A framework to optimize compilation task

Paulo Jorge Matos

Instituto Politécnico de Bragança
Departamento de Informática e Comunicações
Campus de Santa Apolónia,
5300 Bragança, Portugal

Abstract. Nowadays, compiler construction is supported by several tools, many of them are based on frameworks, composed by several components that can be combined or instantiated to build new components or even entire compilers. This paper is a software engineering exercise applied to the compiler construction tools. It is used a concrete framework for compilers development - the Dolphin, that supplies several components that work over a single code representation model, to show that the simple composition of such components is not enough. It raises serious obstacles that make the compilers construction more arduous.
The exercise evolves for a reformulation of the framework, resulting on an independent architecture that could be adapted to similar frameworks. Defining the behaviour and the relationship of several elements of the framework, this architecture allows to surpass most of the obstacles, but also releases the compilers developer from some duties, making the construction of compilers more accessible. It also promotes the development of compilers that are more stable and efficient that is, they can run faster using fewer resources.

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. Nowadays, the source languages are quite more powerful and distant from the syntax and the paradigm of the output language, which raises more difficulties for the translation of the source language; and the computational architectures (target machine + operating system), are more elaborated and demanding, making the generation of the output code more difficult. 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, but also 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), but also about microprocessor architectures. Attending to the complexity inherent to the compilers development, many tools appeared to help on this task. Most of them perform a very nice job, specially the

ones that are dedicated to the development of a single compilation task; it is the case of the Lex [4], Yacc [17], BURG [1], NJMCT [5], and many others. But it is much harder to conceive and implement a tool that supports the full development of compilers. The compilation process evolves many different techniques and solutions. Create a tool that can accomplish all these techniques and solutions is a very complicated task. But there are some successful examples, like: SUIF Compilers System [3], GENTLE [15], and some others.

There are two main approaches used on the implementation of such tools:

Generation of components: Using a specification language, the user describes the variants of the compilation task. The solution is generated by the tool, based on the information supplied by the specification;

Code reuse: The solution (component) is obtained, extending or simply reusing pre implemented components.

Dolphin [6] is an example of tool that makes use of both approaches. Essentially, it is a data centric framework that was conceived to build modular compilers. It is composed by several ready to use components that work based on a single model of code representation. But also supplies some generation tools, to develop components, like the ones related with the source language or with the computer architecture. These generated components could be integrated into the framework and used with the native components to produce full compilers. The native components are normally used to implement the middle-level tasks, the ones that are independent the source language and the computer architecture characteristics. It is also important to explain that a single component could accomplish several compilations tasks.

Dolphin is a data centric framework since all components work over a single form of code representation, designated by intermediate code representation (ICR). It is the *Dolphin* Internal Representation - DIR [10] that defines the type of elements used on the ICR. DIR is based on the Register Transfer Language and more precisely on the model used by the RTL System [13] (another framework for compilers development). Conceptually, is a specification of some generic elements that can be found on the ICR (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 used to build the ICR.

DIR classes supply several interfaces that permit different levels of control over the ICR. Many of these classes are sets or aggregations of other classes, as consequence, the construction of the ICR results on a hierarchic data structure (a tree), where the intermediate elements are abstract entities that represent parts of the program. The root is typically an object of type *Program* or *DIR*, that encapsulates the whole code submitted to the compiler. The lowest levels are represented by objects derived from *Expression* and *DT* classes, that are strongly related with the output code (assembly or binary). In the middle, there are objects like *Function*, used to represent functions or procedures; or like *CFG*, used to represent the Control Flow Graph. The use of different levels of abstraction allows to choose the one that is more adequate for the implementation and execution of each component.

Dolphin framework contains five categories of components: *Front-End*'s, that translate the source code and build the ICR, instantiating objects from DIR; *Back-End*'s, that convert the ICR into other code formats, like C, assembly, binary code or even XML [7]; *Analysis*, that compute extra data about the ICR used to support some code optimizations and back-end tasks; *Optimizations*, that transform the ICR normally to improve the quality of the output code; and the *Measure* or *Inspection* components, which compute several parameters that are used to infer about the efficiency of the compilation process.

