Automatic Prototype Generating via Optimized Object Model

Sheldon X. Liang, Lynn Zhang, Luqi
Email: {xliang, lzhang, luqi}@nps.navy.mil
Software Engineering Automation Center
US Naval Postgraduate School
833 Dyer Road, Monterey, CA 93943

[Abstract] *Computer-aided prototyping shows promise that one system under development frees designers from implementation details by executing specifications via reusable components. Ada is first choice for constructing such reusable object-oriented components because Ada95 is the only international standard programming language that supports object-oriented real-time distributed systems. But Ada has diversified object forms that are so intricate that people feel it difficult to find an equivalence of a class between Java (C++) and Ada95. In computer-aided prototyping, automatic prototype generating is facing one of key problems on how to map a well-defined prototyping specification to an executable prototype with different OOPLs. This paper addresses several key problems in automatic prototype generating with reusable object foundation based on an optimized object model. These problems include transformation of a PSDL specification to the executable system composed from componential object entities, compositional pattern enforcing interactions among components, generalized role wrappers from which physical components can be derived and an optimized object model used to unify different object forms in popular OOPLs.*
Key words: Automatic Prototype Generating (APG), Compositional pattern, Encapsulation, Inheritance, Polymorphism

A. Introduction

A significant improvement in software technology is needed to improve development productivity and software dependability. Computer-aided, rapid prototyping via specification and reusable components is a promising approach that makes this improvement possible. The computer-aided prototyping system (CAPS) is an integrated software development environment aimed at rapidly prototyping hard real-time embedded systems. PSDL has facilities for recording and enforcing timing constraints. PDSL prototypes are executable if supported by a reusable component foundation in an underlying programming language (e.g., Ada) [1,2, 3]. Since Ada95 is the only international standard programming language that supports object-oriented real-time distributed systems [4], it is reasonable for us to take Ada95 as first choice for constructing reusable componential and architectural entities used for automatic prototype generating (APG).

Computer-aided prototyping approach includes two phases: *prototyping specification* and *automatic prototype generating*. In order to obtain portability and flexibility of APG, the second phase inevitably involves how to map PSDL specification into componential and architectural object entities in different OOPLs, such as Java, C++ and Ada95, based on reusable object foundation (ROF). Ada provides full capacities of supporting object-orientation, but it lacks pristine notion of class, which makes it difficult to build a desired ROF for APG with Ada95. The pristine class constructs in Java or C++ is diversified as befuddling object forms in Ada 95, such as task unit (concurrent object), exclusive unity (protected object), instance of abstract data type (variable object) and abstract state machine (packaged object) [5, 6, 7, 8].

In object-oriented philosophy, a class is characterized by features, including attributes (representing fields of the objects of the class) and routines (representing computations on these objects), and is defined as both a structural system component -- a module (unit) -- and a type [9]. Ada95 enhances the capacities of supporting full object-orientation, but such intricate concepts introduced in Ada95 as *controlled*, *class-wide* and *tagged* types [4] could be considered befuddling. Since Ada does lack a pristine notion of class, hereby one hardly finds a perfect notion of class in Ada95 [5, 6, 7, 8]. In general, each of the encapsulation languages offers a modular construct for grouping logical-related program elements, but Ada95 provides three modular constructs: *package*, *task* and *protected* (unit). A package is the module structure that has rich functionality used for the design of ADT & ASM [5], but is not a type. Both task and protected unit are very important constructs that they can be described as a semantic component -- unlike a package, and like a class, so they actually come closer than packages to support object-oriented concepts. But both task and protected unit are unable to support inheritance.

In order to build reusable object foundation for automatic prototype generating, research has been undertaken to address three problems; (1) can a PSDL prototype be transformed into an executable system that is composed from object-oriented entities, (2) can a process for converting the object-oriented constructs to modular mechanism of Ada95 be developed, (3) what is the set of optimized properties for an intermediate object model that can be pre-compiled into specific object forms (e.g., Ada95). In papers [11, 12], the artifacts of system modeling for rapid system prototyping and its transformation via code generation were discussed. While that work established the fact that a PSDL prototype could be developed using object-orientation and that the resulting

transformation could be mapped to popular object-oriented concepts (problems 1 and 2). In papers [5, 6, 7], the artifacts for remolding object forms in Ada 95 to a unified object model established the foundation that is feasible to build a sophisticated pre-compiler (problem 3), it stopped short of addressing problem 3 on what object model should be used to unify the diversified object forms in Ada 95. This paper introduces a new object model that opens out multiple optimized properties for object-orientation, which leads to a unification of diversified object forms via a pre-compiler.

This paper is structured as follows. Section B discusses the basic PSDL constructs that are used to render a prototyping description and describes a process for creating object-oriented structures from prototyping system. As part of that process, compositional patterns are imposed so that an executable prototype can be composed from the decomposed entities. Section C describes the formulation of OOM, and compares the description in OOM with programming facilities in Ada 95 by means of case study. In this case, class modifiers are a key discriminant that directs pre-compilation from the optimized object model to different object forms and associated programming facilities in Ada95. Finally, section D provides the conclusions and directions for further research. The authors assume that the reader has some familiarity with Ada and a basic understanding of object-orientation, in particular, the concepts of inheritance, polymorphism and so on.

