The design of a framework for compilers development

Paulo Jorge Matos
Instituto Politécnico de Bragança
Campus de Santa Apolónia
5300 Bragança, Portugal
pmatos@ipb.pt

Pedro Rangel Henriques
Universidade do Minho
Campus de Gualtar
4700 Braga, Portugal
prh@di.umihho.pt

Abstract

DOLPHIN framework is a solution conceived to support the development of modular compilers. Its supplies a large set of components, like: front-end's, back-end's, code analysis, code optimizations and measure components that can be combined to build new compilers. All these components work over a single form of intermediate code representation, the DOLPHIN Internal code Representation.

The main principle that guides the conception of DOLPHIN framework was to build a user-friendly solution to develop high quality compilers. Such solution was achieved based on three main concepts: components, components reuse and data consistency. This paper, that aims to present the architectural design of DOLPHIN framework, demonstrates: how the concepts presented above influence the framework architecture; how they were "implemented" on the framework, namely shows the interfaces defined for the components and for the code representation; how the components are related; how to use the components to implement concrete compilers; and how to evolve the components and the framework to support new features.

1. Introduction

The compilation process, that converts a program written in a high-level programming language (source language) into assembly or machine code (output language), is more and more a complex problem. By one hand, the source languages are quite more powerful and distant from the syntax and the paradigm of the output language, which raises difficulties for translation. By the other side, the computational architectures, i.e. the target machine + operating system, are more complex and demanding, making the generation of the output code more difficult, at least to generate efficient code. Even the software development process is much more tolerant and flexible, trusting on the compiler to compensate the faults and inexperience of the programmers, requiring elaborated error detection and error handling mechanisms, and complex code optimizations.

The different stages of the compilation process use very distinct solutions, requiring simultaneously a strong knowledge about conception of programming languages (syntax and semantics) and about microprocessor architectures. Considering the complexity that is inherent to the development of compilers but also the possibility to specify formally some of the parts, many tools appeared to help on this task. Some of these tools make use of very stable solutions and accomplish very well the requirements for which they were built, but most of them can only support the development of a single compiler task. In practice, to implement the whole compiler, developers have to resort to several of these tools. But the ability to understand and manipulate them is by itself an obstacle. As a consequence, in the last years, some projects appeared aiming the integration of such tools (and technologies) into single solutions that support the whole development of compilers. DOLPHIN is one of these projects [8, 9]. It is basically a framework that supplies several ready to use components, and incorporates a set of tools to build new specific components. All these features are based on the use of an intermediate code representation model over which most of the components work, the DIR - DOLPHIN Internal code Representation [13]. This paper focuses on the architectural design of the framework that was conceived concerned with a single purpose: build a user-friendly framework without given up of the quality of the compilers built with it.

The next section introduces some essential requirements for tools and frameworks that aim to support the development of compilers. It is also showed how they influence the design of such tools/frameworks and how some concrete tools and frameworks accomplish these requirements. Section three introduces DOLPHIN framework, namely the code representation (CR) elements, the type of components and the structure of the compilers built with this framework. Section four describes the architectural design that supports

the framework, explaining the type of interactions that occur among components and presenting the interfaces and protocols defined to guarantee the correct function of the framework. At section five, the conclusion is drawn.

2. Tools and frameworks to build compilers

At this section some fundamental requirements are introduced for a framework that aims the construction of compilers, emphasizing the requirements that contribute to obtain a user-friendly solution. It is also done a brief analysis to some solutions that aim the same type of utilization.

Compilation is a process that involves many tasks; some are related with the interpretation of the source language, others with the code optimization or with the generation of the output code. These tasks can be specifically built as a whole, to produce faster and more efficient compilers; or can be conceived as individual components, allowing their reuse and making compiler modifications easier. Since a framework is basically a set of pre-implemented components, that can be parameterized and joined to assembly new compilers, it is easy to understand how fundamental is that the components can be easily reused and modified.

Strongly related with these two requirements, is the way how the code is represented along the compilation process. Typically, a compiler can have and handle several forms of internal representations, but sharing a single form by a large number of tasks, also promotes the reuse of the components and makes easier to change them. The tasks that do not depend on the source language, neither on the target architecture, are designated by *middle-level tasks*, and the generic code representation used by them is designated by *intermediate code representation*.

Other important requirement is to supply mechanisms/solutions that allow to connect the framework components to external elements, since normally a framework only contributes with part of the final solution. It is also important that the framework supplies mechanism/solutions to add new components.

There are other requirements, that are not so fundamental for the architectural design of a framework or that are important, but seep out of the subject of this paper. So, based on the presented requirements, we will now make a brief review of the main contributions found in the literature.

Some tools are dedicated to specific tasks. It is the case of BURG [4], which based on a description that relates the intermediate code operations with the target architecture instructions, can generate an optimal selector of instructions; or the many examples of lexical and syntactic analyzer generators, like Lex and Yacc. There are also some dedicated frameworks, like the OPTIMIX [3] that aims the construction of code optimizers; and the PAG [7] to build data flow analyzers. All these tools and frameworks are great solutions, but they only make part of the job.