Notice that the *Front-End* and the *Back-End* components include several compilation tasks. ***Dolphin*** framework does not supply tools to implement the *Front-End* components, since there are lots of good solutions that can be used to build the lexical, syntactic and semantic analyzers (like Lex [4], Yacc [17], Eli [12], JavaCC [14]. But ***Dolphin*** framework supplies an integrated tool to build *Back-End* components capable to generate assembly code. These components include an optimal instruction selector and a register allocator (user might choose between local or global allocation).

ICR is the mechanism used to pass the information among the components. It behaves like a pipeline, over which the components work. To use a component, it is only necessary to perform two, eventually, three steps:

- Get an instance of the component (components are implemented as C++ classes);
- Make the registry of ICR into the component (instance). This could be done during the instantiation of the component or later using the *bool setElem(DObject*)* method;
- Execute the component (instance).

As it is possible to confirm by the Fig. 1, it is very simple to build a compiler reusing the components of ***Dolphin*** framework. Of course, that the framework does not supply all the components that user might need. So, sometimes user will have to implement their own components which could be done using the tools supplied with the framework or even using external tools.

This very brief description intends to show that ***Dolphin*** framework is conceptually a very simple solution, based on the notion of components and code reuse, implemented as a data centric framework.

This is the starting point to show that the solution used by ***Dolphin*** framework, which is apparently simple and functional, presents several drawbacks that complicate substantially the development of compilers, namely when is important (and is always important) to implement efficient compilers. The damage will not be on the code generated by the compilers, but on the compilation process that will be less efficient (slower and requiring more resources).

This paper shows the obstacles that users must surpass to implement efficient compilers and then proposes an architecture (set of solutions), that defines the behaviour and the relationship of several elements of the framework. This architecture allows to surpass most of the obstacles, but also releases the compilers developer from some duties, making the construction of compilers even more

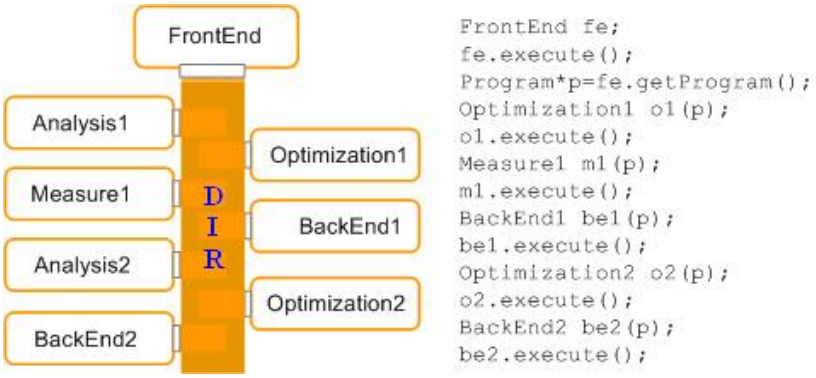

Fig. 1. Specification of a hypothetical compiler built with ***Dolphin*** framework.

accessible. It also promotes the development of compilers that are more stable and efficient that is, they can run faster using fewer resources.

The next section exposes the problems that are subjacent to solutions like ***Dolphin*** framework that are based on components reuse. Section 3, describes the proposed architecture, 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 4, the conclusion is drawn.

2 Using *Dolphin* framework

A compiler produced using the ***Dolphin***, typically contains one *Front-End* and one or more components from other types, that may include several *Back-End*'s and even, several times the same component. Many components have common proceedings, that are typically used to accomplish some requirements, but also to execute other tasks (before, during and after the execution of the main process). As it is natural on good practice of software engineering, many of these proceedings are implemented apart, to be reused. At ***Dolphin*** they are implemented as components of the framework. They will be designated here generically by *support components*. Components that make use of the support components will be designated by *main components*. And, of course, there is a relation of dependency between them, which will be designated by a *functional dependency*.