B. Automatic Prototype Generating

Completely automatic application generating from very high-level specification is not practical, but automatic prototype generating is feasible. Together with CAPS [2, 3], prototyping system description language (PSDL) is considered most useful for requirement analysis, feasibility studies, and the design of distributed embedded systems. Computer-aided prototyping technique is particularly effective for ensuring that the requirements accurately reflect the user's real needs, increasing reliability and reducing costly requirements changes. Computer-aided prototyping technique provides an effective solution for requirement validation. The underlying prototyping model characterizes components, their interactions, and their refinements, providing a vehicle for developing top-down decompositions. Such decompositions let large prototypes be executed with practical computation times [2, 11], compositional patterns[12, 13] that are used to enforce interactions among decomposed components, and bottom-up object-oriented construct of components.

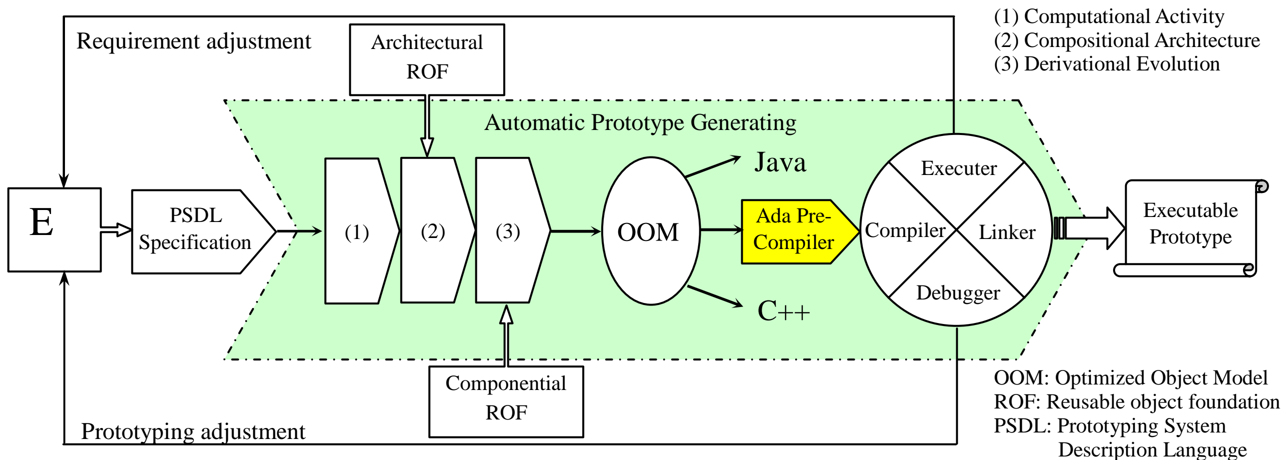


Fig. 1 Automatic Prototype Generating Process

Fig. 1 illustrates automatic prototype generating process. The transformation from PSDL prototype to an executable prototype involves three important aspects: computational activity (what functionality is needed to support customer's operation), compositional architecture (a set of rules/patterns governing the interconnections among components) and derivational implementation (components refinement and their links). Two kinds of reusable object foundation (ROF) are used to support automatic prototype generating in different phase -- architectural -- and componential ROF. They are well designed in object-oriented philosophy with the aim of their extendibility and maintainability. Under the support of well-designed ROF, an executable prototype can be automatically generated and designers only need to refine some of componential entities that are derived from both architectural and componential ROF.

APG has a key centric part: OOM and the associated pre-compiler. Ada95 provides full capacities of supporting object-orientation [4], but for the reasons mentioned in early papers [5, 6, 7], this research would like generated prototype to be unified to an optimized object model by exploiting pre-compilers, because the OOM provides significant simplification that the generated prototype can be conveniently mapped to executable program in different OOPLs, esp. in Ada95.

B.1 Computational Activity

In PSDL, the user' s needs is characterized as computational activity that is mainly concerned with what activities are needed and

how their interactions are associated with workflows, networking and plans necessary to support the operations of customers. The enhanced dataflow diagram that PSDL is based on is a directed graph with associated timing and control constraints, stated as following augmented graph

$$G = [V, E, T(v), C(v)]$$

Where $\boldsymbol{V}$ is the set of vertices, $\boldsymbol{E}$ is the set of edges, $\boldsymbol{T}(\boldsymbol{v})$ is the set of timing constraints for each vertex v, and $\boldsymbol{C}(\boldsymbol{v})$ is the set of control constraints for each vertex $\boldsymbol{v}$. Each vertex is an operator and each edge is a dataflow.