GCC [15], GENTLE [14] and SUIF [2, 1] are examples of tools capable to support the development of the whole compiler. These examples were chosen by their relevance and by the kind of approach used by each one.

The GNU Compiler Collection (GCC) is not a tool to build compilers, but a compiler used as a base to build new compilers. The presence of GCC is justified because it is a widely spread solution and the reasons are obvious: it is a very powerful and efficient compiler, available over the GNU license, that contains lots of reusable code (that runs over GCC intermediate code, the Register Transfer Language), and is supported by a large community. But GCC does not supply any special mechanisms to support the development of new compilers. The user needs to go deeply inside GCC and have a strong knowledge about its structure, to be able to build a new compiler. Even the structure of the GCC does not help to easily reuse the code, since it is very intricate.

GENTLE is a toolkit to implement compilers for domain specific languages that covers all main compilation tasks, namely the ones related with translation and code generation. The tasks are specified using a uniform notation. But GENTLE does not have a single intermediate representation and, some tasks, like elaborated forms of code analysis and optimizations, were relegated to a second plan (probably, a consequence of their relevance for domain specific compilers). Users can always implement these tasks by hand, but GENTLE does not define any kind of interface or protocol to do this, which raises some difficulties to the users, forcing them to understand some internal details of GENTLE.

SUIF Compiler System is one of the most well succeeded systems and, as a consequence, is an obligatory reference in our work. SUIF uses an internal form of code representation, the SUIF - Stanford University Intermediate Format, that supports many of the compilation tasks and that can be extended (there is a specific language for such job, the Hoof); it is strictly based on a modular structure; has several support tools, namely to deal with the CR (consistency checkers, browser, visual shell, etc); there are several projects to build front-ends for the SUIF system, namely for object-oriented languages (like C++ and Java); and there is a large community of users, including some of the most well known researchers of this area. It does not include specific tools to develop front-end's, but supplies a framework to develop the back-end's (that uses a specific form of CR, the MachSUIF). It also supplies several modules for code analysis and optimizations (implemented as dynamic libraries), and a kernel with several support features. The differences start with a driver, that is supplied with the system, that allows to load and execute dynami-

cally the modules. Between the execution of each module (designated by a ”pass”), the user can request the generation of a file with information about the intermediate CR. Both things are very important to help the development of new modules (to detect, for example, semantic errors). But better than that, SUIF defines an interface for the implementation of new modules, that guarantees the full integration with the SUIF system, namely with the driver.

It is difficult (we might dare to say dangerous) to detect disadvantages on such solutions, and even more difficult present a better proposal. But at some specific points we believe that it is, at least conceptually, possible to do a better job. Let's think as the user that resorts to this type of tools. In most cases, the user is somebody that needs a compiler with special characteristics (produce code for a specific target architecture or compile a specific source language); or somebody that wants to implement a particular component and needs a full system to test it. No matter the type of user, he will probably appreciate avoiding the details that are not essential to accomplish his task. Even, when he obtains a solution, most probably he will not spend his time reading more documentation to verify if it is possible to improve the solution or improve the way how the solution is integrated in the system.

As was told above, this paper describes the architectural design of DOLPHIN framework, that as a single purpose: build a user-friendly framework, without lose the capability of building high quality compilers. We intend to show that the concept of user-friendly and the concept of professional framework, capable to support the development of high quality compilers, have not to be antagonistic. So, when compared with the examples presented previously and other analogous systems, DOLPHIN framework aims to improve the way how to use this type of tools, but simultaneously reinforces the consistency of the system and the efficiency of the compilation process. This was possible designing an architecture for DOLPHIN framework that promotes a controlled use and implementation of components.

3. DOLPHIN framework

DOLPHIN framework was conceived to build a modular compiler, composed by a front-end, several middle-level tasks and one or more back-end's. The middle-level tasks work over a single form of representation, designated by DIR - DOLPHIN Internal code Representation, that is based on the Register Transfer Language and more precisely on the model used by the RTL System [6] (which is also a framework for compilers development).

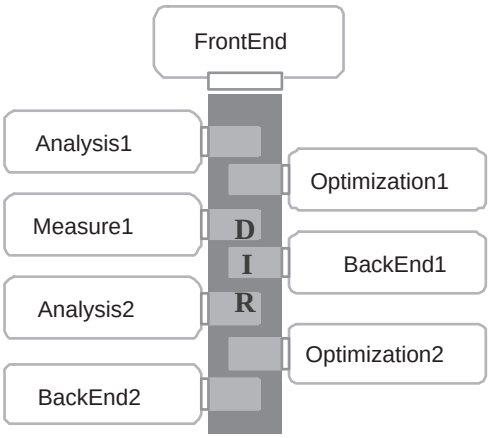
Conceptually, DIR is a specification of some generic elements that can be found on the middle-level forms of CR (where the code is purportedly independent of the details of the source language and the compiler target architecture). In practice, consists on a set of C++ classes that can be derived to build the CR. DIR classes supply several interfaces, that permit different levels of control over the CR. Many of these classes are sets or aggregations of object from other classes, which allows to obtain a hierarchic organization of the CR, where each intermediate node is a full or a partial abstraction of the CR. The highest level of abstraction is represented by the class *Program*, that encapsulates the code submitted to the compilation; the lowest levels are represented by objects from classes like *Expression* or *DataTransfer*, which have practically a direct correspondence with the instructions of the output code (assembly or binary). In the middle there are things like *Function* classes, that represent functions or procedures; or like *CFG* classes, that represent the Control Flow Graph's. Some classes, like the *IdentTable* (Identifier Table) are transversal to all levels of abstraction.

