Object-Oriented Support for Generic Application Frames [1]

Claudio Trotta[2] and Oscar Nierstrasz

Abstract

One step in trying to define a reuse-based software development paradigm is reasoning about the development process itself and the required information to support it. We work towards this goal by proposing a tool for designing Generic Application Frames based on the careful structuring of past experience as well as domain information. We claim that the benefits of the object-oriented paradigm have yet to be properly scaled, and that they can be achieved by applying object-oriented design techniques to describe both software components and development methods.

1 Introduction

Achieving a real improvement in software development by means of reusability is an old and elusive goal [5], [13], [43]. Recently the object-oriented paradigm has been explored as a means to accomplish this goal.The Ithaca Esprit II project [31], [32], [41] is an example of such an effort. It proposes a reuse-based paradigm of software development in which Application Engineers (AE) develop problem domain specific Generic Application Frames (GAFs) and Application Developers (AD) build running applications (Specific Application Frames or SAFs) reusing the information that is provided by the GAF [29]. The objective is to support the ever increasing changes in requirements and environment that present-day applications have to cope with, as well as to reduce the costs of application development and maintenance in selected application domains. Sample domains that have been considered as "demonstrators" in Ithaca include Public Administration and Financial Applications [18].

The Ithaca Application Development Environment consists of a collection of cooperating tools. The Software Information Base (SIB) [44], [15] is the common repository for software descriptions, and functions as the connecting element between the AE and AD tools. It contains not only structured descriptions of software components themselves, but also knowledge about the development process [37]. The SIB prototype was built using TELOS knowledge representation language [27], allowing queries to be answered using its inferencing mechanism.

1. In *Object Frameworks*, Centre Universitaire d'Informatique, University of Geneva, July 1992.

2. The author is at COPPE/UFRJ. This work was performed while visiting the Centre Universitaire d'Informatique, University of Geneva. The support of this research was provided by Ithaca and by CNPq — Brazil — 200.348/91-4(NV).

AD tools include the CooL object-oriented language, VISTA [12], [33] and RECAST [16]. VISTA is a "visual scripting" tool for interactively composing software components to create running applications. Different component sets can be supported by defining "scripting models," which characterize the compositional interfaces of the components, and drive the composition task during a VISTA session. RECAST enables the AD to specify requirements of a new application by reusing existing requirements descriptions in the SIB. RECAST and VISTA work in an integrated fashion. For a more comprehensive description of Ithaca's tools environment the reader may refer to [17].

Although some tools that assist the AD work are already running, making it possible to compose an application from existing pieces of software described in the SIB, we still lack a comprehensive understanding of the reuse method and the necessary information to support it. In this work, we try to look inside and define the contents of a GAF. The current structure of the SIB binds the scope of our proposal and provides a starting point which we build upon.

1.1 The Problem: From class inheritance to GAF/SAF relationships

One of the main difficulties in GAF design is the proper organization of a collection of software descriptions that allows a new application to be derived by taking this collection as a starting point. An ideal environment would allow one to organize a hierarchy of application frames, starting with a very generic one at the root (a GAF), passing through various degrees of genericity (as decisions are taken and missing informations are supplied) until a running application (a leaf of the hierarchy, which is also a SAF) is finally produced [29], [18].

Even if an application frame is seen as an object, the idealized GAF/SAF specialization hierarchy is not immediately achieved. The reason is that the internal relationships among parts that comprise an application frame together with its external relationships with other frames are much more complex than what can be reasonably represented by a single object-to-object specialization relationship.

Some steps have been taken in order to relate GAFs and SAFs. For example, the SIB currently supports *specificity links* [9] which are simple relationships connecting a more generic application frame to a more specific one. But an application frame is in fact a network made up of a collection of software descriptions including requirements, designs and implementations. Relating application frames in such an abstract and high level way is a rather arbitrary decision and can only give a basic hint to the application developer about similar previous experiences, but is not enough to guide a development process based on a reuse-through-past-experience paradigm. In addition, it is not clear what makes a frame more or less generic than another one.

Nevertheless, the Object-Oriented paradigm has been widely accepted as a step towards supporting reusability [24], [26], [30], [42], [46]. Object class specialization hierarchies have proven to be very powerful mechanisms for achieving reuse of classes by incremental differences, and polymorphism is the supporting mechanism to define useful generic algorithms. The problem of GAF/SAF representation could be rephrased as "how to scale the benefits of incremental transformations and genericity that can be achieved by an object-oriented class hierarchy up to the level of complex generic application frames and their set of derived specific applications".

1.2 Towards a Solution: Object-Oriented Software Descriptions

To try to solve this problem, on the one hand it is necessary to extend the benefits of inheritance not only to classes, but to every existing building block in a software description, so that it can be possible to understand and organize a set of different representations based on their small differences. On the other hand, it is necessary to provide a meaningful abstraction to the AD, which suggests the need for an abstraction that can encompass a set of classes working together in a cooperative fashion. The majority of object-oriented languages work with the class concept as the only available abstraction to conceptualize and implement systems. Some object-oriented languages and mainly object-oriented development methods have already realized this problem and have included facilities such as packages [23], subsystems [46], [8] and modules [6] in order to enlarge the granularity of the reusable component. In other words, it is necessary to understand and apply object-oriented techniques to concepts with finer as well as coarser granularities than the class concept.

The benefits and trade-offs of decomposing a single representation of a class (class packaging), as well as organizing a collection of classes (class organization) have also been identified in [19] in the context of software communities. We propose that class packaging and class organization representational issues be merged into a single comprehensive structure in the sense that the internals of a software description be exposed in order to be compared to the internals of another similar one. This shift in the software description base's organizational view allows us not only to retrieve a reusable class, but mainly to explore the similarities of previous developed software descriptions in finer as well as coarser granularity levels, making it possible to trace differences, understand project decisions, and achieve generalized representations of software descriptions in a given domain.

In the following we elaborate on the definition of a GAF design tool that is based on the principle of structuring past experience and domain information using object-oriented technology to support previously acquired knowledge. A set of class specialization lattices is used to structure past experience, allowing us to scale the size of the reusable components. Afterwards, we consider the capabilities a Domain Model should have in order to capture domain specific software descriptions, stressing mainly the problems of modelling genericity and how to perform the reuse task. A multi-level meta class frame to represent GAF models and GAF instances in the SIB is introduced, in which past experience is seen as extensional representations of specific domain knowledge. We also sketch a method to create both GAF models and instances. A GAF design process eventually produces a collection of generic and concrete reusable components in several abstraction levels and a domain-specific reuse-based development method that guides the software development process.

The paper is organized as follows: Section 2 provides the bases for understanding Generic Application Frames. Section 3 treats the problem of structuring past experience, while section 4 depicts the requirements a Domain model should have in order to capture domain analysis information. Section 5 shows how to compose and structure both past experience an domain analysis information in the SIB. Section 6 describes initial concerns about the reuse task. Section 7 provides a method to prepare GAF models and GAF instances. Finally section 8 sketches some supporting tools and section 9 presents concluding remarks.

2 Generic Application Frames

In this section we discuss what a GAF should contain in order to support a reuse-based software development process. We start from its responsibilities, which lead us to important information sources to create a GAF and to a discussion of the reuse process.

2.1 Responsibilities of a GAF

The responsibilities of a GAF have been presented and more extensively described elsewhere [29]. Roughly, they can be summarized as:

1. Provide the AD with generic *reusable information* (given a domain) that can be exploited while building new applications. This includes information about, at least, requirements, design and code.
2. Provide the AD with directions *how to reuse* the above information to develop a new application (i.e., provide the knowledge of how to build a new application from existing ones).

2.2 Information Sources for GAF Design

Mainly, there are two sources of information that contribute to structure the reusable information obtained as a result of a GAF design. They are:

- *Domain Schemas resulting from a process of Domain Analysis*

We still lack a commonly agreed upon definition of Domain Analysis. So, we will proceed to establish some definitions. A Domain Analysis process has the objective of gathering and structuring the software development information of a particular application domain, producing a Domain Schema. A Domain Schema[1] can appear in several degrees of complexity, from a simple taxonomy of the concepts of a domain up to a domain language [40]. A Domain Schema shows a collection of related generic concepts of the domain. Compared with a schema of a specific application, the domain schema depicts an overall view of the domain, includes all concepts that have ever appeared in any application within that domain, relates identical or similar concepts in the domain, shows most important concepts, and provides generic descriptions of the domain concepts. On the other hand, a specific schema described in a common software development model (normally called a *description model* in Ithaca), fixes and makes concrete a set of generic concepts, choosing between equivalent ones the more appropriate for the current application being developed.

1. We use the term Domain Schema to denote domain-specific descriptions represented using a Domain Model, differently from the definition given in [40].

A broad view of the domain analysis process entails not only representing the generic concepts of the domain, as depicted by the domain schema, but also establishing a way of (re)using the domain schema to produce an application schema, i.e., providing the necessary information to accomplish the reuse task.

- *Software Information from Past Experience.*

Although existing implementations can be seen as an input to the Domain Analysis process, they have a particular importance to GAF design because they will be the source of reusable components in the reuse environment. Also, one of our important assumptions is that the differences between already built applications will be a valuable knowledge in the process of developing a new one. Some approaches to the domain analysis problem use past experience as an input to the process in order to get an understanding of the domain and also in order to build a classification schema [39] or a domain language [14], [28]. The collected information is usually discarded after being used to develop generic domain concepts, unless some parts are agreed upon as reusable components. Any new application derived from the domain analysis product starts from this set of generic concepts represented by the Domain Schema. On the contrary, we propose to keep and organize all past experience information, based on the assumption that a new application will be much more a collection of reused components from similar past applications in the same domain than an instantiation of a very generic all-encompassing one.

2.3 Methods of the Reuse Task of a GAF

The Reuse Task is the process of reusing software components during the development process. This term was first introduced by [1].

A GAF should provide an adequate framework for both approaches to software reusability, i.e., generation-based and composition-based. In the former, the GAF should work as a frame in which missing information can be supplied by the user and the new application is generated, whereas in the latter the user will build the new application by composing pre-existing parts. Whereas purely generative approaches are successful in very narrow domains, support for automatic generation can be relative easily achieved for formal and unambiguous description models.

A mixed approach is our main objective. The user will be carefully guided in the composition process by the existing knowledge in the GAF. This knowledge will provide modelling alternatives in a given abstract level, and will also help in finding the existing corresponding components at the lower abstract level. If the desired corresponding concept (or set of concepts) cannot be found at the level below, it can ideally be either tailored by the user — from an existing generic concept — or totally/partially automatically generated using a model transformation primitive or a domain specific transformation primitive. The existing guiding knowledge in the GAF is built upon Domain Schemas and software descriptions about past experience. In the following we discuss how to structure both information sources, and how they can support both methods of reuse.

3 Structuring Past Experience

The objective we seek in this section is to organize the reusable software information in a way that makes it possible to trace existing differences among previously built applications. We assume that the Application Engineer is responsible for the task of understanding and reconciling all the past experience in a common model using a common vocabulary. The idea is not only to suggest possible reusable *object classes* but mainly to show how pieces of *software descriptions* have been selected and tailored in the past. By providing the Application Developer with the history of modeling decisions we guide and restrict the search space and the set of available development possibilities. The AD can find out that the current application that he or she is trying to build is more similar to a particular already-built application than to a very generic and abstract specification. This increases reusability by lowering the amount of software descriptions that have to be tailored or generated from generic specifications.

We approach the problem by applying object-oriented design to the reusable information itself. Several other approaches have already proposed to decompose and store software descriptions as object classes [22], [36], [37], [44] but they have not exploited the benefits of bringing inheritance to these small fragments and were also unable to identify that generic reusable frames normally arise when building blocks are factored out and grouped at that level. In the following we describe a simple and yet powerful way of representing software descriptions from previously developed applications.

3.1 GAS and SAS nodes and lattices

Past Experience is structured by comparing *Specific Applications Schemas* (SAS), which are application-specific software descriptions represented in a given description model. One can imagine a hierarchy of SASs in which similar SASs are close to one another in the tree, and whose root is a *Generic Application Schema* (GAS). It is possible to find other GASs in the tree that correspond to intermediate steps produced while developing a SAS.

To compose such a hierarchy it is necessary to have a common model to which real world concepts of specific applications can be translated in order to be compared. As the concepts are compared, a tree can be built, identifying and extracting the common ones. These concepts are normally represented by the basic elements of the model. For example, considering the ORM model [35] or the F-ORM model [11], concepts are represented by classes, roles, states, rules, messages, properties, etc. We call *software description* an instantiation of a concept or a set of instantiations of related concepts.