Fig. 2 shows a PSDL design of a simple control system to illustrate major language features and introduce syntax. The example is presented on two hierarchy levels: the first comprises the specification of operator *control_sys*, and the second comprises its implementation part, which contains a graph showing the decomposition of the system into two time-critical subsystems. The numbers associated with each operator symbol indicate ***met*** (maximum execution times) of that operator. Operator *control_sys* is embedded in an environment comprising a simple *switch* served by some human operator, an external *sensor* providing signals at relatively regular intervals, and an *actuator* that manipulates the behavior of the controlled technical process.

```
OPERATOR Control_Sys
    SPECIFICATION
        INPUTS input_switch : BOOLEAN, Sensor_data: REAL
        OUTPUTS control_signal : REAL
        STATES    filter_data: REAL INITIALLY 0.0
        DECRIPTION {top level of a simple embedded system}
    END
    IMPLEMENTATION
        GRAPH
            … …
        CONTROL CONSTRAINTS
            OPERATOR filter PERIOD 100 ms
            OPERATOR controller TRIGGERED BY ALL input_switch
    END
END
OPERATOR controller
    SPECIFICATION
        INPUTS sensor_data, filtered_data: REAL
        OUTPUTS control_signal : REAL
        MAXIMUM EXECUTION TIME 200 ms
        AXIOMS …
    END
END
OPERATOR filter …
END
```

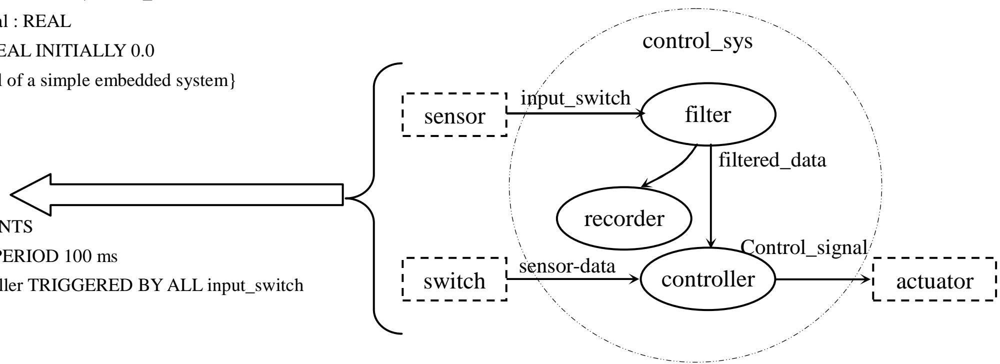


Fig. 2 Example of a PSDL Design

From the viewpoint of architecture, a software system involves coarser-grained *components* from which the system is built and *interactions* among those components, *patterns* to guide their composition, and *constraints* on these patterns [10]. In this way, vertices could be seen as components from which the system is built and edges as interactions among components, so computational activity can be defined as follows:

$$Computation_{\text{activity}} = [COM, INT, Const\ (COM, INT)]$$

Where ***COM*** is the set of components, ***INT*** is the set of interactions among components, ***Const*** (***COM***, ***INT***) is the set of constraints for both components and their interactions [11].

B.2 Compositional Architecture

Compositional architecture is associated with minimal set of rules (patterns) used to govern the interactions among components and dispense the *desired constraints* on the compositional patterns, so that computational activity can be accomplished. In order to build interactions among components, the concept of patterned compositions was presented in paper [12, 13], which is used to build software architecture that enforces the system composition from the decomposed components.

Figure 3 illustrates a compositional pattern on how to promote the interaction between components. There exist three kinds of important factors: *interactive roles* ($\boldsymbol{r_1}$, $\boldsymbol{r_2}$) components act as during the interaction, *architectural style* ($\boldsymbol{s}$) in which interaction performs, *communicative protocol* ($\boldsymbol{p}$) specified for data transportation. The roles are treated as wrappers that responsible for interaction and let components specify functionality. A compositional pattern is designed as an architectural entity known as

patterned composer, while components deigned as distributed autonomous entities.

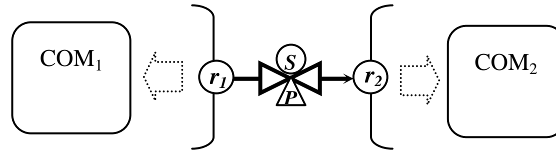


Figure 3 Role-deputizing Interaction between components

Formal compositional patterns can be represented by rigorous mathematics, which are suitable for property reasoning and automated manipulation by CASE tool. Supposing that there are three sets: ***R*** {roles that interact with each other}, ***S*** {architectural styles}, and ***P*** {communicative protocols}, e.g.,

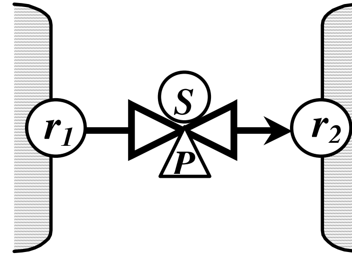
R = {		***S*** = {	***P*** = {
(Caller,	Definer),	Explicit-invocation,	Message-passing,
(Announcer,	Listener),	Implicit-invocation,	Access-memory,
(Outflow,	Inflow),	Pipe-filter,	Dataflow-stream,
(Source,	Repository),	Rep-knowledge,	Sampled-stream,
(Request,	Reply), …	Inter-distribution,	Datagram-stream,
}		}	}

So the formal definition of compositional patterns is as follows:

$$Composition_{\text{architecture}} = \boldsymbol{CP}(\boldsymbol{R}, \boldsymbol{S}, \boldsymbol{P}) = \{\text{grw}(r_i) \overset{s}{—}/_{p}\!\rightarrow \text{grw}(r_j) \mid (r_i, r_j) \in \boldsymbol{R},\ s \in \boldsymbol{S},\ p \in \boldsymbol{P},\ \text{Const}(r_i, r_j, s, p)\}$$