A characteristic that differentiates DIR from other forms of CR, is to allow, by one side, to create the CR using a single level of abstraction (the other levels are internally built by DIR classes), and, by the other side, it lets to handle the CR using the several levels of abstraction.

Other important part of the framework are the components, which are classified into five groups: *Front-End*'s, that make the translation of the source code into the DIR (resulting on a *Program* object); the *Back-End*'s, that convert DIR to other formats, like C, assembly, binary or simple XML/HTML [11, 12]; the *Analysis*, that compute extra data about DIR, without change the code representation; the *Optimization*'s, that produce transformations over DIR that aim the optimization of the code; and the *Measure* components, which quantify several parameters that are used to infer about the efficiency of the other components.

The top component of a compiler built with DOLPHIN framework, is always a *Front-End*. By the middle, the compiler could have components of *Analysis*, *Optimization* and *Back-End*'s. The first two share the same *modus operandis*, they take the intermediate code, make the job, and put on the output again the intermediate code. They could be compared with a car washer machine, that takes the car, wash it, and put at the output the same car, eventually, with a better look.

For the subject of this paper, it is important to emphasize that the data computed by each *Analysis* component stays inside the component. Other components access to that data, by making a request. Figure 1 shows a schematic representation of a hypothetical compiler, that uses several components, and the code required to build this compiler using DOLPHIN framework.




```
FrontEnd fe;
fe.execute();
Program*p=fe.getProgram();
Optimization1 o1(p);
o1.execute();
Measure1 m1(p);
m1.execute();
BackEnd1 be1(p);
be1.execute();
Optimization2 o2(p);
o2.execute();
BackEnd2 be2(p);
be2.execute();
```

Figure 1. Schematic representation of a hypothetical compiler and the code required to build it with DOLPHIN framework.

4. Design of the architecture

The goal established for the architectural design of DOLPHIN framework was to make life easier for all the potential users, but allowing simultaneously to develop efficient and high quality compilers. To accomplish this goal, we based our strategy into a single principle: promote, whenever possible, the reuse of the code (components). Why? One simple reason is that the components reuse reduces the length of code that must be written and, consequently the sources of errors and the maintenance of the framework.

But notice that the type of reuse that we are talking about here, consists essentially into reapply the process implemented by the component, which is also the type of reuse implicit to the conception of framework. But sometimes it will be necessary, or at least desirable, to reuse the data computed by the component and not the component itself. For us, this is also a form of reuse, that has several advantages, such as: allows to reduce the number of components used to build a compiler and consequently the quantity of code that must be executed during compilation.

The reuse of data is typical of the *Analysis* components. The particularity of such components is that they compute extra data about the intermediate code, which is retained inside the component. Consequently, an *Analysis* component can be reused as a process, which means apply again the process over the intermediate code; or it can be reused as a source of data, supplying data that was previously computed and kept inside the component. Implicit to both forms of reuse is a dependency (between the component and the component user). The first will be designated as a *functional dependency*, while the second one, that is a specialization of the first, will be designated as *data dependency*.

But the reuse of components, namely when there are data dependencies, raise some new problems. For us, the user should not need a deep knowledge about the framework, to safely use it. The idea is to explain the essential things, like what a component do, but avoid all the other things that are not relevant. For example, the user should not have to know:

- How the component is implemented?
- Which are the pre-conditions to use it?
- Is or is not necessary to use previously other components?
- Is safely to use the component on the present state?
- Is the use of the component redundant?

Notice that all these questions are quite relevant, especially when we think that a complete task may be composed by several components, which is quite common and even desirable.

To simplify the use of the components, it was defined a very simple interface, designated by *Component*, that should be implemented by all components. By now, it is enough to say that this interface defines two methods: the *execute()* and the *update()* (see figure 2). The first forces the execution of the component, while the second requests an update of the component (the component is executed only if it is outdated). As we will show, to use a component it is enough to instantiate it and call *execute()* or *update()* - the user does not have to worry about with any of the questions presented above.

Of course, that sometimes is necessary to deepen the knowledge about the framework, namely to develop new components. But even at these circunstances, the user effort is quite reduced, simply because the user is guided to reuse the components that are already implemented to build its own components; but also because DIR was conceived to facilitate the implementation of the component parts that are not feasible to be replaced by the framework components (as is showed at [13]).