The decomposition of a solution into several components, aims to minimize the quantity of code and, as consequence, the potential source of errors and the costs of maintenance. In some cases, simplifies the implementation of the components, since it is possible to reuse other components to implement new ones. And also makes possible to implement more elaborated components. However, this solution of decompose a solution into several components, that apparently is very simple and inconsequent, in practice raises some problems, namely to implement efficient compilers.

2.1 Components reuse

If a component is implemented without reuse other components, then it will be independent and its execution obeys only to the sequence of the compilation process. The restrictions to use the component are essentially structural. For example, the execution of a *Front-End* component must be done before the other components.

The decomposition of the solutions into several components creates functional dependencies, which means that the execution of a main component might depend of the execution of one or more support components. The support components could be executed before, during or after the main component.

This kind of dependency, that results from the fact of a solution be decomposed into several components, does not raise big problems. If each main component includes the necessary support components, everything will work exactly the same way (like an independent component). In this case, we say that the support component is implicitly used, since it is inserted into the compiler by a component and not by the compiler developer.

The implicit inclusion of the components has the advantage of hide the support components from the users (compilers developer), simplifying the compilers specification (type and sequence of the components used to build a compiler). Example 1 shows the use of this form of inclusion.

Example 1

The Single Static Assignment (SSA) form is used on the ICR to make the implementation of the components more accessible, namely the code analysis and optimizations routines. ***Dolphin*** framework supplies a component to convert from the normal form to the SSA form, the *cnv2SSA*. This component reuses other components, like it is represented at Fig. 2. If the support components are included implicitly by the *cnv2SSA*, the user does not have to know nothing about them, namely how to use them. It is enough to instantiate and execute the *cnv2SSA* as it is showed at Fig. 3.

$\oint$

It is possible to observe by Fig. 2 that some components are used to support more than one component, for example *DFrontiers* supports *cvn2SSA* but also *IDFrontiers*. Using the implicit inclusion means that there are two instances of *DFrontiers*: one that supports *cnv2SSA*; and other that supports *IDFrontiers*.

The implicit inclusion of the components is particularly severe for the compilation process. The problem is that we are reusing the code, but not the process executed by the code, as consequence, some components will have more than one instance, that will be executed at least once. And this contributes directly to deteriorate the compilation time.

The compilation process gets even worse, when the support components aim to compute and supply data about the ICR (*Analysis* components). The data is computed, held and maintained by the support component. It is that data

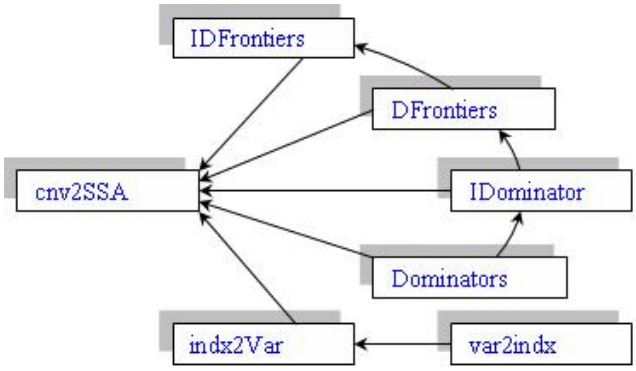

Fig. 2. *cnv2SSA* and the support components.

```
(1)     littleC fe;                 // Front-end instantiation
(2)     DIR *d=fe.execute();        // Front-end execution
(3)     …
(4)     cnv2SSA cnv(d);             // cnv2SSA instantiation
(5)     …
(6)     cnv.execute();              // cnv2SSA execution
(7)     …
```

Fig. 3. Partial specification of a compiler using components implicit inclusion.

that will be used by the other components (and not the component itself). So, replicate instances of these components, means replicate the data computed by them. The consequence is a compilation process that will waste more resources (memory).