Where ***grw***(***r***) abstracts role ***r*** as generalized role wrapper that separates interaction (GRW provides) from computation (role component defines), —s/p→ represents interaction between wrapped components via specified style ($s \in \boldsymbol{S}$) while complying specific protocol ($p \in \boldsymbol{P}$), ***Const***(r_i, r_j, s, p) represents constraints dispensed on roles, styles, and protocols, respectively. Such constraints are abstracted as **consistency** of interactive roles, **compatibility** between architectural style and communicative protocol, **validity** of software composition, and **effectiveness** of timing constraints [12].

```
composer Pipeline is generalized
      type Data is private;
      Size : Integer : = 100;
style as   <#pipe-filter#>;
protocol as   <#dataflow-stream#>;
wrapper
   role Outflow is
   port
        procedure Output(d: Data);
        procedure Produce(d: Data) is abstract;
   computation
        Produce (d);
     *[ Output (d) → Produce (d) ∨ latency() →exception; ]
   end Outflow;
   role Inflow is
   port
        procedure Input(d: Data);
        procedure Consume(d: Data) is abstract;
   computation
         *[ Input (d)→ Consume (d) ∨ latency() →exception; ]
   end Inflow;
collaboration
     Outflow• Produce(d);
   *[ Outflow•Output(d)        → Outflow•Produce(d)
     □ Inflow•Input(d)          → Inflow•Consume (d)
     ]
end Pipeline;
```



Where,
r_1 and r_2 are Outflow and Inflow, respective,
s is architectural style : pipe-filter, and
p is communicative protocol : dataflow-stream

Fig. 4 A Formal composer for Pipeline

Fig. 4 gives the typical composer *pipeline* that exhibits excellent architectural properties (e.g., loose component coupling, asynchronous communication, possible data buffering). As an architectural entity, the composer *pipeline* is used to enforce interaction between components with dataflow. The two sides interconnected by the compoer are *Outflow* and *Inflow* roles, respectively. The *Outflow* deputizes for the producing component to output the data, while the *Inflow* deputizes for the comsuming component to inoput the data via the Pipeline.

In automatic prototype generating, interactive roles are formally abstracted as generalized role wrappers (GRW), -- a kind of generic and abstract class in object-oriented philosophy. A GRW is responsible for performing interaction and also provides the adherence to restricted, plug-compatible interfaces for composition and activity, while components separately for performing

functionality, so system components can be derived from the wrappers. In previous example, there are two parts involving CSP-based formal semantic [14], which is fit for specifying concurrent behavior.

B.3 Derivational Evolution

Derivational evolution is concretely concerned with system components and physical links that will be instantiated to carry out the computational activities via compositional patterns. When composing a system with the associated compositional facilities, patterned composers offer a means for refining or deriving physical components and can act as a bridge between computational activity (software requirement) and derivation implementation (system implementation). In this way, derivational implementation mainly involves generalized role wrappers and derivation of the associated components with redefining or overriding of restricted, plug-compatible interfaces defined by the generalized role wrappers. The formal definition for derivation implementation is defined as follows:

$$Derivation_{\text{implementation}} = [\text{CP}(\boldsymbol{R}, \boldsymbol{S}, \boldsymbol{P}), \text{GRW}(\boldsymbol{R})\ \square\text{COM}, \text{Const}(\boldsymbol{R}, \boldsymbol{S}, \boldsymbol{P})]$$

Where ***CP***(***R***, ***S***, ***P***) is a set of compositional patterns that involve three kinds of factors: interactive roles ***R***, architectural styles ***S*** and communicative protocols ***P***, ***GRW***(***R***) ▯***COM*** represents that ***GRW***(***R***) is a set of generalized role wrappers and from which components can be derived, ***Const***(***R***, ***S***, ***P***) represents that constraints will be mapped on compositional patterns and naturally dispensed on their essential factors. The derivational connectivity between GRW(***R***) and COM is stated as follows:

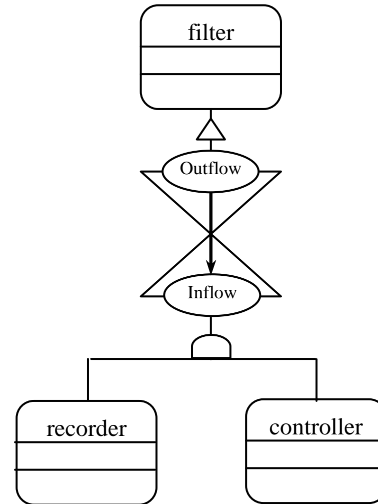
- Asscoiation: it allows system components ***COM*** to be associated with architectural properties provided by ***GRW***(***R***) and then to refine their functional computation.
- Inheritance: it allows system components ***COM*** to be derived from the correspondent GRW(***R***) and then to extend their functional computation.
- Aggregation: it allows system components ***COM*** to aggregate one more set of architectural properties provided by multiple ***GRW***(***R***) and then to refine their functional computation.