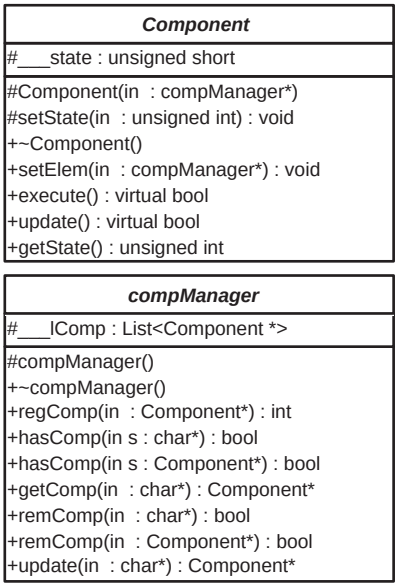


Figure 2. *Component* and *compManager* interfaces.

4.1. Dependencies management

Reuse components that are data dependent creates new problems, namely related with the scope of the components and with the validity of the data. Typically, a component is executed over a concrete type of element of the CR. The problem is that during the compilation, it could be necessary to compute several forms of analysis for each element and, the number of elements of the CR can be very high. Under these conditions, it could be very difficult to reuse the components, specially to identify if a concrete component was or was not already applied to a particular element of the CR. The problem could be even worst, if the user detects more than one component. Which one should be used? Are all the components valid? Is there any valid component? These are the reasons why the framework design should impose some organization that facilitates the components reuse.

To understand a little better the situation, imagine that we have a component *C2* that is data dependent of component *C1*. At the development of *C2*, we could force the creation of *C1*, but if *C1* was previously used, we would not be reusing it. *C2* should be able to determine if *C1* was already used and, if so, how to get access to *C1*. This could be solved, forcing all components to be declared as global variables or to belong to a global array/component. Of course, that this solution is not pretty, since exposes to many components to the end-user and could raise some difficulties at the component management.

As was told DIR allows to organize the CR as a hierarchic structure, where the root represents the whole program and the intermediate nodes represent partial abstractions of the CR. Each of these intermediate nodes is represented by a DIR class (CR element). Our experience, led us to conclude that most of the times a component do not require the whole CR to accomplish its task, normally it is enough to use part of the hierarchic structure, which corresponds to a concrete element of the CR (that can aggregate other elements). We also realize that components that are data dependent, normally, make use of the same level of abstraction (same type of CR element).

Based on these conclusions, it was defined a protocol to control the use of the components. The first rule of this protocol is that all components should be registered; the second rule is that each component should be registered into the CR element that contains all the necessary information for its execution. To allow the registration, was appended a new method to the interface *Component*, the *setElem()*. This method receives the CR element where the component should be registered. Figure 2 shows the full representation of (*Component*), including the constructor and destructor (since the components are implemented as objects). The *modus operandis* of the registration is quite simple: if the element *E* contains the necessary information for the execution of the component *A*, then *A* should be registered into *E*, calling *A->setElem(E)* or passing *E* at the constructor of *A* (*TypeA A(E)*). *Component* contains two other methods, the *setState()* and the *getState()*. The first is a protected method that allows to set the state of the component (*UPDATED*, *OUTDATED*, ...); the second is a public method that allows to get the state of the component.

Now, that was defined the interface to use the components, it is time to introduce the interface for the CR elements that have to manage the registry of the components. This interface is designated by *compManager* and is represented at Figure 2. Essentially, what this interface proposes is the methods to deal with the data structure, where are maintained the component registries. These methods are: *regComp()*, to register a component; *hasComp()* to test if there is a concrete component; *getComp()* to get a component; *remComp()* to remove a component; and the *update()* to request the update of a component. Notice that for most of these methods, the component is identified by a string (id of the component), this was the solution found to make reference to a component without have to know the exact instance of the component. Also notice that this solution works fine, because the CR elements do not accept registration of two or more components of the same type (does not make sense to have more than one component of the same type applied to the same CR element).

Now, with this very simple mechanism, it was possible to define a protocol to reuse the components, that can be illustrated with the next example: imagine that *A* requires data

```
A::A(E *e){
  …
  setElem(e);
  …
}
void A::setElem(E *e){
  …
  _e=e;
  if(_e)
  //Component Registration
    _e->regComp(this);
  …
}
bool A::execute(){
 if(_e){
  bool st;
  B*c2=_e->update("B");
  if(!c2){
   c2=new B(_e);
   if(c2)st=c2->update();
  }else st=true;
  if(st)
   if(execute0(c2)){
    setState(UPDATED);
    return true;
   }
 }
 return false;
}
```

Figure 3. Code executed by the components to make the registry and to guarantee the pre-conditions (that the required components are conveniently registered and updated).

from component *B*, then *A* may inquire *E* to see if it contains any registered component of type *B*. If so, *A* may request an update of *B*; if not, *A* may request for a registration of a new object of type *B* into *E*. Notice that it is important to register all components, since it is not possible (neither conceptually desirable) to know previously if a component will or will not be reused. Both proceedings, component registration and data dependency test, are represented at figure 3.