This situation can be illustrated by the example of Fig. 2. *cnv2SSA* converts the ICR from the normal to the SSA form. It gets the ICR (at the normal form), processes it, and puts at the output again the ICR (now at the SSA form). No information is held by the component. But the same does not happen for the support components. Each instance of the support components, will get the ICR and compute the data processing the ICR (without change it). For example, *Dominators* uses the ICR to build a dictionary that contains, for each node n of the control flow graph, the set of nodes that dominate n [18]. So, for each instance of *Dominators*, there will be a dictionary with exactly the same data of the other instances of that component (that are applied over the same element of ICR).

We designate this kind of dependency, where the main component depends on the information computed by the support component, as a *data dependency* between components.

One way of improving the compilation process is to include explicitly the components into the compilers specification. But this means that the compiler developer has the responsibility of instantiate and execute all components (main and support components), avoiding the replication of the component instances.

But even this solution presents several drawbacks.

```
(1)     littleC fe;                          //Front-end instantiation
(2)     DIR *d=fe.execute();                 //Front-end execution
(3)     …
(4)     var2Indx vi(d)                       // var2Indx instantiation
(5)     indx2Var iv(d);                      // indx2Var instantiation
(6)     Dominators dom(d);                   // Dominators instantiation
(7)     IDominator idom(d);                  // IDominator instantiation
(8)     DFrontiers df(d);                    // DFrontiers instantiation
(9)     IDFrontiers idf(d);                  // IDFrontiers instantiation
(10)    cnv2SSA cnv(d);                      // cnv2SSA instantiation
(11)    …
(12)    vi.execute();                        // var2Indx execution
(13)    iv.execute(vi);                      // indx2Var execution
(14)    dom.execute();                       // Dominators execution
(15)    idom.execute(dom);                   // IDominator execution
(16)    df.execute(idom);                    // DFrontiers execution
(17)    idf.execute(df);                     // IDFrontiers execution
(18)    …
(19)    cnv.execute(iv,dom,idom,df,idf);     // cnv2SSA execution
(20)    …
```

Fig. 4. Partial specification of a compiler using the explicit inclusion of the components.

By the examples of Fig. 3 and 4 is easy to understand that, for exactly the same operation (conversion from the normal to the SSA form), the specification of a compiler using the explicit inclusion of the components is substantially more complex and larger that the one that is done using the implicit inclusion. The explicit inclusion also requires more knowledge about the components, namely: the way how they are implemented; which are the support components; by which sequence should they be executed; how to use them; which are the effects of the support components; etc. And notice that all these questions will appear recursively for the support components.

2.2 Components association

In the examples presented until now, like the one at Fig. 3, the components work exclusively over an object of type DIR. This was done to simplify the examples, but in practice each component uses the ICR element that is more adequate for its implementation and execution. So, to make the registry of the ICR element into the component, user must "navigating over the ICR to get the required element and this can only be done if the user has enough knowledge about DIR and the way how the ICR was built. Notice that at the explicit inclusion the navigation is done by the component user (the one that intends to build the compiler); at the implicit inclusion the navigation is done by the components developer.

It is also important to emphasize that a program submitted to the compiler has a single object of type *DIR* and *Program*, some dozens of objects like

Function's and *CFG*'s, but most probably thousands of low level objects, like *Expression* and *DT*. So, the components that use low level elements could have thousands of instances. Managing all these instances and the correspondent dependencies is a very complex task. With the implicit inclusion, the problem stays limited to the implementation context of the components and the one that has to manage several instances and correspondent dependencies is the component developer. But with the explicit inclusion, it is necessary to deal simultaneously with thousands of instances which is done by the user of the components (the one that most probably does not know or does not want to know nothing about the implementation of the components).

Imagine for example, that the ICR contains several elements of type A and B ($A_0, \ldots, A_n, B_0, \ldots, B_m$), that component C_A is applied over the elements of type A ($C_{A_0}, \ldots, C_{A_n}$) and component C_B is applied over elements of type B ($C_{B_0}, \ldots, C_{B_m}$), and that the instances of component C_A support the instances of the component C_B on a relation of one to one. The question is to know how associate the instances of C_A with the instances of C_B?