Each node in the GAS/SAS tree represents a network of interrelated concepts, expressing a specific or generic schema using a chosen description model. If a node represents a particular schema used in an application, it is a SAS node, otherwise it is a GAS node. The GAS node in the root would contain only the concepts common to all the specific applications developed so far. Precursor developments of applications have to be translated to this common model in order to build the initial GAS.

To develop a new SAS, not only the root GAS is needed, but the entire tree. As an example, consider the three SASs in figure 1. The squares correspond to basic concepts of the description model and the lines correspond to relationships between these concepts. Figure 1 also shows the corresponding GAS/SAS tree. GAS1 knows concepts *A, B* and their link; GAS2 knows concept *C*; SAS1 adds the link between *C* and *B*; SAS2 adds the link between *C* and *A*; finally, SAS3 adds the concept *D* and its link to *B*.

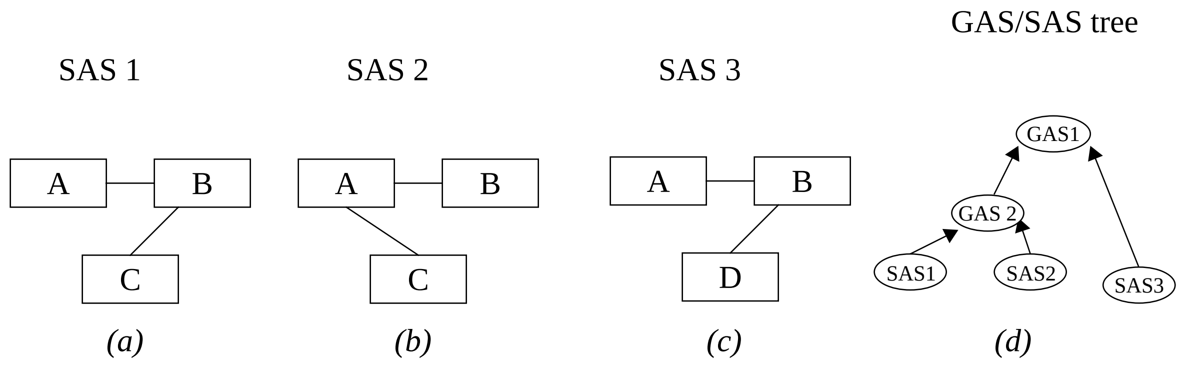

Figure 1 — Several Representations of Applications

The differences that a concept carries through the different SASs can be represented using an object generalization/specialization hierarchy. The object specialization hierarchy corresponding to the concepts appearing inside the GAS/SAS tree of figure 1-d is shown in figure 2. The names of specific SASs and GASs inside the squares stands for the context of the concept. For example, *B-gas1* is the generic *B* concept while *B-sas1* is a specialization of the generic *B-gas1*. In this case the generic *B* concept was specialized in order to refer to *C-sas1*. The arrows in figure 2 represent specializations of the concepts[1]. The arrows in the GAS/SAS node tree (figure 1-d) relate their nodes with a *meta specialization* relationship because they represent the existence of instances of specializations among their internal concepts.

It is possible to use a structure of meta classes to represent these similarities. In Appendix 1 we show a way of modelling and defining such tree using TELOS knowledge-based language [27], so that it can be stored in the SIB.

Normally, instead of a tree-like GAS/SAS structure, we have a lattice, in which a SAS can inherit pieces of software descriptions from different sources. Figure 3 depicts an example. In this case GAS1 knows concepts *A*, *B* and their link, GAS2 knows concept *C* and GAS3 knows *X* and its link to *B*. SAS1 includes the link between *B* and *C*, SAS2 includes the link between *A* and *C* and SAS3 includes *D* and its link to *B*.

1. Note that some specializations appear to enforce the type checking system. For example, it is possible to think that it is not necessary to have an *A-sas1* concept (i.e. *A-sas1* and *A-gas1* would be the same node) once in SAS1 the *A* concept does not have any additional relationship comparing to GAS1. But this would allow the *A* concept in SAS1 to be related to the generic *B* concept (*B-gas1*), and this is not true, because it should be related only to *B-sas1*.This introduces the possibility of having specialization of relationships that are represented with dotted arrows.

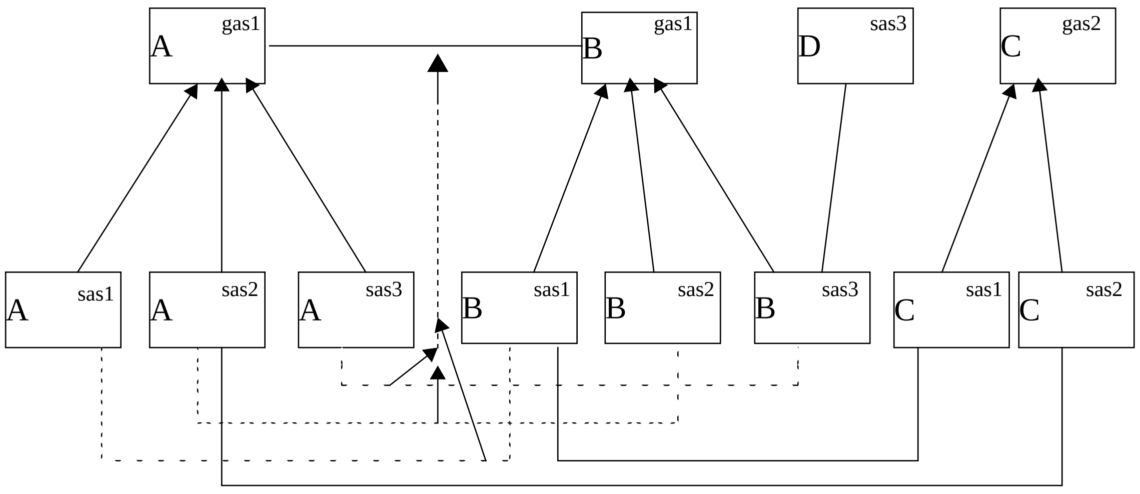

Figure 2 — The specialization hierarchy of the concepts

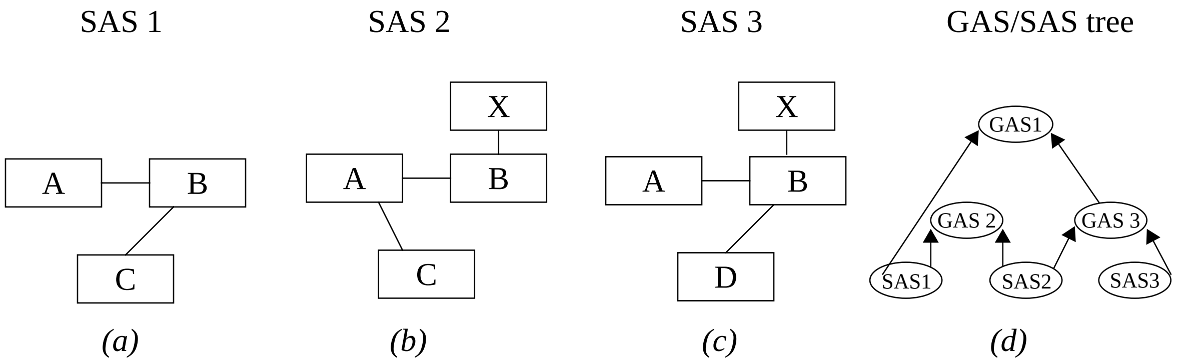

Figure 3 — Several Representations of Applications

3.2 Evolution of the Past Experience Structure

As the understanding of the new application evolves, the new SAS will find its own place in the hierarchy. This can be very useful, since it may be easier to reuse experience from another similar SAS (or an intermediate GAS) than reusing the GAS in the root. If a serious mismatch occurs between the application being built and the information provided by the hierarchy of GASs/ SASs, maybe the tree has to be reorganized, even introducing a new GAS root, but several old relationships between previous introduced SASs may remain unchanged. This approach can mix up the roles of the AD and the AE, or may require the explicit participation of the AE during the developing process.

As an example, consider building a new application SAS4 (figure 4-a) which is very similar to SAS2 (in figure 1), except for the *E* concept and its link to the *C* concept. SAS4 will find its own place in the tree, near SAS2. Figure 4-b shows the new SAS/GAS tree, while figure 5 shows the corresponding hierarchy of specialized concepts. The differences between figures 2 and 5 are shown in *italic*.

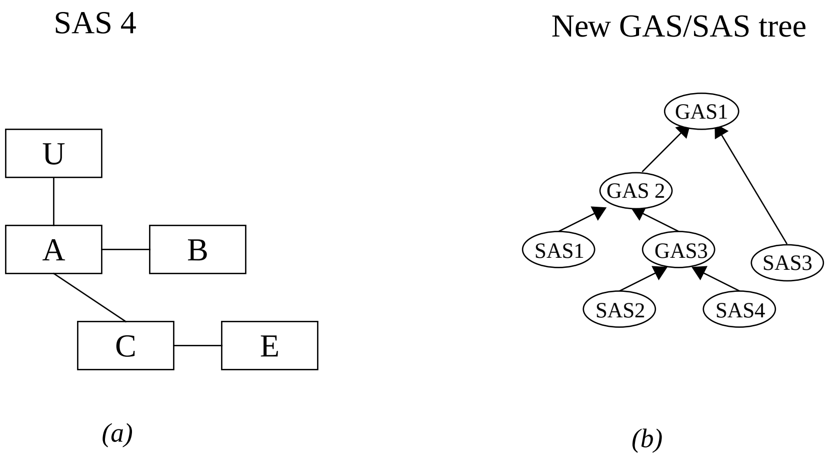

Figure 4 — Building a new application

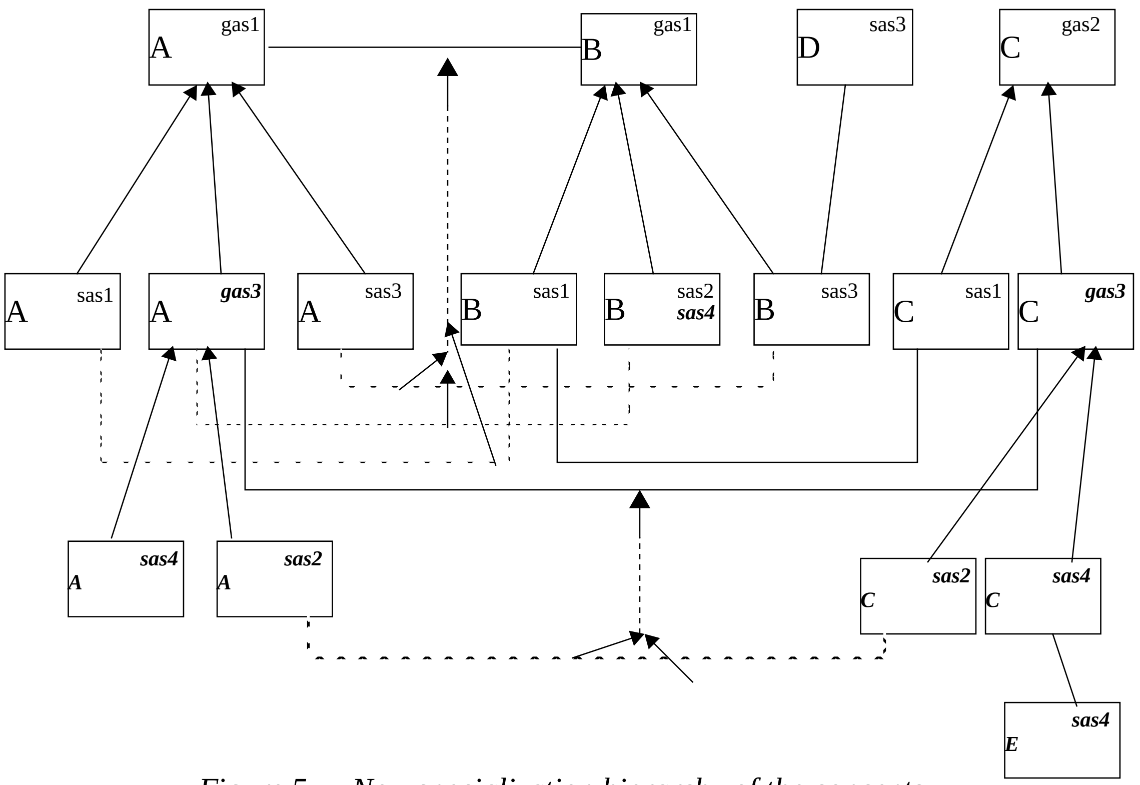

Figure 5 — New specialization hierarchy of the concepts

3.3 Abstraction Levels

A GAF is composed of several types of information, according to the model (or set of models) and the level of abstraction. For example, the part of a GAF that depicts requirements can be a collection of classes and objects (and their relationships) corresponding to concepts in the problem-space. These classes may be described only in terms of their responsibilities [45]. On the other hand, to represent implementation information a GAF may have a different collection of classes and objects (and their relationships) corresponding to concepts in the solution-space. This collection would include classes to handle user-interfaces and to cope with non functional requirements [4], and probably their descriptions would include attribute definitions and source code implementing messages. So, a GAF would have to support the representation of the domain specific information in several abstraction layers. Instead of only one hierarchy of SASs and GASs, one can consider several hierarchies depending on the model and level of abstraction of the descriptions represented in the GAS/SAS nodes. Each hierarchy can correspond to a different phase of the developing method. The interpretation assigned to the concepts (squares) and relationships (lines) among them will depend on the model and abstraction level.