One important characteristic of this protocol, is that all the details are hidden from the component user, that does not need to know that component *A* is data dependent of component *B*; or if *B* was applied to *E*; or if *B* is or is not updated. And if you notice, figure 1 does not contain explicit references to the *Analysis* components. It is possible

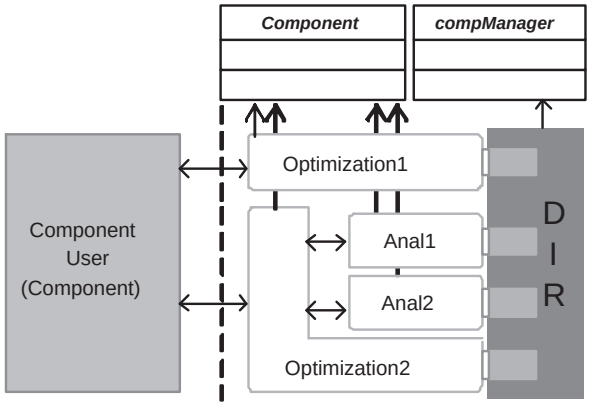


Figure 4. Relation between components, component users, CR and interfaces.

to explicitly use these components, but normally they are introduced automatically by the other components, like *Optimization*'s and *Back-End*'s. So, the user only has to declare and make reference to the effective components, like the *Front-End*, the *Optimization*'s and the *Back-End*'s. Notice that the *Component* interface is also used internally (by the CR elements). Figure 4 illustrates how the protocol and interfaces are related.

But the association of a component to a specific type of CR element, may raise some difficulties for the component users, namely when they work with a level of abstraction that is higher than the one used by the component. In these cases, the component user is forced to be aware of unnecessary details about DIR and about the CR structure. To spare the user, the protocol recommends the implementation of methods or new components that generalize the application of the original component to the higher levels of abstraction. For example: a Reach Definition Analysis (*RDA*) is typically applied to objects of type *Function* that can appear inside other *Function* objects or inside a *Program* object. The easiest way to deal with the code representation is to work directly with the object *Program*, like is showed in figure 1, so it may be useful to build a class that can receive the object *Program* and applies the *RDA* to all sub-elements that require this type of analyses (instantiating and registering the necessary *RDA* components into the *Function* objects). This example is illustrated at figure 5.

4.2. Controlling the framework consistency

Unfortunately, the solution presented until now is not enough to safely reuse the components. The problem is (again) the internal data of the *Analysis* components that can become outdated if meanwhile it is executed a component, like a code optimization, that changes the CR. It could

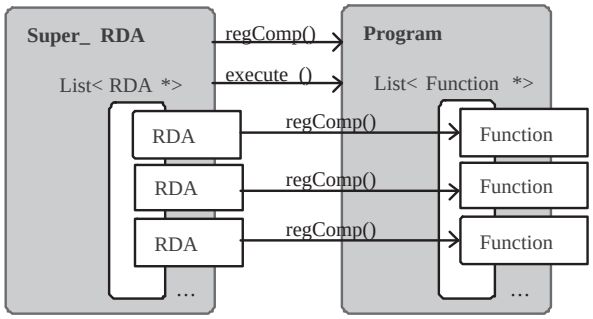


Figure 5. Generalizing the application of a component.

be very difficult to detect if the data is, or is not, updated and, consequently, to know if is safe to reuse the component. So, it is quite crucial to control the state of components internal data.

Making a sketch of the problem, we have: *Analysis* components that compute data based on the CR; and *Optimization* components, that produce changes on the CR. Changes that can invalidate the data computed by the *Analysis*. So, if the data computed by the component is consistent with the CR, then we say that the component is updated, by the other side, if the data is not consistent with the CR then the component is outdated. To develop a solution that guarantees the consistency of the component state, we analyzed three possible solutions:

1. Delegate on the Analysis components the responsibility of control their own state. For example: capturing the "state" of the code representation before the execution, and later use it to compare if the internal data is or is not updated;
2. Delegate on the *Optimization*'s the responsibility of notify the *Analysis* components (eventually, via CR);
3. Delegate on the code representation the responsibility of notify the *Analysis* components, whenever the CR suffers a modification.

The first alternative can be very complex or at least as complex as re-compute everything again, depends mostly of the nature of the component. But for us, has an unacceptable drawback: it does not make life easier for the users, especially for the ones that have to implement their own components (requires a deep knowledge of the CR). The second alternative has another inadmissible problem (which is common to the first alternative): the reuse of the components is not secured by the framework, but by the component builder. Imagine now the problems that may occur if the developer does not implement correctly the notification mechanisms; or, even worst, if the developer does not know

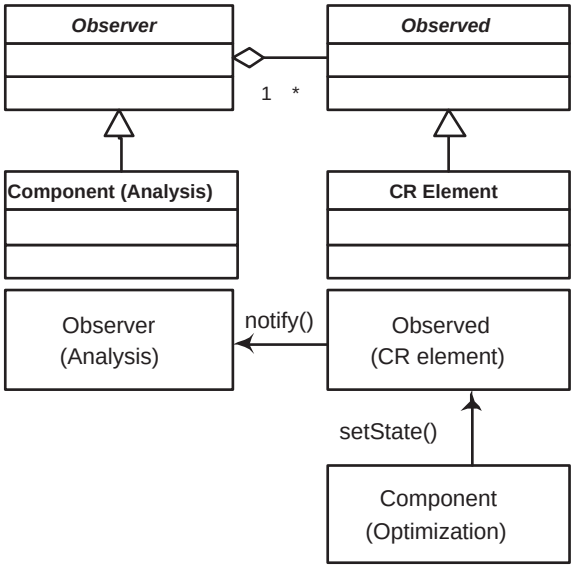


Figure 6. *Observer* design pattern and its application to DOLPHIN framework.

that such mechanisms are required. So, for us only the third solution was acceptable.