To answer this question, it is necessary to understand how the ICR elements are related. For example, to execute C_{B_i} is necessary the previous execution of the support component (C_{A_j}). To get C_{A_j} from C_{B_i}, it is necessary to get B_i, then get the correspondent element of type A (A_j). Both operations are possible and easy to do, it is enough to know DIR and the way how the ICR was built. But is necessary one last operation, get the instance of component C_A applied to A_j (C_{A_j}) and this is not possible using just the native mechanism of ***Dolphin*** framework. It is the users that must create and manage a dictionary that gives all the components applied to each ICR element.

2.3 Data consistency

Suppose that the registry of the components is already solved. The next question is: Has the main component guaranteed that is safe to use the support component? Since this question is particularly relevant when there is a data dependency between the two components, it should be reformulated to: Can the main component reuse safely the data computed by the support component? Notice, that we are not doubting of the implementation of the support component. The problem is to know if the data computed by the support component is still coherent with the ICR when is used by the main component. Imagine that at line eighteen of Fig. 4 is executed one or more components that aim to optimize the control flow of the program submitted to the compiler, like the *elimJumpChains* component of ***Dolphin*** framework. This is a very simple component that eliminates chains of unconditional jumps, joining nodes of the control flow graph. But it is enough to make useless the data computed by components like: *Dominators*, *IDominator*, *DFrontiers* or *IDFrontiers*. Their data, most porbably, will no longer be coherent with the ICR. The user has three alternatives:

- Always forces the execution of the support components before use them (no reuse is done);

– Executes the support components immediately before the main component, minimizing the possibilities that they became incoherent;
– With a deep knowledge about the way of work and effects of all components and controlling all instances, the user could avoid the reuse of incoherent components.

The first two alternatives are not satisfactory and the third is not humanly feasible neither desirable. Many components of frameworks like ***Dolphin***, are implemented by external collaborators, so it is not guarantee that users have access to the code or to the internal details of components implementation.

3 Design of the architecture

Once presented the most relevant problems of a framework like ***Dolphin***, it is time to think on solutions that solve these problems. There were identified three main problems for which are necessary new or better solutions:

Components association: It is necessary a more practical and efficient solution to associate dependent components;
Components reuse: It is necessary a solution that allows an easy and efficient reuse of the components;
Data coherence: It is necessary a solution that guarantees the coherence of the data computed by the support components, but that also minimizes the number of times that each instance is recomputed.

Besides these problems, it is also important to make the whole solution easy to use and capable to support the development of efficient compilers.

The next sections describe the solutions found for these problems. The integration of these solutions results on an architecture. It was conceived for ***Dolphin*** framework, but it is enough generic to be used on similar tools.

3.1 Components association

As already was explained, each component is associated with an ICR element. This association is established when the ICR is registered into the component (instance). With this very simple mechanism, it is possible to the component access the ICR element. However it is not feasible to obtain a component from the ICR element. But this relation would be useful to associate dependent components. The solution proposed before was to delegate into the user the responsibility of create and manage a dictionary that allows to know which are the components associated to each ICR element. But this solution is only feasible with the explicit inclusion of the components. But this kind of inclusion contains severe disadvantages and it could be impracticable (remember that the number of instances could be huge).

This problem was solved with a very simple solution that does not change the native proceedings of ***Dolphin*** framework. When the ICR element is registered

into the component, the component is also registered into the ICR element. This one maintains a set with the registry of all components.

The implementation of this solution was done defining two interfaces (C++ classes): one for the code representation elements, designated by *compManager*; and other for the components, designated by *Component*. Figure 5 shows the UML representation of these two interfaces, with the methods necessary to the registry, but also with the methods introduced to solve other problems.