It is worth noting that the development process is an iterative one, and unless the software requirement specification is complete, there is always some knowledge about the application that is introduced as one proceeds in refining an abstract concept. So, two different applications can have very similar requirements, making their SASs (representing requirements) be very near in the tree. As knowledge is added during the refinement process, their implementing code can be very different, making their SASs (representing code) be very distant in the tree. The problem is rather difficult when the level of abstraction is low, for example different SASs even from different domains can reuse the same pieces of code.

A *complete definition of an specific (generic) application* (i.e., from requirements to code) could be found in several SASs (GASs) nodes, each in a GAS/SAS tree corresponding to a specific level of abstraction. The *complete environment representing the past experience of a specific domain* is composed of a collection of GAS/SAS trees, bound by *meta refinement* relationships among nodes in different trees (figure 6). We call them “meta” because there will be instances of refinements linking several groups of concepts inside the GAS/SAS nodes.

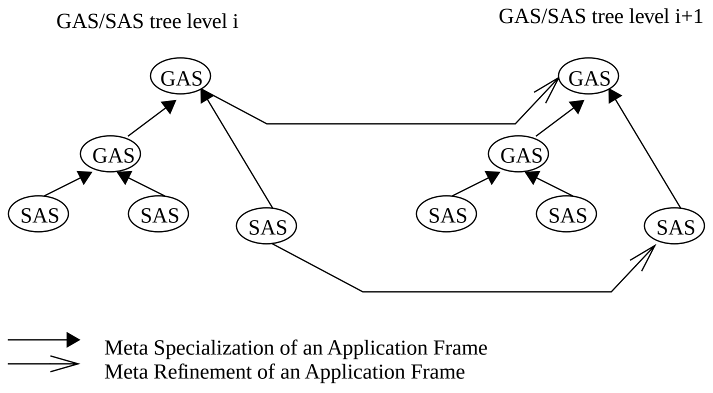
GAS/SAS tree level i
GAS/SAS tree level i+1
GAS
GAS
SAS
SAS
SAS
GAS
GAS
SAS
SAS
SAS
Meta Specialization of an Application Frame
Meta Refinement of an Application Frame

Figure 6 — A Past Experience Environment for a Specific Domain

3.4 Scaling the Reusable Component

Now let us analyse the granularity of the reusable component that is in fact represented by a GAS or a SAS node. It can vary from very fine to very coarse, depending on the semantics of the concepts that appear inside it. Consider once more figure 1, and let us assign three different interpretations to the concepts *A* and *B*, that appear linked in GAS1, using an object-oriented description model, for example one similar to Smalltalk [20]. In the first case *A* can correspond to a method and *B* to an instance variable used by the method. The semantic of the link[1] is that method *A* refers to instance variable *B*, no matter what class they each belong to. In the second case *A* represents a class and *B* an instance variable, the semantic of the link being that *B* is declared in *A*. Finally in the third case *A* and *B* are both classes, the semantic of the link being that *A* is a subclass of *B*. We have ranged from a finer granularity to a coarser one, the second case corresponding to the normal object-oriented reusable component size, i.e., a class, because GAS1 knows about the structure of a specific class, being completely equivalent to an abstract class *A* which has only one instance variable *B*. Sometimes in the design of an object oriented system we face the problem of to whom to assign a responsibility [46]. The first case copes with this problem at the design level because it describes a method (and its relationship with a variable) without knowing at that time to which class we are going to assign it. In the third case, we reuse a coarser frame, an abstract declaration that states that in all applications below GAS1 in the GAS/SAS hierarchy, class *A* has appeared as a subclass of *B*, no matter what redefinition has been done inside the classes. Specific differences in previous projects can be traced following down the tree. For example, if we assign an interpretation that *C* is an instance variable, we are stating that in SAS1 *C* has been declared in the superclass *B*, while in SAS2 it has been declared in the subclass *A*. Finally, the size of the reusable component is enhanced with the corresponding set of concepts that can be reached by following meta refinement links (see figure 6).

The size of a reusable component depends on what you put inside a GAS node. A GAS/SAS tree depicts how a generic reusable component has been "filled up" in the past, by following meta specialization links, which are internally supported by an object-oriented specialization hierarchy of atomic concepts. For example, supposing the third interpretation, GAS2 is a framework that has the knowledge that *A* is a subclass of *B*, and also, that there exists an instance variable *C* in the domain, which has not yet been assigned to any class. GAS2 also shows, following meta specialization links, that the instance variable *C* has been assigned in the past to either class *A* or class *B*. Attached textual explanations and other parts of SAS1 and SAS2 can justify why these decisions have been taken in the past.

3.5 Other issues in Past Experience Structuring

It may be advisable to combine several GAS nodes (or specific parts in the internals of a GAS node) into a single *fragment*, in order to increase the size of a reusable component. Another reason why it should be useful to create fragments is due to the way GAS nodes are structured. A GAS node is created by factoring out common concepts in past applications without regarding the semantics of the individual concepts. On the other hand, fragments can take into account the meaning and importance of a concept in relationship to the description model objective. For example, consider the case of a lattice (see figure 3). To obtain the same view of GAS2 of figure 1-d, it is necessary to form the union of GAS1 and GAS2. Maybe it is worth considering this fragment, depending on the interpretation that is assigned to concepts. For example if *A*, *B* and

1. In fact in our model, links with semantics are depicted as boxes too. We've made such a simplification to make the example smaller and easier to follow.

C are classes and *X* and *D* are instance variables, the union of GAS1 and GAS2 gives an idea of the overall set of classes that have ever appeared in the domain. On the other hand, if the AD wants to see all the instance variables of the domain (a kind of bottom-up approach) it would be useful to consider the union of GAS3 and SAS3. A systematic approach can be used to discover potential interesting fragments by applying graph traversal algorithms that search inside a GAS node looking for particular concepts.

Finally, we should consider what else is needed in a past experience structure in order to help in the process of building a new application. To achieve this objective, a GAS node or fragment would have to keep additionally two types of meta-knowledge:

1. Information giving guidelines to build a GAS/SAS node down in the GAS/SAS tree, i.e. useful information to reuse components at the same abstraction level, helping the AD to find a SAS's own place in the tree;
2. Information helping to refine the GAS/SAS node, i.e. in producing a GAS/SAS one level of abstraction below.

The problem of supporting past experience using GAS/SAS nodes will be revisited under the perspective of Domain Analysis. So, section 5 discusses again the idea of fragment, however considering genericity issues. We address the guidelines to build past experience in sections 6 and 7. Finally, in section 8 we sketch a way of helping the visualization of the browsing process of GAS/SAS trees.

4 Structuring Domain Information

Another important source of information for GAF design comes from Domain Schemas resulting as an output of a Domain Analysis process. This information is merged with the past experience structure in order to accomplish the objectives of a GAF. These two phases are in fact intertwined. We distinct between structuring past experience and doing domain analysis with the objective of stressing the importance of the former. In section 5 we show how both sources of information can be smoothly merged.

The first issue we must consider is what to collect in the domain analysis process. Arango [1] defines what kind of information needs to be collected during domain analysis. For composition-based reuse one must gather information about:

- specification and implementation descriptions;
- mappings between these descriptions;
- how to decompose a system specification into its composite specification descriptions.

On the other hand, domain analysis for generation-based reuse does not need information about the implementation descriptions, instead, one gathers information about:

- transformations and tactics to apply such transformations at reuse time.

We work towards providing a framework where both approaches can be supported by the reuse system. A GAF should be capable of guiding the AD in composing a new application but also could generate part of the solution, if it has been prepared to do so.

The information gathered during domain analysis is represented in a Domain Schema, using a Domain Model. In practice, several types of models can serve the purpose of capturing domain information. A Domain Model can be a model specially designed to capture domain analysis information, or a model that is traditionally used for application specification, design, or implementation, such as functional models or semantic data models. As examples of the former, we find domain meta-languages (for producing Domain Schemas that are themselves domain languages) [28], and data structured models like the first-order Model of the Domain [1].

As we may expect, there is a need to adapt application development models to capture domain information, because they have been designed to achieve a different goal. Also, we have not seen the supporting mechanisms that we really need to do domain analysis even in specially designed domain models. We state that the basic requirement for a Domain Model is the ability to represent both the domain-specific concepts and their relationships as well as the information of how to perform the reuse task in that particular domain. Although both of them will be represented as knowledge in the SIB, we use this partition to better detail the Domain Model's requirements, in the following.

4.1 Reusable Software Descriptions

We discuss the Domain Model's requirements to represent pieces of reusable software descriptions. They will be further elaborated on sections 5 and 7. The requirements include:

- *Concepts and relationships among concepts*

Obviously, the domain model must be able to represent the existing concepts and relationships in the domain. For example, if one is to produce a conceptual domain schema of the data in a domain, it is possible to borrow existing concepts of a well-known semantic data model which will work as a description model. A broad domain analysis process uses domain models at several abstraction levels.

- *Variations over Concepts and relationships among concepts*

A domain schema differs from a specific application schema in its ability to represent knowledge that is valid in the domain, but that may be even inconsistent if it is interpreted as being the knowledge representing a specific application. In other words, a domain schema has to cope with ambiguities, alternatives, and yet-to-be-defined issues that would make the schema invalid if all the rules that define a well-formed schema (taking the description model in which the schema is based upon) are applied. A domain schema should be able to support different ways of representing a real world concept, to permit reasoning about these options, and to provide generic descriptions of concepts that cannot be incorporated into a specific application without further elaboration. This way, besides the concepts and their relationships that are normally represented by the implicit semantics of the description model, a domain schema should add new relationships and concepts that will allow one to reason about the schema itself. This can be accomplished by changing in a controlled manner the semantics of the chosen description model, or by relaxing some constraints that have to be enforced when the description model is used in its usual role. Some of the adaptations that need to be incorporated in description models, include:

- *Concrete versus Generic concepts and relationships*

From what we have said above, a domain schema should support several capabilities that are not usual in a description model. *Generic Schemas* are software descriptions that depict collections of concepts that besides being recurring are not committed to any real application ever built. Generic Schemas can be at any abstraction level. It is possible to talk about generic requirements, designs or implementations. So it is important to define *Generic Description Models* to support the definition of generic schemas. A Generic Description Model is inspired by a common (which we will call concrete from now on) description model. Defining the generic description model implies creating new concepts (and relationships) that are more generic than the concepts (and relationships) of the concrete description model.

The domain analysis modelling process will apply to both the description model as well as the generic description model. The application engineer performs domain modelling by using the description model to represent concrete possible application schemas, and then steps backwards to reason about what is generic in the domain, using the generic description model as a modelling tool. We define a *Domain Schema* as a coherent union of a generic schema and a set of possible concrete schemas described intentionally using the generic description model and extensionally using the concrete one. Section 5.1 is dedicated to discuss how to model genericity.

- *Alternatives and Equivalent constructs*

It is very important to keep distinct the ideas of description model concept and real-world concept. Description model concepts are intended to represent and formalize a real world concept. Unfortunately, in almost all description models it is possible to have several different correct representations for the same real world concept. There are two reasons why it should be necessary to represent alternative possibilities of modelling a situation. The first one is a consequence of the description model being ambiguous. Existing equivalent representations caused by the description model's ambiguity should be represented in the description model itself, and not repeatedly in every particular schema. The other reason is the existence of modelling decisions that can only be taken when creating a specific application in the domain mainly because of a limited point of view which is used to capture a real-word concept. For example, the real world concept "employee" can be seen using the ORM model as a class, a role or a property that simply records the name of the employee. The problem is very similar to the schema integration problem in the database area [3]. It is necessary to have an understanding of the meaning of a real world concept in order to determine if two apparently different schema concepts — even with different names — are in fact the same real world concept with different dressings, caused by the limited (but still useful) view applied during modelling. The problem becomes more complex if we want to reuse past experience, because the same real world concept could have been represented in the past using different concepts in distinct description models.

