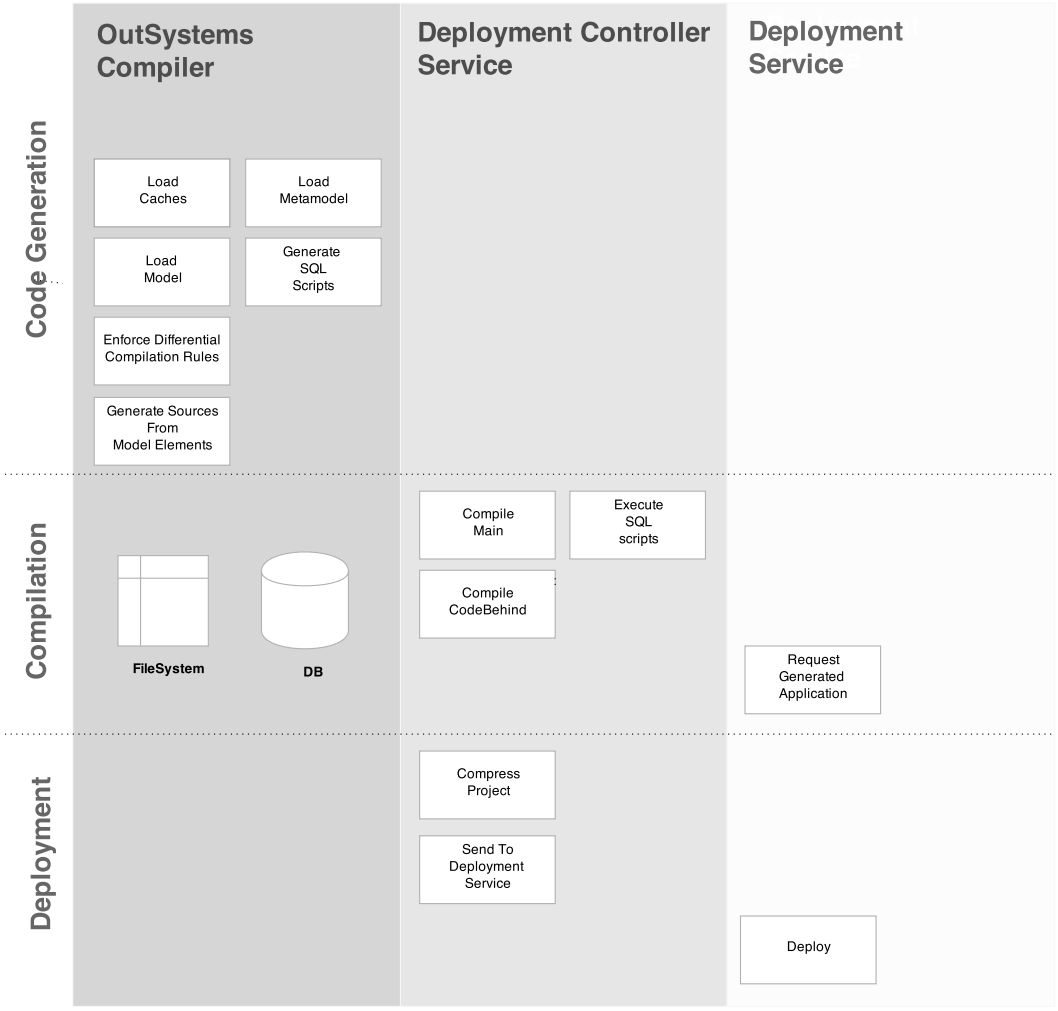
[SA93] Zhong Shao and Andrew W. Appel. Smartest recompilation. In *In ACM Symp. on Principles of Programming Languages*, pages 439–450. ACM Press, 1993.

[TGS08] F.A. Turbak, D. Gifford, and M.A. Sheldon. *Design concepts in programming languages*. Mit Press, 2008.

[Tic86] Walter F. Tichy. Smart recompilation. *ACM Trans. Program. Lang. Syst.*, 8(3):273–291, June 1986.

Publication Sheet

??


OutSystems Compiler
Deployment Controller Service
Deployment Service
Code Generation
Load Caches
Load Metamodel
Load Model
Generate SQL Scripts
Enforce Differential Compilation Rules
Generate Sources From Model Elements
Compilation
Compile Main
Execute SQL scripts
Compile CodeBehind
FileSystem
DB
Request Generated Application
Deployment
Compress Project
Send To Deployment Service
Deploy