Fig. 5 illustrates how to derive components with a pipline composer (also see Fig. 2 example of a PSDL design). Component *filter* acts as ***Outflow*** role of Pipeline and both *recorder* and *controller* serve as ***Inflow*** roles.

```
composer My_Pipe is specialized Pipeline (
      Data => aType,
      Size => 100
)
component filter is associate Outflow
port
   procedure Produce(d: Data) is redefined;
computation
   Produce (d);
  *[ Outflow.Output (d) → Produce (d)
  ∨ latency  (80) →exception;
   ]
end filter;

component controller is inherited Inflow
port
   procedure Consume(d: Data) is overridden;
computation
   *[ Input (d)→ Consume (d)∨ latency(80) →exception; ]
end controller;

component recorder is inherited Inflow
     ... ...
end recorder;
```



Where, △ represents the component is associate the role wrapper, ⌓ represents the component is inherited the role wrapper.

Fig. 5 Derivational Implementation of components

With respect to behavioral computation of components, CSP-based formal semantics [14] provides not only synchronous constraints but also asynchronous control transit. Output(d) defined in Outflow role is treated as an execution guard that coordinate concurrent synchronization. For instance, the role Outflow is subjected to real-time constraints *latency* (80) referred to asynchronous control transit for hard real time constraints [2, 3]. That is, when outputting a produced data onto the given Pipeline, the component *filter* (acting as Outflow role) requires to be synchronized within latency (80), otherwise the synchronization is considered failure and an exception is raised (∨ represents asynchronous select). Similarly, the component *controller* (acting as Inflow role) requires to be synchronized within latency (80).

Fig. 6 shows the whole scenario of automatic prototype generting process. Under the support of compositinal architecture, automatic prototype generating from computational activity to derivational implementation only needs implementers to “fill holes” of components that are automatically generated as well-designed entities in object-oriented philosophy. Semantically, all

components are deigned as distributed concurrent entities that can meet the need of information systems on the unprecedented level of interopability that support the various units of a coalition.

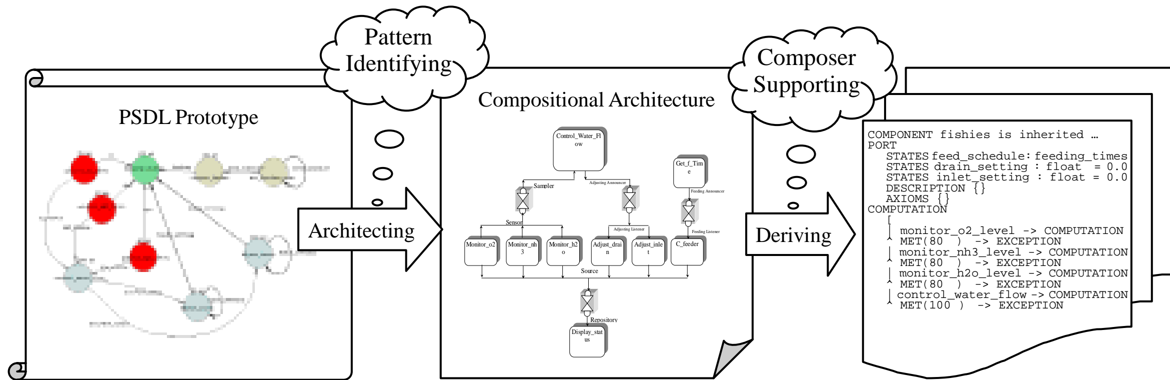


Figure 6 Prototyping Generation Process via Software Transformation

C. Optimized Object Model

In raipd prototyping generation, there exist two kinds of object-oriented entities: components and composers. The formers are functional entities that perform specific functionality, while the latters are non-functional entities that deputize for components and enforce interactions among components. Because computational activity is explicitly architected by compositional patterns that enable a compositinal architecture, concrete components are derived from the generalized role wrappers (GRW) provided by composers and their physical links are built by composers.

C.1 Class Construct in OOM

Optimized properties

Software architecture technique typically separates computation (components) from interaction (architectural facilities) in a system, which is well emboddied by components and composers in our automatic prototype generating[11, 12, 13]. Both components and composers could be mapped into popular class in typical programming languages (Java, C++ and Ada95), which is concerned with following optimized properties for object-orientation

1) Class encapsulation that allows attributes and a group of related routines to be encapsulated in a class construct, which is very essential to support object-orientation.
2) Hierarchy Inheritance that allows a subclass to inherit all features from a superclass, C++ and Java have intuitive inheritance, but Ada95 is much different from the popular syntax.
3) Abstract class declaration that allows routines to be unspecified (only interface declaration without body), e.g. abstract method (Java), abstract subprogram (Ada95), or pure virtual function (C++).
4) Polymorphism and dynamic binding that allow one interface of routine in a class to have one more method, e.g., a component acting as *Outflow* role will not care who is serving as *Inflow* role, so the routine *Consume* should be such a polymorphical interface that allows several subclasses to refin or override it with different methods.
5) Discriminat class that allows objects to have different attributes specifed during instantiation, e.g., discriminat record in Ada allows a discriminat factor to determine attrbutes of objects (data fields of the objects of the class).
6) Generic or template design entity that allows user to specify parameterized type, e.g., for generic composer Pipeline, a gneric type is needed so that the composer can accommodate different dataflow of data type.
7) Concurrent calss whose objects can be executed concurrenly, e.g., *thread* in Java, or *task* in Ada can instantiate concurrent objects.
8) Exclusive class whose routines will only be executed exclusively, e.g., *protected unit* in Ada95, or *synchronized* method in Java.