A Domain Model should produce schemas in which real world concepts can be represented in several forms, and still provide guidelines towards the best representation within a specific application. These guidelines can show modelling alternatives and justifications about them, showing how and why they have been applied.

- *Justifications*

Collecting justifications is an important way of providing meaning to the AD. They can vary in complexity from simple character strings to annotated relationships among the concepts that appear in schemas both at the concrete description model level and at its generic one. They provide the rationale that explains why the domain schema has its current form. A *Justification Model* is created to represent useful points where a justification is expected.

- *Software-development relationships*

Simple extra relationships among concepts can convey information that are not usual in description models, but that are very useful when it is necessary to understand and keep track of the dependencies among software descriptions that are created during the software development process. This can be implemented using the SIB's Correspondence links [37] as for example, mappings among concepts that are at different abstraction levels representing a refinement or derivation, mappings among concepts that depends on other concepts, etc.

4.2 Method of accomplishing the Reuse Task

The reuse task is the way reuse is achieved during the development process. The Domain Model's requirements to represent the reuse task are the following.

- *Domain-independent development process*

The domain-independent development process is simply a software development method. Although we seek a reuse-based development paradigm, not always there will be available reusable components during development time. Overall development strategy can be reuse-based, but the method has to leave hooks to perform normal development. So, the reuse task — which is domain-dependent in our approach — should be based on building blocks that allow us to develop new software pieces either from existing ones or from scratch.

- *Composition and Generation-based approach*

The reuse task should support both composition and generation-based approaches. Both approaches have proved in the past to be useful to achieve reusability.

- *Domain-dependent Knowledge*

A reuse task is a kind of domain-specific software development method, so it incorporates a domain-specific way of carrying out the development process. The reuse task will consider domain characteristics as well as special ways of handling some instances of software components.

Summarizing, a new piece of software description will be produced from scratch or by a composing or generation process, guided by domain-specific knowledge. Sections 6 and 7 will show how to cope with these requirements.

5 Merging domain analysis information and past experience

We now show how part of a domain analysis schema can fit into the past experience structure. We work firstly on the problem of generic descriptions.

5.1 Modelling Genericity

One of the major problems for reusability is defining what is exactly a generic component or a generic software description. Several possibilities arise. ADA generic packages allow one to define a single module to represent a data abstraction, fixing the real data types only when module instances are created. Another example is frames [2]; in this case instead of fixing types, the reuser can fix pieces of code that exist as parameters described in a generic frame syntax. Object-oriented techniques go one step further because they allow flexibility by providing a hierarchy of generic modules and by not obliging one to hard code the decision of what "routine to call". Up to this point we have an understanding of what genericity means at the programming level, but it is not that clear when we talk about designs, requirements or system architectures. So we proceed by establishing Generic Description Models that will allow us to reason about genericity at several abstraction levels.

Generic models are inspired by concrete ones. Not all concepts in a description model are candidates for having a correspondent generic one. On the other hand, it is possible to have a many to many mapping between concepts in the concrete description model and the ones at the generic level.

The past experience structure, described in section three, provides us with some insight into modelling genericity. GAS nodes are the result of factoring out syntactic commonalities, but fragments can be created by considering significant ways of aggregating GASs based on the semantics of particular concepts in the description model. Before doing domain analysis, it is necessary to establish what types of fragments can be relevant. Interesting *fragment types* form *generic concept types* that are defined in the generic description model. A fragment is a domain-specific instantiation of a generic concept type. Creating the generic description model is the same as defining what can be recorded as generic about a concept.

We define two *orders* of fragment types. The notion of order is related to the way one describes a description model. A first-order fragment type is one that has a corresponding concept in the concrete description model. On the other hand, second-order fragment types are formed by considering interesting ways of relating first-order ones.

Another difference between first-order and second-order fragment types are their relationships to the past experience structure. First-order fragments are directly supported by the internal past experience specialization hierarchy of concepts, as depicted in figure 2. On the other hand, a second-order fragment type does not have a corresponding description model concept and needs to be supported by appropriate recording of interesting GAS/SAS (or pieces of GAS/SAS) aggregations.

One way of finding out fragment types of a generic description model is to ask what useful information and behaviour is needed to represent genericity. A list of the description model's concepts can be checked out in order to provide insight about useful first-order generic concepts. For example, taking a Smalltalk-like object-oriented description model, we could model the following concepts: Class, Instance Methods, Instance Variables, Object and their companion relationships: Specialization, Class-has-method, Class-has-Variable, Method-uses-Variable, Method-calls-Method, and Instantiation. We restrict our discussion to the first three concepts and their relationships. We also avoid discussing Meta-Classes, Class Variables and Class Methods.

Now, we can find first-order fragments, asking the following questions.

What is a Generic Class?

A Generic Class is a class that cannot be used to instantiate objects, but normally serves as a template for behaviour that will be fully defined in subclasses. A Generic class can only provide a frame, by defining which data and methods are expected, or even provide a full implementation of a behaviour that will work in a concrete subclass thanks to polymorphism. Sometimes they are called abstract classes [46]. We avoid calling them "abstract" because it is possible to have generic classes at different abstraction levels. In our proposal a fragment representing a generic class encompasses all the information represented by the collection of GAS/SAS nodes that know a given class. In the past experience structure we show how a generic class has been used previously by following meta specialization links, finding out which instance variables, methods and specialization links have been added by specific applications, passing though several levels of genericity. If in the top GAS node where the class first appears it is already linked to a method, instance variable or specialization link, that means that during the domain analysis process these characteristics have been identified as being generic in that domain.

Answering the above question does not add new enhancements to the way people have been using generic classes so far, except for the fact that our representation makes it possible to include also the set of subclasses and super classes of a given class as generic information about it. More interesting results can be found by answering the following questions.

What is a Generic Method?

Normally, we understand a Generic Method as a method that appears in a Generic Class intended to be implemented by a subclass. Two possibilities exist. In the first one only the method's signature and its intended behaviour matters. In the second one an implementation is also given, which may possible be re-implemented by a subclass below in the hierarchy. In this view, a generic method is always related to a generic class. Our proposal is more flexibility in the sense that it allows class-independent generic methods. A Generic Method is a fragment that encompasses the knowledge represented by the collection of all GAS nodes that know a given method. If a method appears linked to an instance variable in the top GAS where it first appears, what is generic about the method is that it always uses that instance variable; if it first appears linked to a class, this fact is also generic about the method. As we follow meta specialization links we can find out other variables and classes that the method has been assigned to in the past. If we even define and structure the internals of an implementation of a method it is possible to show small differences between redefined methods. With this kind of generic method it is possible to reuse the knowledge that an intended behaviour is needed in the system, without having to take (a sometimes early) decision as to which class to assign the behaviour. Past experience structure shows and justifies different past decisions about an intended behaviour, even if the behaviour was assigned to unrelated classes. This kind of knowledge is not represented in a classical object oriented specialization hierarchy, unless the behaviour is assigned to one of the classes in the same specialization hierarchy.

What is a Generic Instance Variable?

The same ideas described for Generic Methods apply to Generic Instance Variables. They represent knowledge that is needed to be recorded about the system, without having been committed to specific classes.

These are considered first order fragments because they are drawn from concrete concepts Class, Instance Method and Instance Variable respectively. Also, as said before, first-order fragment types can be directly supported by the internal past experience hierarchies. For example, taking figure 1 and considering that concept *A* is a class, we can find the *A generic class concept* in figure 2, depicted by the part of the hierarchy which is headed by the square named *A-gas1*. On the other hand, if we take only the representation in figure 1-d, it would be necessary to form the union of parts of GAS1 and SAS2 is to represent genericity and specificity about class *A*. The reason why this is so is because the goal of GAS/SAS trees is to represent aggregations of pieces of generic concepts, each GAS (SAS) node depicting normally a set of generic (concrete) related concepts.

Now we can talk about other interesting generic information to find out second-order generic concepts. Guidelines to discover second order fragments are to reason about what could be useful generic information extracted from relationships among concepts and meaningful sets of first-order fragments. We show respectively examples of both by answering the following questions.

What is a Generic Class Hierarchy?

A Generic Class Hierarchy is a skeleton of common specializations relationships among classes. A fragment that knows all GASs that include specialization relationships represents a Generic class hierarchy. Note that a Generic Class Hierarchy is not a generic schema itself, but only a recurring pattern of classes that relate to each other by a means of a specialization conceptual abstraction. Figure 11 shows an example.

What is a Generic Client or Server?

A Subsystem is a set of classes that work in cooperation to accomplish a set of tasks. A subsystem supports a protocol, i.e., a set of behaviours that a client of the subsystem can rely upon. A protocol is a kind of generic concept. Ideally subsystems could be changed without requiring modifications in their clients (for example, changing the windowing system of an application should not cause modifications in the application itself). This question assumes design for reusability as before, but the difference is that a protocol is not as obvious a concept as are those of class, instance variable and method. A protocol is a set of related behaviours that accomplish an objective, providing a clear interface for using a subsystem. A fragment that aggregates a set of Generic Methods that work to accomplish a coherent behaviour is a Generic Server, following a protocol. The set of Generic Methods that uses the protocol (totally or partially) is a Generic Client. From this abstraction it is possible to follow meta specialization links and discover how the same protocol has been implemented in different subsystems. A new subsystem can be constructed from the Generic Server, allowing even a completely different distribution of the methods among the classes.

An idea that is yet to be developed is to represent a subsystem as a class (the protocol of the subsystem being represented by the set of behaviours of the class) and then applying again the process of looking for genericity in the collection of classes that represents subsystems. We could create a kind of second-order GAS/SAS tree, which would allow us to trace differences among subsystems, acquiring the notion of the overall subsystem generic objectives and also showing differences between similar specific protocols.

Finally, we could ask what comprises a Generic Application. We prefer to use the term Generic Schema, to avoid confusing with Generic Application Frame, which will encompass besides the generic schema, past experience structure, alternatives, justifications and a model of the reuse task.

What is a Generic Schema?

A Generic Schema in an object-oriented description model is the set of generic classes, methods, instance variables, class hierarchies, clients and servers. This is represented by a GAS/SAS tree or lattice and the collection of fragments defined above.

With this bunch of generic descriptions available in an application domain, object-oriented design is a matter of deciding which encapsulation scheme better represents the requirements of a specific application. Design guidelines, such as the Law of Demeter [25], can be enforced if the model is enhanced by supplying instance variable types, method parameters types, and method implementation code. These guidelines will appear represented by model-dependent adaptation design operations (see section 6).

5.2 An Example with Generic Descriptions and Past Experience

We illustrate the points discussed so far using the following example. Take figure 7 as two SASs using an object-oriented description model with the following concepts: Class, (instance) Variable, (instance) Method and the following relationships between concepts: Class-has-variable, Class-has-Method, Method-uses-Variable, and Class-isa-Class (specialization).

Figure 8 shows the corresponding GAS/SAS lattice. Concepts that appear without a box in SASs are concepts that have been initially defined in a parent GAS. A GAS node is internally

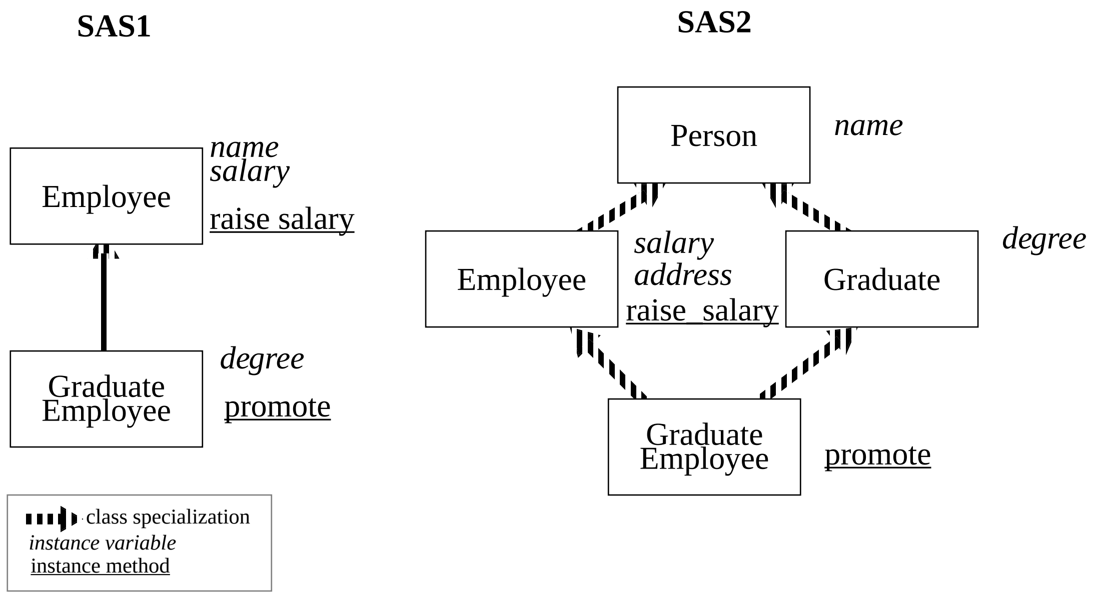

Figure 7— An Example — Schemas in a Smalltalk-like description model

represented by an aggregate of *conceptGAS1* classes as depicted in figures 9 and 10, which will be explained below. We now are able to illustrate the following generic concepts.