To implement the solution, we resort to a design pattern designated by *Observer* that defines the way to solve problems that have "*a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated*", page 293 of [5], which is basically the problem that we have: components that should be notify and, eventually, updated, whenever the CR is modified.

Figure 6 shows the structure of the *Observer* design pattern and the adaptation to the present situation. The idea is very simple, when a component (*Optimization*) executes an operation that changes the content of a CR element, this one sends a notification to the components. Of course that to be notified, a component must be registered on the CR element.

Figure 7 shows the interfaces defined for the entities of the *Observer* design pattern: (*Observer*, *regObserver* and *Observed*). The *Observer* interface defines the methods that the components have to implement to be notified (*notify()*). The *regObserver* is the interface of the CR elements that have to process registration requests, which are not necessarily the observed objects. The *Observed* interface, that derives from *regObserver*, defines the methods necessary to access the data structure where are effectively made the registry. So, if a CR element could be observable than it should implement the *Observed* interface; a CR element that is not observable but contains other CR elements that are observable, should implement the interface *regObserver*, that es-

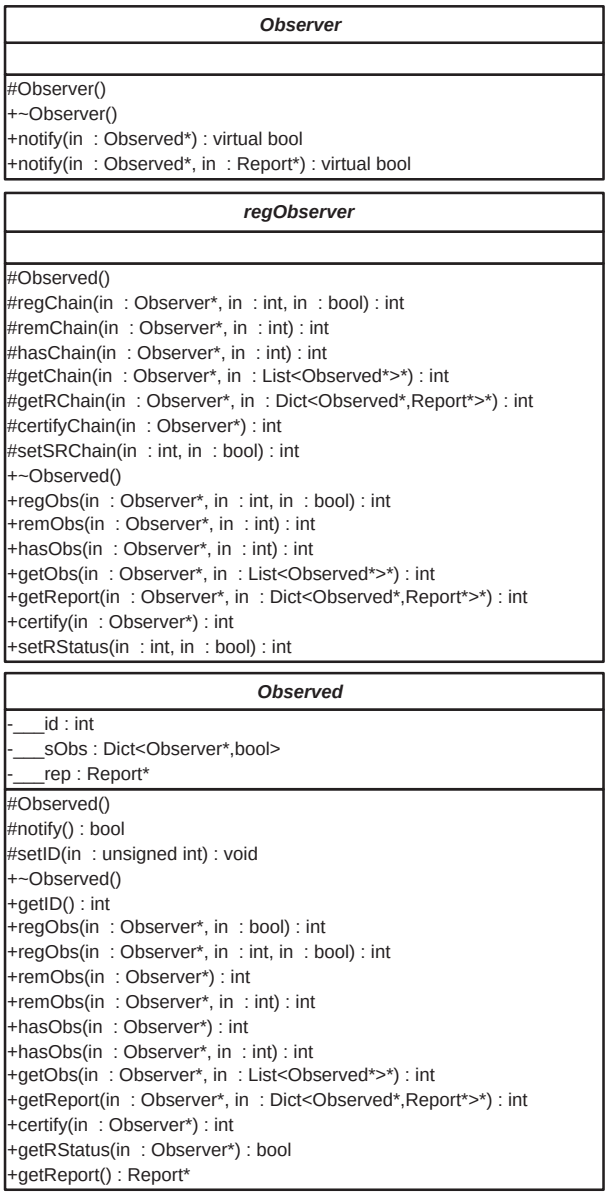


Figure 7. *Observer*, *regObserver* and *Observed* interfaces.

sentially allows to a CR element to dispatches the registration requests to its sub-elements.

Now we have two types of registrations: one to register the components into the CR (defined by the *Component* and the *compManager* interfaces); and other to register the components to observe the CR elements (defined by the *Observer* and *Observed* interfaces). Let's designate the first one by *component registration* and the second one by *observer registration*. This raises some questions, like: could both types of registrations be done simultaneously and in the same manner? Is it possible to implement both registrations into a single step? Should they be done using the same element of the CR?

Well, conceptually there are differences between them that help to answer these questions. But the main one is that the internal data of an *Analysis* component could depend only of some parts of the element where was done the *component registration*. Notice that these parts are themselves elements of the CR. For example: imagine a *Function* object that contains a *CFG* (Control Flow Graph object), an *IdentTable* (Identifiers Table), and other sub-elements; now imagine that there is a *CFA* component (that computes information about the control flow) registered on that *Function*. To compute the component internal data it is only required the *CFG*, all the other sub-elements are irrelevant. Now, imagine that the *CFA* component makes an *observer registration* into *Function*, and that *Function* can detect the modifications occurred in *CFG*. What will happen? *CFA* will receive the notification from *Function* and make an update. Now, imagine that the modification occurs into *IdentTable*, what will happen? *CFA* will receive the notification and make an unnecessary update. So, make both registrations over the same element is a specific situation, the common is to make the *component registration* into an element *E*, and one or more *observer registrations* into the sub-elements of *E*.