Ada is the only international standard programming language that supports object-oriented real-time distributed systems and provides full capacities of supporting object-orientation, but the diversified object forms in Ada are so intricate that people feel it difficult to find an equivalence of a class between Java (C++) and Ada95, because Ada lacks pristine notion of class. In fact, the optimized properties above mentioned are all available in Ada95, and they can be merged into a new object known as optimized object model (OOM) so that automatic prototype generating can be supported by excellent object-oriention in OOM.

Figure 7 illustrates optimized properties and evolutional scenario of OOM. Ada provides not only multiple type definitions for abstract data type (ADT) design, but also excellent unit mechanism for abstract state machine (ASM) design. In paper [5], the object form based on ADT was defined as V-Object form (object is a variable), while the object form based on ASM as U-Object

form (object is a unit). OOM unifies both U-Object and V-Object into a specific class construct charachterized by features, including attributes (representing fields of the objects of the class) and routines (representing computations on these objects).

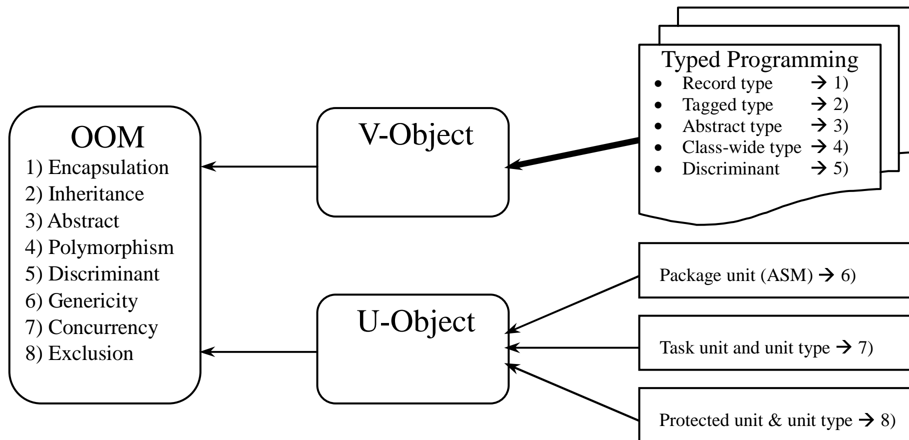


Figure 7 OOM Evolution from object-oriented properties in Ada95

Syntactic constructs

A class construct resides in a package, whose features include attributes and a collection of routine interface. Generally each routine interface corresponds to a routine that is implemented in the related package body. Task and protected typed unit can be treated as modified class, that is, special modifiers can be used to control the semantic performance for the objects of the current class. For instance, a *common* class (without modifier) has sequential objects, while a *task* class can have concurrent objects. The sketch of class construct in OOM is outlined as follows

Class construct of OOM

```
generic
   -- ... generic parameter declarations
package CName_P is
   -- ... other elements declarations
  [ task | protected | abstract] class CName [(discriminant)] is [new <superclass>]
  public
      procedure Proc1;                  -- common routine
      procedure Proc2 is interface;     -- dynamic binding routine
      procedure Proc3 is abstract;      -- abstract routine to be redefined or overridden
      procedure Proc4 is overridden;    -- overridden routine shares the same interface useful for
                                        -- supporting polymorphism and dynamic binding
   private
      attribute    : SomeType;
  end CName;
end CName_P;
```

Because semantic limitation of Ada95, *task* and *protected* classes is not allowed to have inheritance, which means no superclass can be inherited by such classes. All classes can encapsulate both attribute and routine (property 1) and but only common class (has no modifier) or abstract class (has modifier *abstract*) can support inheritance (property 2). An abstract class should include at least one abstract subprogram (property 2). If a class has an *abstract*, *interface* or *overridden* subprogram, it is supposed to support polymorphism, that is, this routine might have multiple method (property 4). The difference between an *abstract* routine and an *interface* routine lies in that abstract routine provides no method while interface routine will have method. A common class allows to have a discriminant (a parameter) to determine a selection of attributes (property 5). Genericity of a package allows a class has some attributes whose type might be unspecified (property 6). The class declaration with modifiers *task or protected* will reflect property 7 and 8.

C.2 Case Study in OOM

Generic & parameterized class

As an attractive property, genericity supported by packages can help class construct provide flexible encapsulation of features. A class enclosing the attribute whose type is unspecified with genericity is known as *generic class*. A class parameterized by means of discriminant is known as *discriminant class* that allows its objects changeable in storage. Below is a typical class of genericity and discrimnant:

Generic and Discriminant class

Features in class construct	Application
generic	**with** Stack_P;
type *tElmt* **is private;** -- undetermined type	**procedure** Main **is**
package Stack_P **is**	**package** iStack_P **is new** Stack_P(tElmt=>Integer); -- stack for integer
type tList **is array**(Integer **range<>) of** tElmt -- variant size	use iStack_P;
class Stack(*Size*: Positive) **is** **--** discriminant	
procedure Put (x: **in** tElmt)**;**	iS1 : Stack := **new** Stack(100); -- object of 100 integer elements
function Get **return** tElmt**;**	iS2 : Stack := **new** Stack(50); -- object of 50 integer elements
List : tList(1..Size);	**begin**
Top : Natural;	**... ...**
end Stack;	
end Stack_P;	**end** Main;

From the above example, a flexible class *Stack* is designed as a generic and discriminant class, which allows a user to instantiate objects of different element type, but also to produce different objects of different size.