Generic Class *Employee* is represented by the internal class *EmployeeGAS1* in figure 9, which depicts common relationships with generic method *Raise_Salary* (represented by *RaiseSalarGAS1*) and generic variable *Salary* (represented by *SalaryGAS1*). Two specializations shows how *Employee* has been adapted in applications SAS1 and SAS2, represented respectively by *EmployeeSAS1* and *EmployeeSAS2*.

Analogous reasoning leads us to ***Generic Variable*** *Degree*, which is represented by *DegreeGAS1* in figure 10. From GAS1 we can see that the variable *Degree* is linked to a method that uses it, namely method *Promote*, but it is generic because it is not yet bounded to any class. Method *Promote* is also generic, but it has been assigned in GAS1 to class *Graduate_Emp*.

As we said in section 4, we can have two types of knowledge concerning genericity. The intensional knowledge will be supported by Design Operations (discussed in section 6) that know that to get a concrete application, *Degree* must be visible to *Graduate_Emp*. So, Design Operations will provide guidelines how to go from the generic description model to the concrete one (in this case, stating that it is necessary to find a relationship of the type Class-has-Variable between *Degree* and *Graduate_Emp* or any one of its parents). Extensional knowledge is supported by the past experience structure that shows two possible solutions adopted in the past, namely defining *Degree* in the *Graduate_Emp* class itself or defining it in class *Graduate*, a superclass of *Graduate_Emp*. Any other extensional knowledge that the AE wants to record from his understanding about the domain can be represented as a hypothetical past built application.

A ***Generic Class Hierarchy*** is supported by the relationship Class-isa-Class between *Employee* and *Graduate_Emp* in GAS1. Other specializations relationships (application specific class hierarchy) have been defined in SAS2.

A ***Generic Server*** can be defined by GAS1, as the one that answers to the protocol comprised of Generic Methods *Raise_Salary* and *Promote*. In this case, the set of methods are in the same GAS, but this is not necessarily true for every useful generic protocol.

Finally, a ***Generic Schema*** is comprised of the generic examples described above. Note the size of the GAS1 generic reusable component. It includes a Generic Class Hierarchy, two Generic Classes, two generic variables and two generic methods. Some basic generic relationships are supported by it.

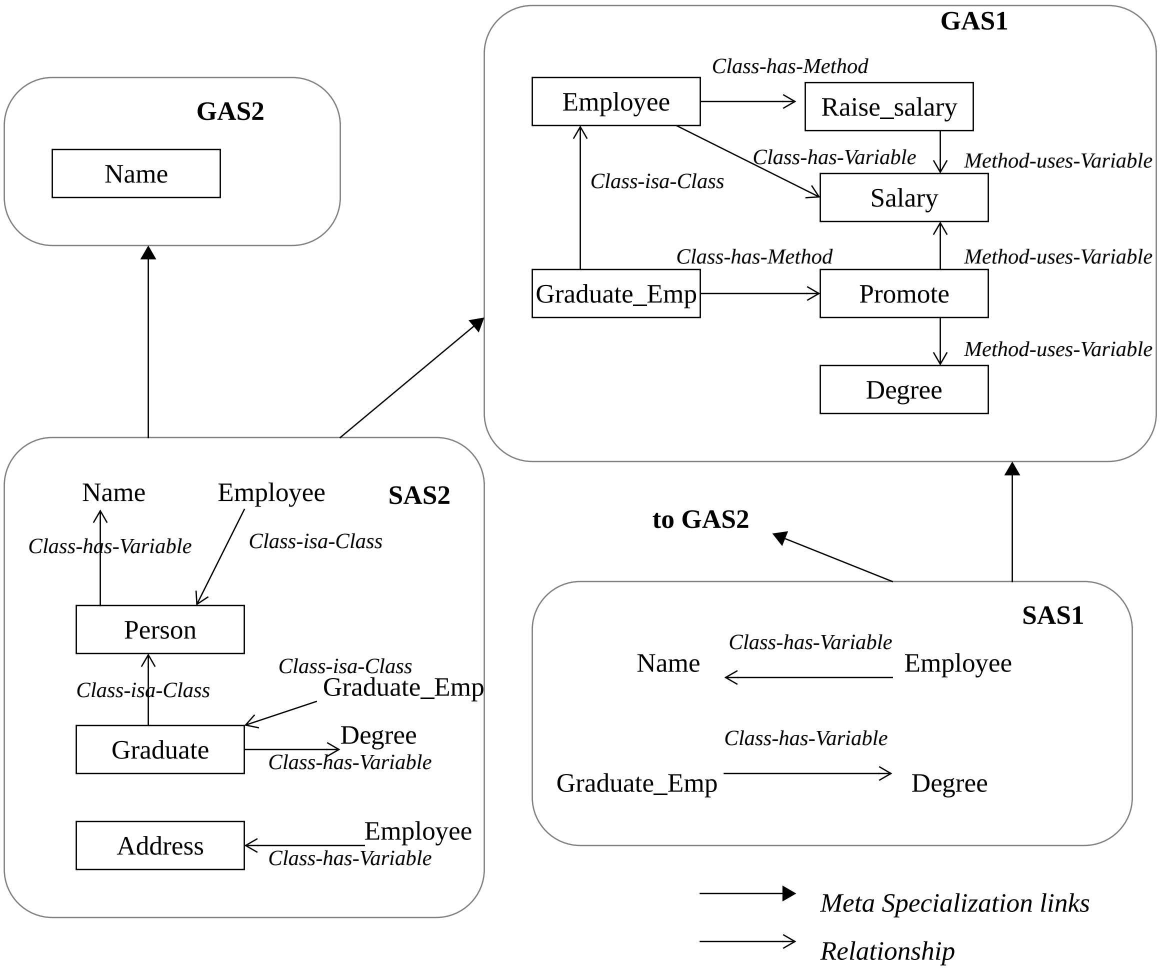

Figure 8— An Example — GAS/SAS lattice

5.3 Generic Concepts and Generic Relationships

The generic description model may need to create new relationships among generic concepts that are not useful in the concrete description model. The AE should create the generic description model by carefully examining the meaning of the concepts in the concrete description model. The understanding of the interaction among these two models is also useful to describe the rules that allow one to adapt a generic concept to a specific one, later on.

The new relationships among generic concepts may need to be represented implicitly in the generic description model, or explicitly in a particular, domain-specific generic schema (i.e, in an instance of the generic description model). As an example of the former, consider that it is

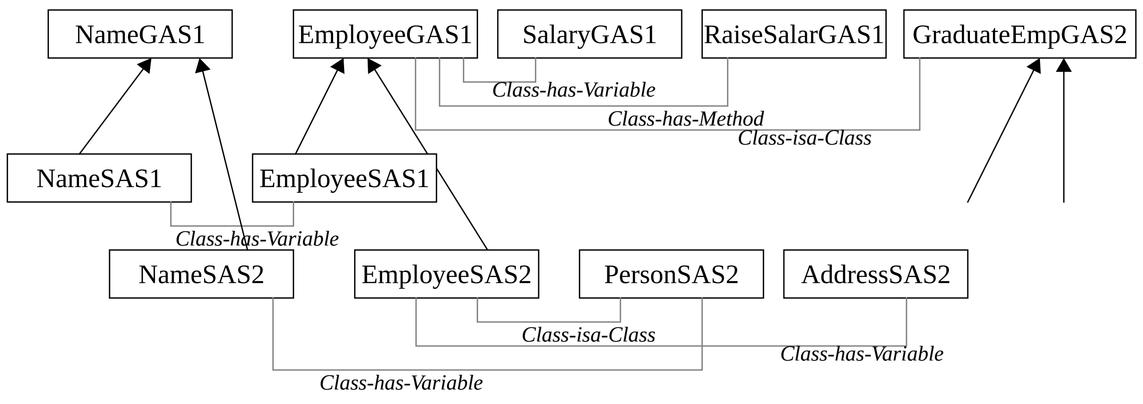

Figure 9— An Example — Generic Class

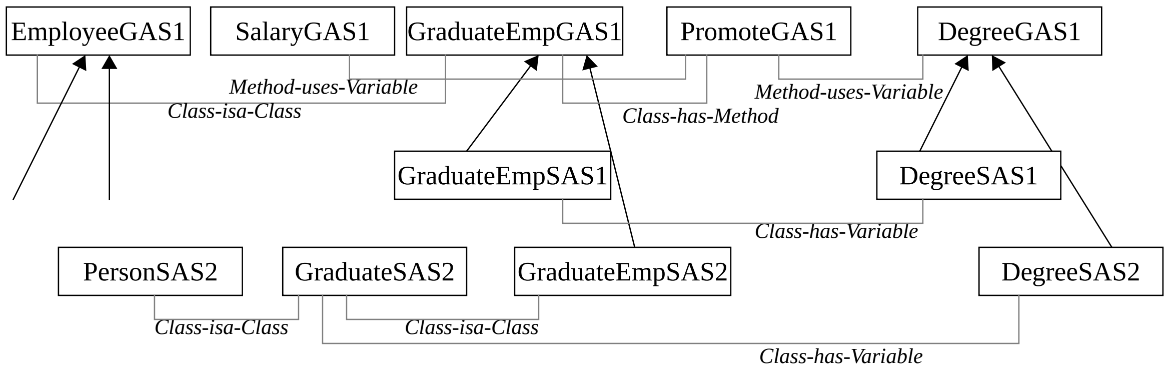

Figure 10— An Example — Generic Variable

necessary to represent that variable *Degree* has to be visible to *Graduate_Emp* in any SAS ever built taking GAS1 as a starting point, because both relationships hold in GAS1: *Graduate_Emp* Class-has-Method *Promote* and *Promote* Method-has-Variable *Degree*. This knowledge is represented at the model level, because it is an assertion about any schema conforming to our Smalltalk-like description model. In other words, it says that if an instance method refers to an instance variable, the instance variable has to be declared in the same class in which the instance method is defined or in a superclass of it. In fact, SAS1 and SAS2 obey this rule.

On the other hand, domain-specific knowledge may need to be represented by adding a relationship in a particular generic schema. Note that class *Employee* always has access to variable *Name* in all SAS ever built, and suppose that this fact has been recognized as being domain-wide during domain analysis. We could merge GAS1 and GAS2 into a single GAS node, adding a generic relationship of the type *Class-knows-Variable* between *Name* and *Employee*, which has the meaning that *Name* is an attribute of *Employee*, defined in it or inherited from one of its super classes. The difference between the two examples is that in the former, the relationship *Graduate_Emp* Class-knows-Variable *Degree* can be derived from the semantics of the model, while in the latter, it is a domain-specific, not derivable, piece of information. Another example of a generic relationship caused by inheritance is *Class-knows-Method*.

Another concern about generic concepts and relationships is to see how a generic relationship can be transformed into a generic concept, also affecting the concrete description model. For example, if it is important to model and represent Generic Hierarchies, we could promote the *Class-isa-Class* concrete relationship to a concrete entity in order to allow to model even seemly incompatible views. If we want to represent the generic concept that classes A, B and C (figure 11-a) are always related by specialization relationships, we could have the Generic Hierarchy G as an entity, forming a GAS/SAS tree as the one depicted in figure 11-b. This example shows how defining the generic model can effect the definition of the concrete one. Of course, the order of a fragment is directly related to the way we describe a concrete description model. In our example, Generic Hierarchy has been promoted to a first-order fragment type.