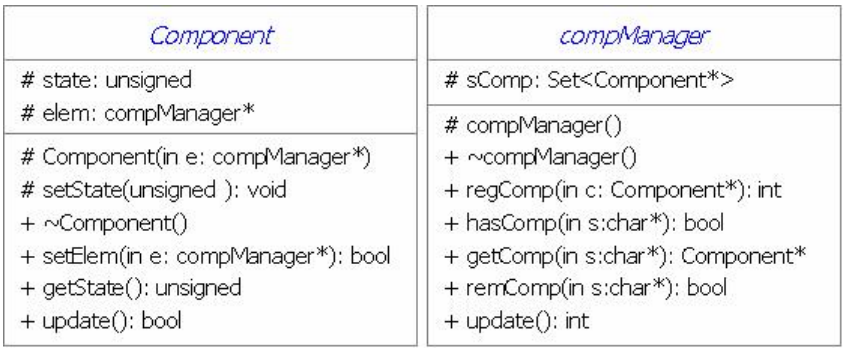

Fig. 5. Interfaces of *Component* and *compManager*.

Component interface defines four public methods, but only the *setElem(...)* is fundamental for the association of the components. Is the one that lets to make the registry of the ICR element into the component, but also the registry of the component into the ICR element. The method *execute()* forces the execution of the component and the method *update()* forces the execution if the component is outdated, which is part of the solution to solve data coherence of the components (see section 3.3). The maintenance and the management of the components is done by *compManager*, using a set that holds the addresses of the components.

Fig. 6 shows the code implemented on the constructor and on the *setElem(...)* method of the *Component* interface, to make the registry.

3.2 Components reuse

Once defined the way how to associate dependent components and establish the relation between components and the ICR elements, it is time to forward to the next problem: conceive a solution that allows an efficient reuse of the components. The solution proposed on last section contains already all the necessary features to reuse efficiently and easily the support components.

The example of Fig. 7 helps to explain how this is done. E_1 and E_2 are ICR elements (that are someway associated). C_2 is an instance of $Component_2$ that works over E_2. C_2 also requires some data about E_1, that are computed by components of type $Component_1$. To execute, C_2 has to access to E_1 (via E_2)

```
(1)   Component1::Component1(compManager *e){
(2)     ...
(3)     this->setElem(e);
(4)     ...
(5)   }
(6)   void Component1::setElem(compManager *e){
(7)     ...
(8)     this->___e = e;                          // Registry of the ICR element
(9)                                              // into the component
(10)    if(this->___e)
(11)      ___e->regComp(this);                   // Registry of the component
(12)                                             // into the ICR element
(13)    ...
(14)  }
```

Fig. 6. Proceedings for the component registry.

and using the *getComp(char*)* method, get access to C_1. If E_1 does not contain any instance of $Component_1$, then *getComp(char*)* return null. In this case, E_2 can request a new instance of $Component_1$ and makes its registry into E_1.

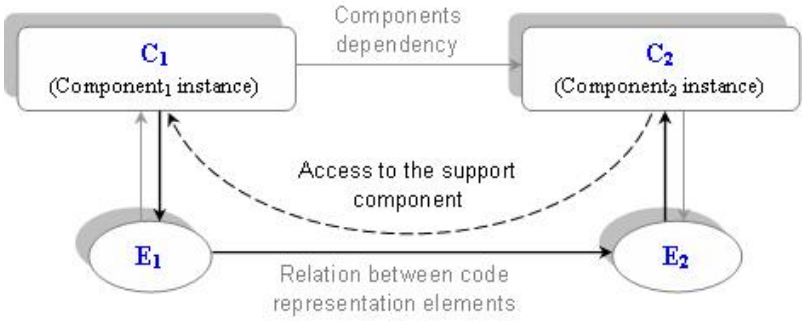

Fig. 7. Registry of dependent components.

Notice that, except for the *regComp(...)* method, all the other methods of *compManager* identify the component by a string (*char**). It could be used any other mechanism that identifies the components by the type (and not by the instance). This is fundamental, since C_2 only knows the type of the support component and not the exact instance. So, when C_2 requests an instance of $Component_1$ to E_1, it makes use of the component identifier, because it doesn't know the address or even if there is any instance of $Component_1$ registered into E_1.

Fig. 8 uses the *cnv2SSA* to show (partially) how a component implements the *Component* interface. It is given a special emphasis to the *bool execute()* method, where are executed the proceedings for reuse the support components.