Now remember what was said about supply components that generalize the application of other components to more abstract elements of the CR (see section 4.1). With the *observer registration*, we have the inverse problem. The user should not be obligated to use a lower level of abstraction to make the register. For example, the register of *CFA* into *CFG* should be allowed using *Function*, avoiding that *CFA* has to deal directly with the internal elements of *Function*.

To implement a solution for this problem, it was used other design pattern, the *Responsibility Chain*, that allows to "*chain the receiving objects and pass the request along the chain until an object handles it*", page 223 of [5]. With this solution there are two implicated CR elements: the CR element that should be maintained under observation, here designated by *target element* (corresponds to the observed element), and the one where the component makes the register (the *register element*). Now, when a component makes the registration (to be an observer) into the *register element*, it must identify the *target element*. This is done setting a bit (flag) to the *unsigned int* parameter of the *regObs()* method (purposely defined this way to allow the identification of several target elements into a single register request). When a CR element receives the registration request, tests the *unsigned int* parameter to see if it is the *target element*. If so, the registration is done on that CR element; if not, the CR element dispatches the request to the lower level elements.

Figure 8 shows a sequence diagram with the *component*

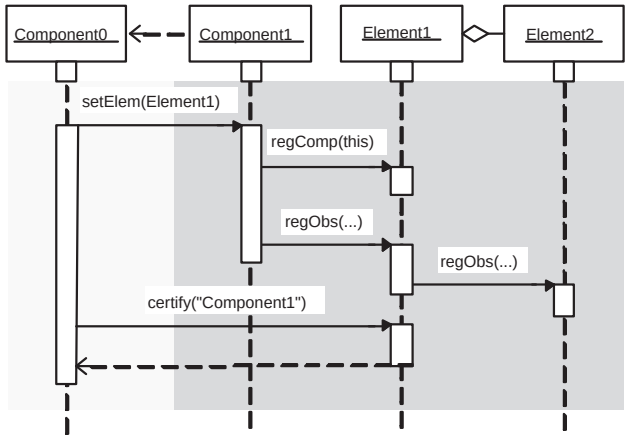


Figure 8. Sequential diagram of the *component registration* and *observer registration*.

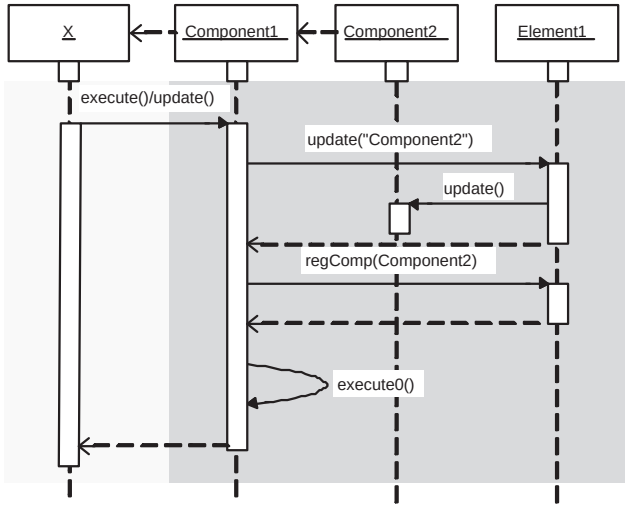


Figure 9. Sequential diagram of the component execution.

registration (*regComp()*) and the *observer registration* (*regObs()*). The diagram also shows the *certify()* method. The solution presented until now contains a lack: a component could refuse to do the *notification register*, and DOLPHIN framework relies on that mechanism to guarantee a safely use of the components. Imagine what happens if we are using an *Analysis* component believing that its internal data is correctly updated, but the component is not even register as an observer (and, as a consequence, does not receive any notification). It is not possible to force the developers to respect our protocol, but we can supply a mechanism for the users of the component, test how safely is to use a component. The *certify()* method allows to determine which are the CR elements observed by each component. Notice that it is possible to have a component that works well, but is not automatically updated. In these cases the user should force the execution of the component instead of the update.

Figure 9 shows the sequential diagram for the component execution. *X* represents a CR element or a component that is data dependent of *Component1*. *X* requests an update of *Component1*, calling the *execute()* or the *update()* method, and waits for the confirmation. *Component1* by its side, requests to *Element1* the update of *Component2* (*Component1* is data dependent of *Component2*) and waits for the confirmation. If *Element1* contains an instance of *Component2*, sends it an update request and returns that instance to *Component1*. But if the *Element1* does not contain any instance of *Component2*, then returns a null pointer. In this case, *Component1* may instantiate an object of type *Component2* and make its registration into the *Element1*. When *Component1* obtains the confirmation from *Element1* that *Component2* is registered and updated, it proceeds the execution (*execution0()*). At the end sends a confirmation to *X*.