Inheritance and Polymorphism

An inheritance can be built via hierarchy classes. Being derived from a parent class, the sub-class inherits all features enclosed in the parent, adds new features for itself (extension) or redefines some features. For instance, with a generalized class *Person*, more specialized class *Teacher* can be derived from. As a convention, all classes in OOM are descendant of the upmost class Root, which is very significant for OOM implementation (thoroughly discussed in paper [6,7]). Without specifying the parent class, any common class is supposed to be the immediate descendant of Root. An inheritance hierarchy is described as follows

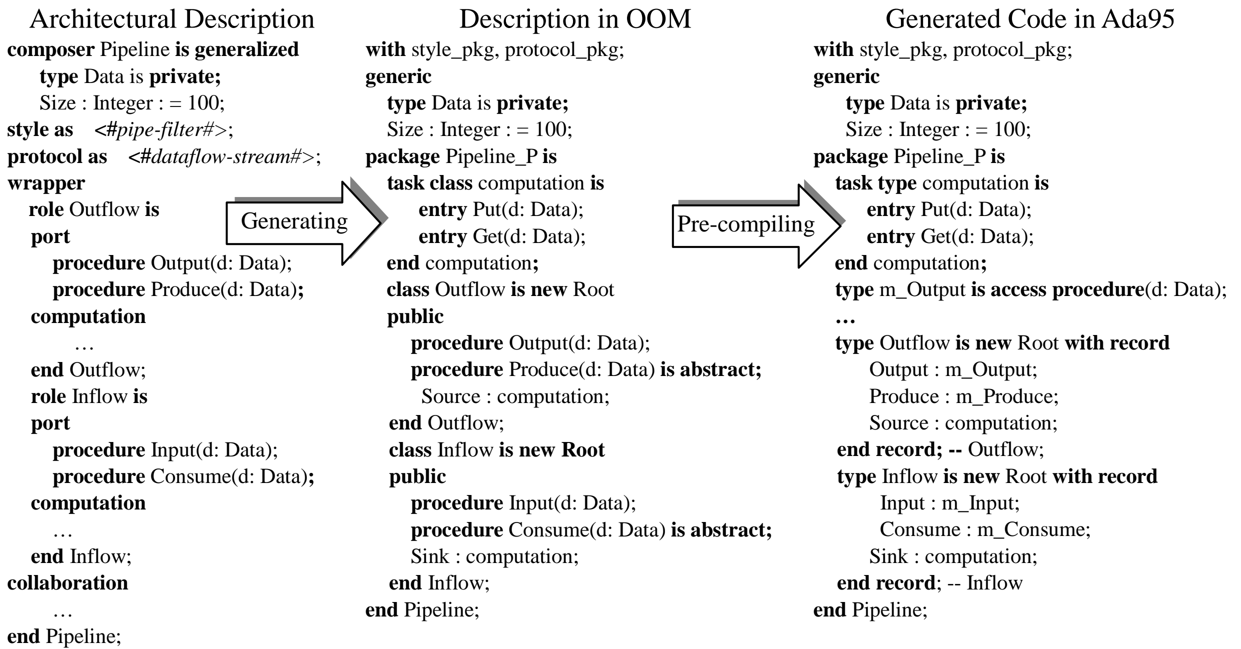
Hierarchy classes in OOM

Parent class	Derived class
with Root_P;	**with** Person_P;
package Person_P **is**	**package** Teacher_P **is**
… auxiliary declarations for class construct	**…** auxiliary declarations for class construct
class Person **is new** Root	**class** Teacher **is new** Person
procedure Display **is interface**; -- polymorphism	**procedure** Display **is overridden**;
procedure Set_Name (n:**in** tName);	**procedure** Set_School (s: **in** tSchool);
function How_Old **return** tAge;	**function** Get_School **return** tSchool;
Name : tName;	School : tSchool;
Age : tAge;	Speciality : tSpeciality;
end Person;	**end** Teacher;
end Person_P;	**end** Teacher_P;

The example states that a sub-class wants to endue some inherited features with new meanings, which means redefinition of inherited features or polymorphism. For instance, class Person has a routine Display which only displays three basic attribute (Name, Age), so class Teacher has to redefine the routine Display so as to deal with the attributes of its own, including inherited and new-added attributes. Redefinition of a routine means that a sub-class inherits the same interface (say Display), but provides a new routine version. In other words, an inherited interface may be associated with one more method, hereby which well embodies "one interface, multiple methods" – polymorphism [9].

Prototype Generating with OOM

In automatic prototype generating process, two kinds of well designed entities are involved: architectural composers and fucntional components. Components are used to perform computational activities and composers are used to built compositional architecture that enforces interactions among components. In order to support auto prototype generating, CAPS provides reusable object foundation (ROF) libraries: architectural and componential ROF, all of which are descirbed by OOM and can be mapped to typical object-oriented programming languages, such as, C++, Java and Ada95. Because of diversified object forms in Ada95, it is necessary to exploit a pre-compiler to translate the description in OOM into Ada95 code. Following examples illustrate code generation between architecture description, OOM description and Ada95 code.