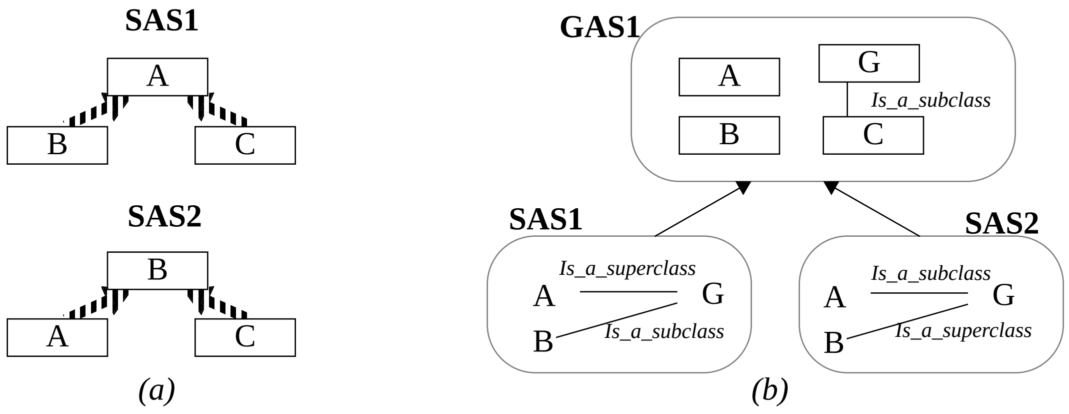

Figure 11 — Relationships as Concepts

5.4 Modelling Alternatives

We now address the problem of representing alternative constructs and see how it relates to the generic model and past experience structure.

The Application Engineer is responsible for forming a domain vocabulary which is composed of real world domain concepts. This dictionary has simple entries and can constitute a domain taxonomy as proposed in [39]. In our case, the dictionary is comprised of *informal* textual descriptions that make a *user-understandable* domain vocabulary, including synonyms and homonyms. A *detailed* and *formal* description of the domain concepts is represented by a fragment that aggregates other concrete and generic ones, showing modelling alternatives. For example, the domain concept “employee” is informally represented in the dictionary by a simple description, together with other relationships among other domain concepts forming a dictionary cross reference. To get a formal understanding of what “employee” means and how it has been or could be represented in several applications, we find an employee fragment that shows alternatives (exclusive or not) of representing it in one or more description models. This fragment depicts several other fragments that represent “employee” as, for example, a generic class, role, or property (taking the ORM as an description model).

The generic description and the domain vocabulary are built in fact at the same time as the domain analysis process proceeds. The Past Experience structure is a starting point to build a Domain Schema. After an initial GAS/SAS tree is built, the AE needs to go back to the real world to answer three main questions concerning generic concepts.

- With respect to the domain's objectives, what caused the differences among past applications that were detected in the process of structuring past experience?
- Is it possible to define a set of generic fragments by abstracting sets of related differences according to the generic description model?
- What others generic fragments can we preview based on the acquired knowledge about the domain (i.e., generic fragments for which there is no past experience supporting evidence)?

An understanding of the real-world domain concepts is done by answering the following questions.

- What generic or concrete fragments represent in fact different modelling views of the same real world concept?
- How do they relate to each other? Are there exclusive modelling alternatives?
- What justifies these past experience decisions? What has to be investigated in the real world to allow taking a decision at the time that a new application is being built?

Of course, these questions need not be answered in this order, but are in fact modelling domain analysis guidelines.

Answering the last three questions will allow us to restructure the past experience structure based on modelling alternatives. The AE will define a unique way of representing each real world concept in the domain. SASs will be redesigned transforming each concept in the corresponding chosen representation. This creates for each SAS what we call a corresponding *shadow SAS*, which is semantically equivalent to the original one. Shadow SASs will create a more coherent past experience structure because real world past experience concepts will appear depicted in an unified representation, increasing the number of commonalities that can be extracted. We still maintain links between original and shadow SASs, keeping track of the reasons that caused the differences between the representations.

In the following we show how to model the reusable information taking the framework provided by the SIB as a starting point.

5.5 A Structure to hold the Reusable Information

Currently, there is a standard way of describing a description model to the SIB [37]. Each concept in a description model has to be classified as an entity, connection or construct. The method takes advantage of the TELOS knowledge-based language [27] facilities to provide a hierarchy of meta classes that allows one to reason about description models. Figure 12 shows a simplified representation of the structure for meta classes.

Meta classes of the third level (M3) are named Entity, Connection, and Construct, that aggregate the basic concepts of the description models. In the second level we see their instances, for example, we have the class *ORM_Entity* that will be, in its turn, the meta class of all concepts that are considered to be entities in the ORM description model, while *ER_Entity* will do the same job for the Entity-Relationship data description model. In the first level of meta classes we see instances of the model's concepts. *ORM_Class* and *ORM_Message* are two M1 classes of *ORM_Entity*, the second one allowing us to model behaviour.

The simple class level keeps domain-specific concepts, corresponding to concepts of a particular application schema. For example, the *Employee* class represents the fact that there is some ORM Schema that has modelled the real-world concept "employee" as an ORM class. Finally, there will be another simple class that will gather all the concepts of a particular application schema, for example, *Payroll*. Dynamics aspects of a model, as for example behaviours and events, have to be represented as instances of the M3 Entity class, although the AE is free to represent any differences (for example dynamic versus static aspects) at the M2 level.

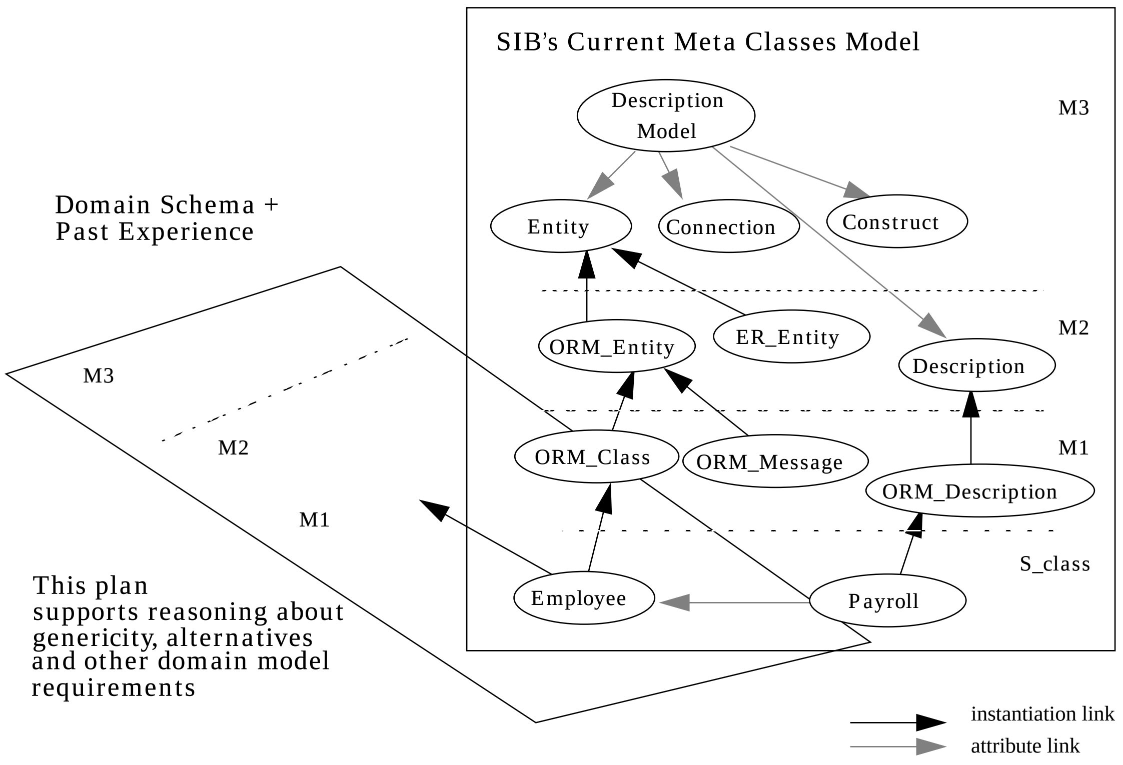

Figure 12 — Software Descriptions Meta Classes

The current SIB model supports reasoning about a given description model by representing constraints and relationships among meta classes at the M2 and M1 levels. They are useful to describe an application specific schema, but are not enough to model domain schemas embedded in past experience. If there are two or more already developed *Payroll* ORM Schemas, they will be related by simple SIB links (see [37]).We further need to represent generic schemas, past experience, modelling alternatives, domain taxonomy, and the domain-specific reuse task.

We need another plan in which we can describe possible associations for reusability. In this plan, also depicted in figure 12, we have the following parallel structure:

- *Simple class* level is populated by concrete classes, like *Employee*, corresponding to the most atomic correlation between both plans. This level also includes generalizations of concepts that form nodes in a GAS. These generalizations will not be instances of the M1 classes of the description model plan, but instead, will be instances of generic concepts. The requirements are different because new relationships can be introduced and constraints can be relaxed. For example the relationship Class-knows-Method (example in figure 13) exists in this generic plan, but does not exist in the concrete description model plan. Examples of constraints that can be relaxed are several: a generic object-oriented class is not obliged to be attached to any place in the class specialization hierarchy; a generic relational table is not obliged to have a primary key; a generic structured analyses process is not obliged to have an incoming (or outgoing) data flow, etc.
- *M1 class* level will describe generic description models, defining generic fragments types (first-order and second-order), guiding what instances of relationships and constraints can exist to model the generic instances described above. In this level we can represent alternative modelling, generic relationships types, and useful ways of aggregating sets of fragments in a particular description model. First-order generic types are candidates to be specializations of M1 classes existing in the description model plan.
- *M2 class* level will factor out the requirements of a domain-model, showing what should be common to any one of them, no matter what description model they were inspired from.
- One *M3 class* will gather the generic components in a *Generic Description Model* class, which is a subclass of the M3 class *Description Model*. The former class specializes the latter by mainly restricting the types of *Entity*, *Connection* and *Construct* classes to proper subclasses *Generic_Entity*, *Generic_Connection* and *Generic_Construct* and adding second-order generic concepts.

Figure 13 shows an example. Labelled relationships should be in fact modelled as instances of Connection type classes. We've made this simplification to avoid cluttering the diagrams.

Finally, representing justifications and software-development relationships as special objects seems to be enough to complete the representational requirements of a domain model. Abstraction levels are represented by a set of domain schemas, having fragments related by generic or concrete refinement links. Going through these issues will provide a thoughtful understanding of the application domain's concepts and their relationships, supported by real past experience in other developed applications in the same domain.The next step is to structure the knowledge of how to reuse all these available information.

6 Supporting the Reuse Task

A GAF is a reuse infrastructure comprised of a combination of a domain-specific schema and structured information of previous development *plus* a domain-specific model of the reuse task.