Figure 10 shows the sequential diagram of the notification. *Component3* modifies *Element1*, which sends a notification to the *Observer*'s (*Component1*). The diagram also shows the *report()* method. Imagine that *Component1* is able to make a localized update (*update0()*), avoiding to recompute everything again. Probably, *Component1* can do this optimized update if knows which were the operations executed over *Element1*. So, it was defined a class, designated by *Report*, that can be used by the CR elements, that implement the interface *Observed*, to register the operations executed over them. For each operation, it is saved the used method, the returned value and the value of the parameters. A component that is notified, can request the *Report* object using the *report()* method. Notice, that the implementation of the *Report* class only was possible because the CR elements of DIR were implemented as encapsulated classes.

5. Conclusion

This paper was concerned with DOLPHIN, a framework for compilers development, and with its architecture - principles, structure, elements and their interactions. Using essentially five interfaces (*Component*, *compManager*, *Observer*, *regObserver* and *Observed*), it was possible to design architecture of DOLPHIN framework in order to accomplish all the established requirements. It is important to emphasize the advantages obtained from this approach for the users and developers of DOLPHIN framework:

- Reinforces the implementation of modular compilers;

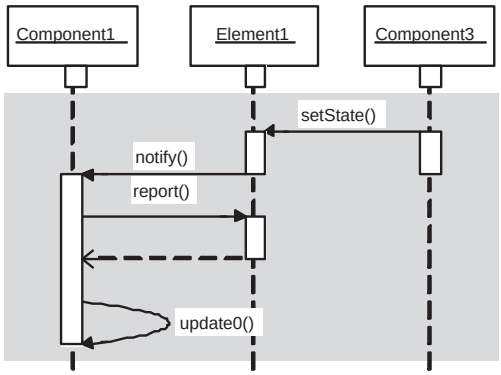


Figure 10. Sequential diagram of the component notification.

- Reduces the framework redundant code, sources of error and maintenance;
- Minimizes the implementation effort of new components;
- Reduces the inconsistencies among components and between components and intermediate representation;
- Minimizes the code implemented by the users and the compiler execution time.

So, even before start to talk about compilers technology, we were able to obtain a consistent solution, completely user oriented, which is not restricted to frameworks for compilers development. For us, the user friendliness of this type of applications is, at least, as important as any other characteristic. But it assumes a special meaning, when we look for the DOLPHIN project [10], that integrates several web services based on the DOLPHIN framework, namely a virtual laboratory for compilers development, strongly oriented for pedagogical purposes. Of course, that it is difficult to quantify the friendliness of our framework when compared with other similar tools. But we can always argue that we dealt with this question since the beginning of the project, it is intrinsic to the design of DOLPHIN framework. Anyway, we hope very soon to start receiving some feedback from the users, and at that time we will probably be able to quantify and compare DOLPHIN framework with other solutions.

But the work does not end here, DOLPHIN project includes the conception and implementation of several tools specially devoted to the framework, namely: a browser integrated with a help system, a textual and a graphical editor, tools to visualize the CR, and even a meta-language to specify the compiler so that it becomes possible, using the framework, to generate the full compiler.

References

[1] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy, and C. Sapuntzakis. An overview of the SUIF2 compiler infrastructure. Technical report, Computer System Laboratory, University of Stanford, Portland, USA, August 2000.

[2] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy, and C. Sapuntzakis. The SUIF program representation. Technical report, Computer System Laboratory, University of Stanford, Portland, USA, August 2000.

[3] U. Assmann. Graph rewrite systems for program optimization. *In ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(4), 2000.

[4] C. Fraser, R. Henry, and T. Proebsting. BURG - Fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, 1991.

[5] E. Gamma. *Design Patterns - Elements of reusable object-orient software*. Addison-Wesley, 1995.

[6] R. Johnson, C. McConnell, and J. Lake. The RTL System: A framework for code optimization. In *Proceedings of the International Workshop on Code Generation*, pages 255–274, Dagstuhl, Germany, May 1991.

[7] F. Martin. PAG - An efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[8] P. Matos. DOLPHIN framework. Technical report, University of Minho, October 2002.

[9] P. Matos. DOLPHIN: A system for compilers development, teach and use. Technical report, Universidade do Minho, Braga, Portugal, October 2003.

[10] P. Matos and P. Henriques. DOLPHIN-COMPLAB: A virtual compilers laboratory. In *Proceedings of Second International Conference on Multimedia and ICTs in Education*, pages 1637–1641, Badajoz, Spain, December 2003.

[11] P. Matos and P. Henriques. DOLPHIN-FEW - An example of a web system to analyze and study compilers behavior. In *Proceedings of IADIS International Conference e-Society*, volume 2, pages 966–970, Lisbon, Portugal, June 2003.

[12] P. Matos and P. Henriques. A solution to dynamically build an interactive visualization system to the DOLPHIN-FEW. In *Proceedings of IASTED International Conference on Visualization, Imaging, and Image Processing*, pages 868–873, Benalmdena, Spain, September 2003.

[13] P. Matos and P. Henriques. DIR - A code representation approach for compilers. In *Proceedings of IADIS International Conference on Applied Computing*, Lisbon, Portugal, March 2004.

[14] F. W. Schroer. *The Gentle Compiler Construction System Manual*, 1997.

[15] R. Stallman. *Using and Porting the GNU Compiler Collection (GCC)*. iUniverse.com, Inc, 2000.