The example makes use of the real ICR elements used by the evolved components.

```
(1)   class cnv2SSA : public Component {
(2)     private:
(3)       Function *___e;
(4)     ...
(5)     public:
(6)       ...
(7)       bool execute();
(8)       ...
(9)   };
(10)  bool cnv2SSA::execute(){
(11)    if(___e){
(12)      CFG *cfg=___e->getCFG();                    // Get the ICR element where
(13)                                                  // is the support component
(14)      IDFrontiers *idf=cfg->getComp("IDFrontiers");
(15)      if(!idf){
(16)        idf=new IDFrontiers(cfg);
(17)        idf->execute();
(18)      }
(19)      ...
(20)  }
```

Fig. 8. Implementation of the *Component* interface.

With this very simple solution, that defines the relation between components and ICR elements, was possible to simultaneously associate the components and supply a nice and efficient solution to reuse them. But this solution supplies other advantages, namely:

- Makes use of loosely couple association between components, that allows to dynamically replace, couple and decouple the components. For example, it is possible to:
  - Instantiate the support components only when they are effectively necessary;
  - Release the support components as soon as they became useless, but maintaining the main component.
- Still use exactly the same proceedings of the native framework, nothing change for the framework user;
- Support components could be included by the component developer (implicitly). The component user does not have to know nothing about them;
- The specification of the compilers is so simple as the one that uses the implicit inclusion of the components;
- The user does not have the responsibility of control and manage the dependencies between the components;
- Ans, as is showed at next section, this solution makes a better job for the reuse of the components than the explicit inclusion of the components.

3.3 Data coherence of the components

There is still one problem to solve, that is visible by the example of Fig 8. At line fourteen, it is used the *getComp(...)* method to get access to the instance of *IDFrontiers*, that is registered into the cfg element. If such instance does not exist then a new one is created, registered and executed by the main component (the *cnv2SSA*). The problem occurs if the cfg already contains an instance of *IDFrontiers*. The instance of *cnv2SSA* does not know if the instance of *IDFrontiers* is or is not coherent with the ICR.

Essentially, it is necessary to control the state of the components, which could be updated or outdated. To get the state of a component, it is used the *getState()* method of the interface *Component*.

The solution was implemented using the Observer design pattern 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 [2], which is basically the problem that must be solved: components that should be notified and updated whenever the state of a ICR element changes. Fig. 9 shows the structure of the Observer design pattern and the adaptation to the present situation. The *Observer* interface defines the methods that should be implemented by the observers (components that want to be notified). The *Observed* interface defines the methods that should be implemented by the observable objects (ICR elements).

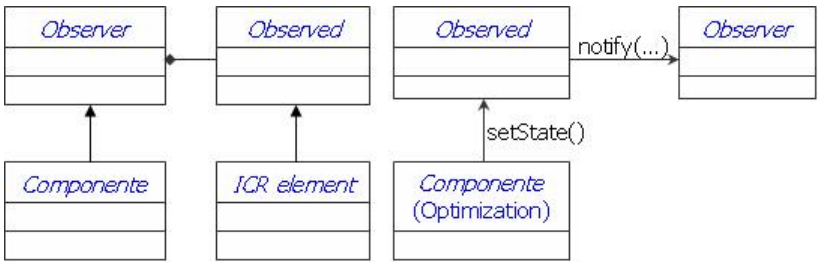

Fig. 9. *Observer* design pattern and its application to ***Dolphin*** framework.

To be notified, each component should make a registry into the correspondent ICR element (using the *regObs()*). When a ICR element suffers a change, as consequence of the execution of other component, like an code optimization, sends a notification to all observers (components). The components receive the notification with the identification of the ICR element that was changed and their state is set to outdated. Now the main component has a mechanism to know if the data computed by the support component is or is not coherent with the ICR. With this mechanism the main component could avoid an unecessary recomputation of the support components.

Besides the solutions proposed at this paper, ***Dolphin*** framework has other solutions to optimize the recomputation of the support components, supplying

information about the methods used to change the state of the support components and information about its previous state.