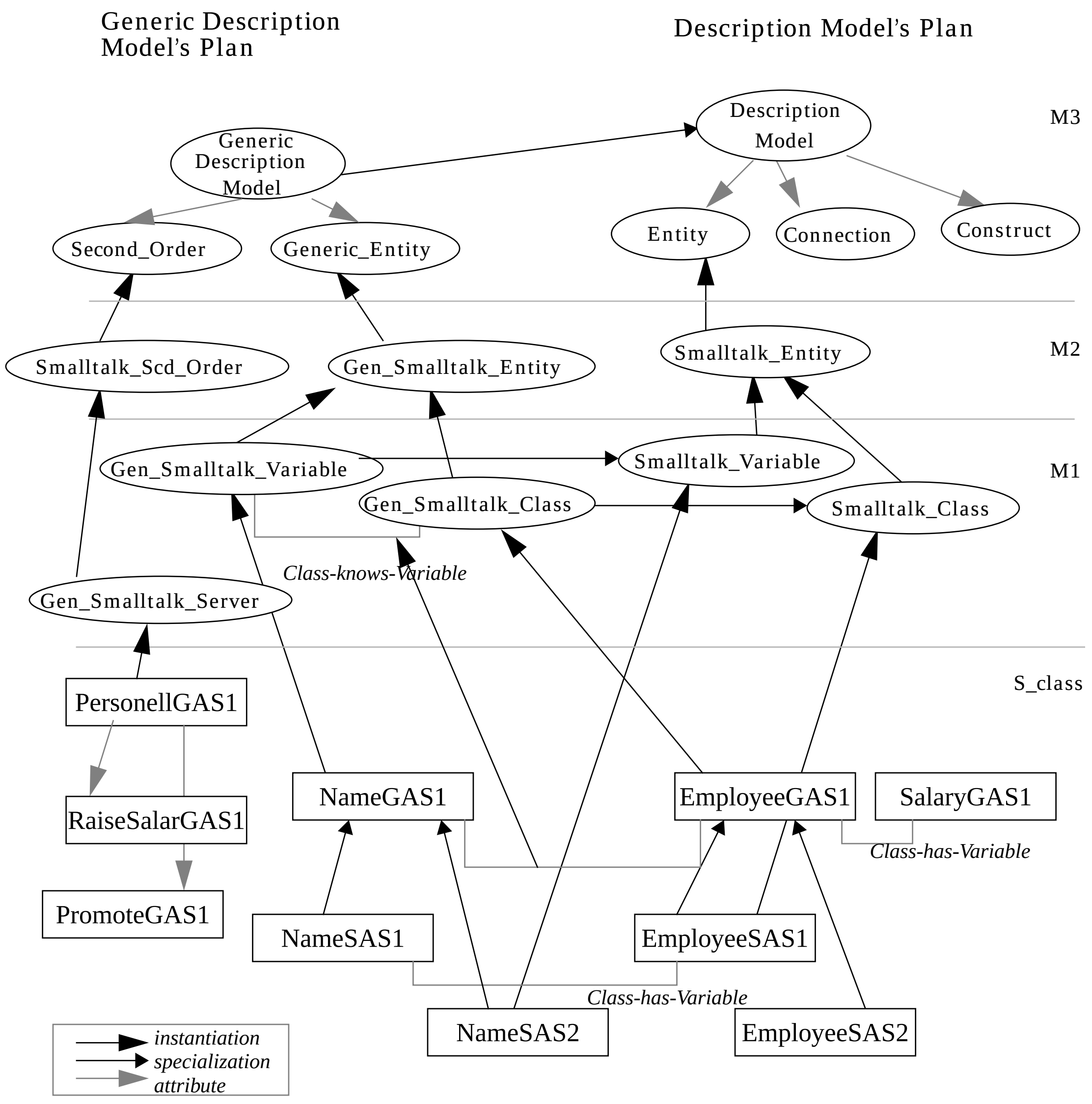

Figure 13 — Generic Application Frame Meta Classes

A new application is built by applying a reuse task to a set of domain-specific software descriptions, producing an application-specific related set of software descriptions. In this short section we roughly describe some of the basics mechanisms that constitute the model of the reuse task.

6.1 Design Operations

The development method is partially described and supported by a set of *design operations*. A design operation is an atomic operation that can be applied to a generic or concrete software description in order to produce another more suitable description for the specific application being built. So, one important feature of a design operation is determining the type of both the input and output concepts which are dependent on specific description models, generic or not. Design operations can be seen as the object-oriented formulation of transformation rules as they appear in program transformation systems. For a survey on this kind of systems the reader may refer to [34].

We classify design operations into two main types:

- *Refinement*

  when the produced software description is at a lower abstraction level compared to the source one. Establishing fixed abstraction levels can be a rather arbitrary decision. Sometimes it is easy to recognize a change of abstraction level, for example when the software descriptions are represented using two different models, and the models have been commonly accepted as models at different abstraction levels. Changes of abstraction levels using only one model can be easily identified in some models (like functional models) or can be arbitrated by the AE.

- *Adaptation*

  when the produced software description is at the same abstraction level as the source one. Normally this happens when creating a concrete description from a generic one, or by adapting an existing past experience description.

A design operation can vary from a purely manual one, in which the AD applies his own knowledge by composing existing software descriptions, to a purely automated one, in which the output is completely generated by the operation. Accordingly, the knowledge inside the design operation will vary from simple guidelines in textual form, to complete operational methods of performing the operation. We call *transformation* a design operation (refinement or adaptation) that can be performed without AD intervention; i.e., the AD commands the transformation, but does not need to supply extra knowledge to accomplish the task. Design Operations are also represented as objects in [22], in which the knowledge of how to perform the transformation is embedded in tools, also seen as objects.

A given description model introduces its own design operations, which can always be classified as a refinement or adaptation. For example, the conceptual design primitives modification, specialization and personalization as presented in Ithaca Object-Oriented Methodology [10] are adaptation design operations. Design operations between models can also be defined; they will almost always be refinement ones, or adaptations from the generic description model to the concrete one. Additional behaviour may need to be defined in meta classes pertaining to the generic or concrete description model, in order to cooperate with the needs of design operations.

Domain Design Operations are based on Design Operations. The latter have the knowledge about descriptions models and how to carry on transformations among software descriptions. The former incorporate domain specific knowledge. Domain design operations are introduced as specializations of design operations during the domain analysis process. They redefine design operations, which are domain-independent, restricting or enhancing the degree of freedom of the reuse process with respect to a given domain. Two main reasons urge us to create domain-dependent design operations. The first one is because of the nature of the domain. Quite often a description model is not completely adequate to describe a particular domain. Domain design operations can help in overcoming the model limitations by adapting the way that software descriptions are worked on during the development process. The second reason is introduced by the knowledge the AE has about a particular set of concept instances in the domain. As an example, consider one design operation that knows how to take an entity described in a semantic data model and create a relational model table. A domain design operation would add domain-specific knowledge, concerning how to transform the particular instance “Employee” entity in a particular “Employee” relation, depicting, for example, guidelines to choose the primary key.

Finally, we note that as domain-information is pushed into a design operation it can be more highly automated, becoming easier to achieve a generation-based approach.

Design Operations (domain-specific or not) are supported by objects. The knowledge of how to perform a transformation is codified as object behaviour. Inheritance allows us to define domain design operations based on domain independent ones.

6.2 Reuse Plans

The Reuse Task is performed by the AD with the aid of a set of Reuse Plans. Reuse Plans can be seen as global transformation rules, or higher level design operations. As design operations, reuse plans are first defined according to inter- and intra-relationships between description models. An example of a Database design domain-independent Reuse Plan is one that establishes a top-down development process by ordering a set of design operations that transforms a conceptual semantic data schema into a relational one. More interesting Reuse Plans are obtained in the process of domain analysis, in which domain design operations are gathered in a domain dependent reuse plan.

Reuse Plans help the AD in both the tasks of:

- composing or generating the application’s parts at a given abstraction level and
- proceeding to the next level.

They have the knowledge that will guide the AD in the task of reusing the available information. They are responsible for providing:

- suggested contexts and ordering for considering contexts and concepts (or clustering of concepts) when building a new application. The objective is to reduce the overloading of concepts that the AD has to deal with at the same time;
- a coherent set of domain design operations that can be applied to the concepts above, in a coherent order;
- possible assumptions to be verified in the real world (application being built) before selecting a concept (or cluster of concepts).

A Reuse Plan not only knows about related software descriptions in a given domain, but also provides suggestions for accomplishing the development process. *So, a set of related Reuse Plans can also be seen as a* ***software development method*** *specially tailored to a specific domain using a set of representational models.*

Section 7 illustrates a method to create design operations and reuse plans taking the reuse infrastructure as a starting point.

7 A method for Generic Application Frame

design

We now sketch a method for GAF design based on the discussion above. We present it in sequential substeps, but the process is highly iterative and may be executed by several people. Each substep, which is here only summarized, may involve a considerable development effort. The method also helps in gathering and showing how to relate all the conceptual ideas introduced in the previous sections. To do so, we provide cross references to the text.

The method defines two main phases. The first one prepares the needed models for GAF design, and the second one performs the task of designing the GAF itself.

The first phase involves a different skill than the second one, defining a different role for the Application Engineer. It needs to be performed when the available description models or domain models are not suitable for the specific domain for which the AE needs to create a GAF.

Anyway, the process is iterative. As description models and domain models are tested in real GAF design situations, they may need to be further enhanced and refined. After a period of GAF creation and subsequent refinement of the models, the first phase activity is expected to decline.

Phase I: Preparing for GAF Design

1. Creating Description Models

Step1 will feed the SIB with descriptions models that are suitable to describe Specific Application Schemas. Besides describing the models, this step works towards establishing semantic links among the models that will be useful to support the past experience structure. The substeps are the following.

1.1 Defining which description models will be used

Choose a set of adequate description models to represent requirements, designs and implementations in the target application domain. Experience in working with the domain helps in this task. Also include a description model if one expects to find out relevant past experience described using it.

1.2 Defining GAS/SAS structure types

Establish abstraction levels (section 3.2) using the set of description models described in substep 1.1. One description model can support several abstraction levels, and one abstraction level can be supported by different description models. In the first case, abstraction levels may have to be arbitrated. The second case happens thanks to differently represented past experiences or because of different purposes (i.e, data view, process view, state transition view, etc.). This substep sets up the GAS/SAS lattices types that will exist. There will be one for each abstraction level in a description model.

1.3 Describing the description models to the SIB

Describe each description model, classifying its concepts as Entities, Connections or Constructs (section 5.5). Connection concepts are binding relationships between Entities concepts. Constructs are valid ways of aggregating Entities and Connections. They can be seen as the rule set of the description model grammar, while Entities and Connections are similar to tokens. The M3 meta class level of the description model provides the framework (structure and behaviour) that helps defining this classification. Use M2 meta classes to model the particular aspects of the description model's proposed classification in relation to the general framework. Finally, M1 classes will detail in their structure and behaviour the semantics of the description model.

It may be necessary to return to this substep in two situations. First, when defining the Reuse task (see step 3) it may be necessary to provide extra behaviour in M1 classes in order to make them cooperate with design operations and reuse plans. Second, while defining the corresponding generic description model (see step 2.2), it may be necessary to return to this step to promote relationships to concepts (section 5.3).

1.4 Creating software-development relationships

Establish useful software development relationship types (section 4.1) among software descriptions necessary to trace possible ways of relating them during the developing process, including ones for inter and intra abstraction levels. This work will be done by modifying M1 level classes. Useful suggestions of relationship types can be derived from the SIB's correspondence links as for example, derivedFrom, impliedBy, dependsOn, explainedBy and provedBy [37]. A software-development relationship type represents a small kernel that will be developed into a design operation. A relationship type that links software descriptions at the same abstraction level represents an adaptation design operation kernel, while one that binds two different levels represents a refinement design operation kernel.

2. Creating Domain Models

Step 2 will feed the SIB with domain models that are suitable to describe generic schemas, modelling alternatives and justifications. Also the internal class lattice supporting past experience structure will be defined. These classes will be instances of the concrete description model defined in substep 1.3 or of the generic description model defined in substep 2.2 (section 5.5). The substeps are the following.

2.1 Preparing to support modelling alternatives

2.1.1 Inside a description model

Record ambiguities in the description model at the M1 meta level (section 5.4). Create possible supporting justifications for adopting each possible alternative. Domain-specific past experience has to present justifications based on this justification model. These recorded model ambiguities also represent adaptation design operations kernels.

2.1.2 Between description models

If one expects to discover past applications represented by different description models (although with the same purpose) but depicting the same abstraction level information, record equivalences and alternatives between models as done in 2.1.1. If description models at the same abstraction level have different purposes, record aggregation types of model's concepts that can represent different views of the same real world concept.

Substep 2.1 will provide the means not only for modelling alternatives but also to relate the user-understandable domain taxonomy with the formal models (section 5.4).

2.2 Defining Generic Description Models

Create the generic description model for each defined (concrete) description model in step 1 (section 5.1), following the substeps:

2.2.1 Use each concrete description model concept to create first-order generic concepts. Generic first-order concepts are candidates to be specializations of corresponding M1 classes in the M1 meta level of the concrete description model (section 5.5). Use this framework to start reasoning about what useful information is generic about each concept. Build upon the semantics of the concrete description model to describe M1 classes in the generic plan. Think about the meaning of described Constructs in the concrete description model to model the generic ones. They are important because GAS nodes will be instances of Construct M1 class of the generic description model.

2.2.2 Consider sets of interesting first-order concepts and concept relationships as candidate second-order concepts.

2.2.3 Reason about the generic description model obtained so far to see what new generic relationships must be included in the generic description model (as relationships in M1 level) and what concrete relationships need to be promoted to concepts (section 5.3).

2.2.4 Describe rules that must hold between a GAS and a SAS in the description model; i.e, describe valid as well as needed adaptations that must be done in a generic description to get a concrete one. This work will create relationships between generic concepts and concrete ones, corresponding also to adaptation design operations kernels.

2.2.5 Create a classification for generic concepts that can be based on criteria such as size, relevance of the concepts, degree of genericity (distance to the nearest SAS).This constitute reuse plans kernels that will be developed into reuse plans that help the AD in composing an application in a fixed abstraction level taking genericity into account.

2.3 Creating Meta-Refinement link types

Define software development relationships types (as described in substep 1.4) taking the generic description model into account. Define possible useful refinement link types between generic description models at different abstract levels (section 3.3). Explore their relationships with the refinements between concrete description models, defined in substep 1.4.