Pre-Compiler for OOM

OOM provides a pristine notion of class with many optimized object-oriented properties. These are reflected in the pre-compiler as generalized ancestor, constructor, designator, inheritance, polymorphism, and dynamic binding [6, 7]. In general, a class in OOM residents in a package, so the package specification focuses on the class construct, while the package body is used to hide implementation details. The compilation from the description in OOM to destination code needs two-step processes: the pre-compiler for the front-end process and Ada95 compiler for rear-end process. Fig.8 illustrates the sketch of this idea.

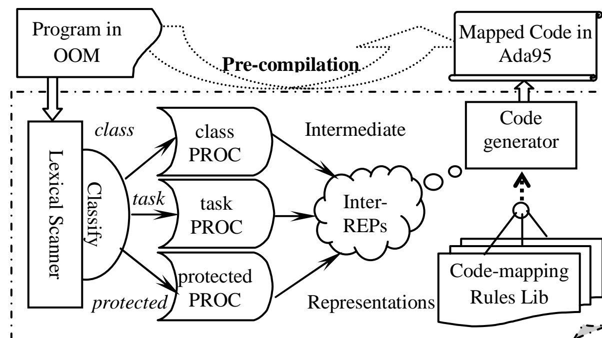


Fig. 8 Sketch of OOM Pre-compiler

D. Conclusion

There has been a growing need to be able to map well-designed entities to implementation, via reusable object foundation, for the sake of increasing software productivity. Our research addresses several key problems in automatic prototype generating with reusable object foundation. These problems include transformation of a PSDL prototype to the executable system composed from componential object entities, compositional pattern enforcing interactions among components, generalized role wrappers from which physical component can be derived and an optimized object model used to unify different object forms in popular OOPLs.

A fully formal transformational process has been shown. It is based on functional decomposition and the interconnection among the decomposed components. It also requires strict adherence to the interactions between components: roles components act as, styles by which the interaction is specified and protocols with which communications are built. The role component plays underlying specific architecture is abstracted as generalized role wrapper (many optimized properties of object-orientation are applied to), so that physical components can be derived from those reusable object foundation. This process has been demonstrated on several specific prototyping examples. We believe the process can be more automatic because many aspects such as computational activity, compositional architecture, derivational implementation, and the optimized object model are formally specified. Once this is done, CAPS can be used not only for rapid system prototyping but also for transformational development of distributed embedded software-intensive systems.

The automatic prototype generating process is based on prototyping for computational activity, compositional patterns for explicit software architecting, and object-oriented derivation. The specification of a prototype written in PSDL provides formal documentation for the system, which is hierarchically structured design with specifications of all components and the interconnections among them. PSDL is extensible because it provides facilities for adding new properties to its specification, for instance, the decomposed components should be assigned specific roles, and the edge that is used to interconnect one component with the other should be specified as suitable architecture style and communicative protocol. According to the role components play under the compositional architecture, what kind of compositional pattern is applied to enforce interaction among components is key in the prototype generating process.

E. References

[1] V. Berzins, M. Gray, and D. Naumann, Abstraction-based Software Development, Comm. ACM, May 1986

[2] Luqi, M. Ketabchi, A Computer-Aided Prototyping System, IEEE Software, June 1988

[3] Luqi, V. Berzins, R. T. Yeh, A Prototyping Language for Real-Time Software, IEEE Trans. on SE, Vol. 14 (10), Oct. 1988

[4] International Standard ISO/IEC 8652:1995(E), Ada Reference Manual, Intermetrics, Inc

[5] Liang Xianzhong and Wang Zhenyu, Ada-based Support for Abstraction, Encapsulation and Unit Hierarchy, Proceedings of Tri-Ada'91 International Conference, San Jose, USA, October 1991

[6] X. Liang, Z. Wang. Remolding Diversified Objects in Ada95: Toward A-Object Pattern. Proceedings of International Software Engineering Symposium'01, Wuhan, China, Mar 23-28, 2001

[7] X. Liang, Z. Wang. Omega: A Uniform Object Model Easy to Gain Ada's Ends, ACM AdaLetters, June, 2001

[8] X. Liang, Z. Wang. Event-based implicit invocation decentralized in Ada, ACM AdaLetters, March, 2002

[9] Bertrand Meyer, Object-Oriented Software Construction, Prentice-Hall International, Inc1997.

[10] Mary Shaw and David Garlan, Software Architecture: Perspectives on an emerging discipline, Prentice-Hall International, Inc1996.

[11] Luqi, Ying Qiao, Lin Zhang, Computational Model for High-Confidence Embedded System Development, Accepted by Monterey workshop 2002 -- Radical Innovations of Software and Systems Engineering in the Future, Venice, Italy October 7-11, 2002.

[12] Luqi, Xianzhong Liang, Transformational Model for Highly Dependable Systems, Technical report of NPS, July, 2002

[13] X. Liang. Interaction-based Compositional Evolutionary Approach for Complex Software Systems, Institute of Software, Chinese Academy of Sciences, Ph. D. dissertation, 2001 (*in Chinese*)

[14] X. Liang. Ada task specification and CSP-based formal description for Concurrent Semantics. Computer & Digital Engineering, 1989.5