4 Conclusion

Using essentially four interfaces (*Component*, *compManager*, *Observer*, and *Observed*), it was possible to design the architecture able to solve most of the problems of ***Dolphin*** framework. Amazingly, with this architecture the compilation time was improved between 30% and 90% and the number of instances required was reduced between 20% and 40% (see [11]).

The use of the framework is now even more accessible, the users does not need a deep knowledge about the framework to safely use it and produce efficient compilers. For example, it is not necessary to know:

- How the components are implemented;
- Which are the pre-conditions to use it;
- Which are the support components;
- If it is or is not safe to use a component;
- If the use of the component is or is not redundant.

Besides that, the architecture presents other important advantages, such as:

- Reinforces the implementation of modular compilers;
- Reduces the framework redundant code and, as consequence, the sources of error and maintenance costs;
- Minimizes the implementation effort of new components (reinforcing the reuse of the components);
- Reduces the inconsistencies among components and between components and CR;
- Reduces the execution time and the resources required by the compilation process.

It is also important to emphasize that the architecture is generic enough to be applied to other frameworks with similar benefits. For example, the project that is used as reference to appraise our work is the SUIF Compiler System. It is a huge project that contains lots of tools, wonderful solutions, and lots of other things. But amazingly, they did not deal with problems like data coherence of the components. The SUIF ICR aggregates all the information. Even the data computed by the components (modules) is attached to the ICR as annotations. But no mechanism is supplied to control the state of the data, neither to minimize its re-computation. So, even on a huge system like SUIF, this architecture could be a surplus value.

References

1. C. Fraser, R. Henry and T. Proebsting. BURG - Fast optimal instruction selection and tree parsing. SIGPLAN Notices, (1991), 68–76.
2. E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns - Elements of reusable object-orient software. Addison-Wesley, (1995)
3. G. Aigner, et. al. An overview of the SUIF2 compiler infrastructure. Technical Report Computer System Laboratory, University of Stanford, Portland (2000).
4. M. E. Lesk. A Lexical Analyzer Generator Computer Science Technical Report, Bell Laboratories, (1975).
5. N. Ramsey and M .Fernndez: The New Jersey Machine-Code Toolkit. USENIX Technical Conference, ACM SIGPLAN, New Orleans, USA, (1995), 289–302.
6. Paulo Matos. DOLPHIN framework. Technical Report, University of Minho (2002).
7. P. Matos and P. Henriques. DOLPHIN-FEW: An example of a Web system to analyze and study compilers behavior. IADIS International Conference e-Society, Lisbon, Portugal, (2003), 66–970.
8. P. Matos. DOLPHIN: A system for compilers development, teach and use. Simpsio Doutoral do Departamento de Informtica da Universidade do Minho, Universidade do Minho, Braga, Portugal, (2003)
9. P. Matos and P. Henriques. A solution to dynamically build an interactive visualization system to the DOLPHIN-FEW. International Conference on Visualization, Imaging, and Image Processing, Benalmdena, Spain, (2003), 868–873.
10. P. Matos and P. Henriques. DIR - A code representation approach for compilers. IADIS International Conference on Applied Computing,Lisbon, Portugal, (2004), 518–526.
11. P. Matos. Um modelo arquitectnico para desenvolvimento de compiladores: aplicao framework Dolphin. PhD Thesis, Universidade do Minho, Braga, Portugal, (2005).
12. R. Gray, et. al. Eli: A complete, flexible compiler construction system. Research Report, University of Colorado, (1990).
13. R. Johnson, C. McConnell and J. M. Lake. The RTL System: A framework for code optimization. In Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, (1991), 255–274.
14. S. Sankar, et. al. Java Compiler Compiler (JavaCC).
15. Friedrich Wilhelm Schroer. The Gentle Compiler Construction System manual. (1997).
16. Richard Stallman. Using and porting the GNU Compiler Collection GCC. iUniverse.com, Inc, (2000)
17. Steven C. Johnson. Yacc: Yet Another Compiler Compiler. UNIX Programmer's Manual, vol. 2, New York, USA, (1979), 353–387.
18. T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. ACM TOPLAS, (1979), 121–141.