2.4 Creating a model to capture Justifications

Define a model for justifications (section 4.1), describing justification types and useful places in the concrete or generic description model in which a justification is awaited. Use the three level of meta level classes in both generic and concrete description models as a starting point to consider what types of justifications can be expected. An example of a model that supports design reasoning and rationale can be found in [38].

2.5 Defining the internal past experience model structure

Define a method to model the internal relationships that will support the GAS/SAS lattice, based on the facilities provided by the object-oriented supporting environment. Appendix 1 shows a method of how to do it in TELOS and an example based on figure 1. One requirement of this method is that leaves of the lattice be instances of the concrete description model, while other nodes are instances of the generic one.

3. Creating Domain-Independent Reuse Task

Step 3 will create the domain-independent, but model-dependent Reuse task. The substeps are the following.

3.1 Defining domain-independent Design Operations

3.1.1 Define Adaptation Design Operations taking as a starting point design operation kernels established in 1.4, 2.1 (adaptation of past experience) and in 2.2.4 (making generic concepts concrete). These kernel objects have already the knowledge of valid types of input and output software descriptions. Provide behaviour to them in order to guide the reuse process. The more manual the process is, the more the AE should relate design operations and the justification model.

3.1.2 Define Refinement Design Operations using kernel ones established in 1.4 and 2.3, following the same guidelines as 3.1.1.

3.2 Defining domain-independent Reuse Plans and Reuse Task

3.2.1 Define one (or a set of) General Approache(s) for Software Development using the set of description models. One of the chosen approaches may need to create a specific second order generic concept, for example, a data base bottom-up design approach may need further consideration of useful ways of relating attributes, rather than entities.

3.2.2 Define Adaptation Reuse Plans, taking into account adaptation design operations. Establish coherent orders of applying design operations. This can be represented as a cyclic graph in which it is possible to represent steps in the reuse plan, depicting for each step which are the possible applicable design operations, with the most promising expected results. This is so because design operations can have different power and influence area.

3.2.3 Define Refinement Reuse Plans as 3.2.2.

3.2.4 Define the interaction among adaptation and refinement reuse plans (using a reuse plan) according to the general approach established in 3.2.1. This constitutes a domain-independent Reuse Task.

Phase II: Doing GAF Design

4. Structuring Past Experience

Step 4 can be seen as a preamble to domain analysis. It will collect and structured existing software descriptions created during past developments. The substeps are the following.

4.1 Collecting past software development information

Collect all the information available about past developed applications in the target domain. At this point some reverse engineering may be needed to get to schemas described in known description models.

4.2 Creating the domain past experience structure

Partition the collected schemas according to the models and abstraction levels, using the frame established in substep 1.2. Decompose the collected schemas according to the description model definition established in substep 1.3. Create one past experience lattice for each combination of description model and abstraction level by factoring out common relationships and creating appropriate supporting classes according to the method defined in 2.5. This creates the supporting internal class hierarchy, each class being an instance of a concrete or first-order generic description model concept. The GAS/SAS lattice is also created, each SAS node being an instance of a Construct concept and each GAS being an instance of a generic Construct concept.

4.3 Relating past structure to the software development process

Create semantic relationships among descriptions in the past experience structure by instantiating the software-development relationship types defined in substep 1.4.

5. Doing Domain Analysis

Step 5 is a main one in the GAF design process, it will actually create the GAF. The substeps are the following.

5.1 Creating the Domain-specific infrastructure

5.1.1 *Understanding differences among past developments*

Use the past experience structure defined in step 4 as a starting point. Record in the justification schema the reasons that caused differences between past developed applications.

5.1.2 *Capturing domain modelling alternatives*

Look for alternative ways of representing real-world concepts, by comparing several SAS and by observing the real world. Identify real-world concepts and record them in the user-understandable domain dictionary (section 5.4). Identify for each real world concept the most reasonable or recurrent way of representing it. Create one shadow SAS for each SAS needing to be restructured by translating concepts to the chosen unifying alternative. Provide justifications between the real SAS and the shadow one. Restructure the GAS/SAS lattice using shadow SAS (return to substep 4.2). Collect and record suppositions to be tested in the real world that will help the AD in taking a design decision concerning to alternatives.

5.1.3 Creating Generic Descriptions

Generalize the concepts of the SAS using the generic description model as a conceptual tool. Start with first-order and then pass to second-order. Look for other generic concepts that can be regarded as useful in the domain but that do not have any past experience supporting evidence. Create relationships between generic and concrete software descriptions according to substep 2.2.4. Extensionally existing relationships between first-order generic software descriptions and concrete ones are already provided by the GAS/SAS lattice internal structure, defined in substep 4.2.

Provide any useful hypothetical past experience, drawn from acquired experience in dealing with the domain.

Finally, create instances of meta refinement relationships defined in substep 2.3.

5.1.4 Creating the user-understandable domain dictionary

Create a domain taxonomy taking real world concepts recorded in substep 5.1.2. Record the taxonomy and cross references in the dictionary.

5.2 Creating the Domain-specific Reuse Task

5.2.1 Defining Domain Design Operations

Take as a framework model-dependent design operations, defined in substep 3.1. Introduce what is particular about the development process considering domain characteristics and individual instances of software descriptions (section 6.1). Instances of domain specific relationships among software descriptions have been created during substeps 5.1.3 and 4.3. Remember that domain design operations are based on domain-independent ones. The later ones, however, are drawn from kernels that are in fact relationship types.

5.2.2 Defining Domain Reuse Plans

Domain-specific Reuse Plans will further refine what is already stated in model-dependent reuse plans by establishing suggestions of ordered working contexts, first-order fragments, second-order fragments, and GAS nodes that need to be tacked, providing a guided collection of suggestions. Domain Reuse Plans use model-dependent design operations when domain design operations are not available.

5.2.3 *Defining the Domain Reuse Task*

Define interactions among adaptation and refinement Domain Reuse Plans created in substep 5.2.2, if they are available, taking a domain-independent reuse task (substep 3.24) as a framework. As a consequence, the Domain Reuse Task follows the corresponding development approach defined in substep 3.2.1.

8 Tools for improving reusability

We now consider two cooperating tools to support reusability. A *GAF Design Tool*, valuable for the Application Engineer to design and create a GAF, and a *SAF Design Tool*, used by the Application Developer, which is the tool that will really perform the reuse task. The tools have in common the ability to work with several models and sharing a common set of software descriptions stored in the SIB. Also, as a new application is developed, it is necessary to store a new experience and probably reconsider the Domain Schema of that particular domain. In order to achieve reuse in several development phases and to provide an evolutionary approach to the reusability problem, the main characteristic of these tools is extensibility. They should be capable of handling a set of software development methods and models.

8.1 SAF Design Tool

We can talk about a family of SAF Design Tools, each one specially designed to deal with a specific model or development phase. In general, they should have the following set of responsibilities that define a common layer among them.

• *Provide browsing facilities through past experience and domain schema, hiding as much as possible the complexity of the SIB.*

During the browsing process, it should be possible to choose between displaying only the Domain Schema, the Past Experience Structure, or both. Considering the Past Experience Structure, for example, instead of only showing the hierarchy of SAS and GAS as a tree, it should be possible to have a superimposed view of a set of SAS and GAS. As each node is in fact a network of related concepts, the tool could display a union of all these networks. The more frequently recurring concepts — corresponding to the root GAS — could be displayed with the thickest lines, and so on, up to the concepts appearing only once in specific applications, that would be displayed with very thin lines. In this way, it is possible to have in a glance an overview of the main concepts and also of all the specific applications. Figure 14-a shows an idea of such representation, for the SAS in figure 1. Of course, if the diagram is too cluttered, it could be necessary to prune the tree (in order to see only the main concepts, figure 14-b) or to follow a specific path (when the AD finds a pattern of similarity between the new application and the hierarchy, figure 14-c). Several hypertext techniques could be used to browse the tree. Different weights can be given to the concepts of the model, to allow multiple superimposed views.

The superimposed view supported by the tool can also help during the evolution process. When creating a new SAS, it is much easier to manipulate a graphical representation of a schema such as the one of figure 4-a, and let the system automatically generate the new classes like the ones of figure 5 than having to deal with the complexity of the concepts specialization hierarchy.

Another important feature would allow the AD to step from a superimposed view of SAS and GAS nodes to a generic description. The AD could toggle between these two views. He could understand a generic concept in the generic model and quickly have a look on the past experience supporting evidence about the same concept.

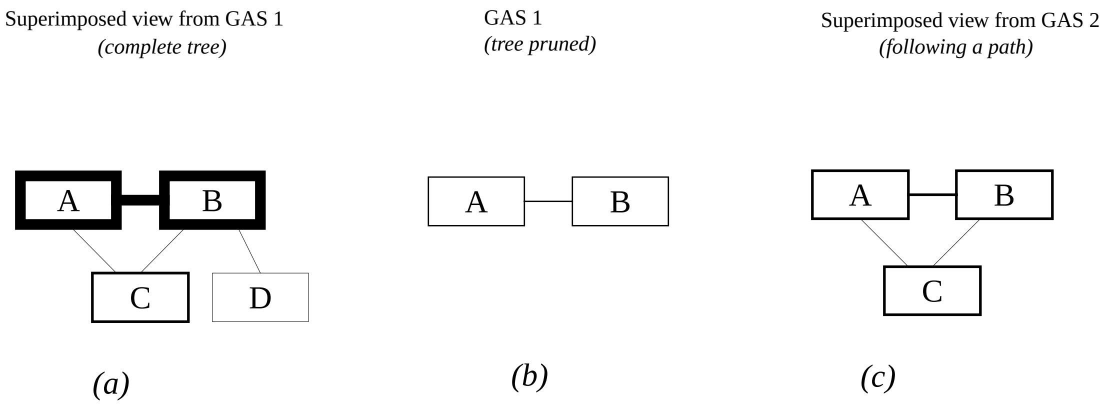

Figure 14 — A tool for browsing past experience

- *Guide the reuse task, applying Reuse Plans*

  These include the following.

  - suggesting development tasks based on domain-knowledge;
  - displaying and keeping tracking of assumptions to be verified in the real world, even providing an agenda to coordinate the AD's work;
  - advising the AD about the decisions that he or she is taking during the reuse task. For example, on following a past experience GAS/SAS path, the tool can advise if the AD is choosing a set of concepts for which a corresponding set of refinements or domain specific transformations are already defined. If not, the tool can give a measure of the amount of existing generic concepts at the subsequent abstraction levels, so that it is possible to assess the future development effort and the consequences of taking a decision;
  - showing past developments, for example, playing back an particular application development process, helping the AD to understand the rationale of a project. This can also be very useful for maintenance;

8.2 GAF Design Tool

The GAF Design Tool should at least help the AE in storing the necessary information, such as:

- description models and their changes to capture Domain Schemas;
- software descriptions;
- knowledge of how to perform the reuse task.

In this case, all the “intellectual” work is done by the AE, and the GAF tool only holds the necessary information, supported by the SIB. But the GAF tool could go one step further and help the AE in the process of gathering and structuring the GAF information, i.e., during the tasks described in section 7.

The first phase, namely *Preparing for GAF Design*, can be seen as a task of a GAF design meta-tool, i.e., one that is domain independent and will allow the GAF design tool to work with several different models. It guarantees the extensibility of the tool.

Figure 15 depicts the architecture of our proposed GAF Design Tool, and its relationship with the SAF Design Tool.

The *Model Loader* is responsible to load the SIB with Description Models which will be useful to describe software pieces of specific applications (new ones and past experience). The *Domain Model Adapter* is responsible to take a software development model and adapt it to make it capable of capturing domain information, including alternatives, generic concepts, justifications and the set of model-dependent (and domain-independent) design operations and reuse plans. These two modules are responsible for the extensibility of the tool.

The *GAF Design Modeller* module supports the task of structuring past experience, doing domain analysis and also defining the domain dependent reuse method. Its output is a Domain Schema that shows the main concepts of a domain and its interrelationships. Also, it outputs how reuse is performed in a particular domain, comprised of Domain-dependent Reuse Plans and Domain Dependent Design Operations. The *SAF Design Tool Common Layer* provides the common facilities needed to any Application Developer development tool conforming to the GAF’s model.

9 Conclusion

We have identified some of the problems in achieving reusability including lack of understanding the reuse process; insufficient requirements for a domain model’s capabilities; lack of understanding what comprises a generic software description; and lack of understanding of how to perform a process of domain analysis and how to represent the output of such a process.

We have proposed an object-oriented based structure to keep domain information in which the suggested software components can be properly scaled and are represented by collections of object-oriented lattices which intertwine generic concepts, alternatives, and justifications, being supported by a collection of concrete past developed examples. We’ve also shown how a domain-specific development method can be defined based on both a reuse-based approach and

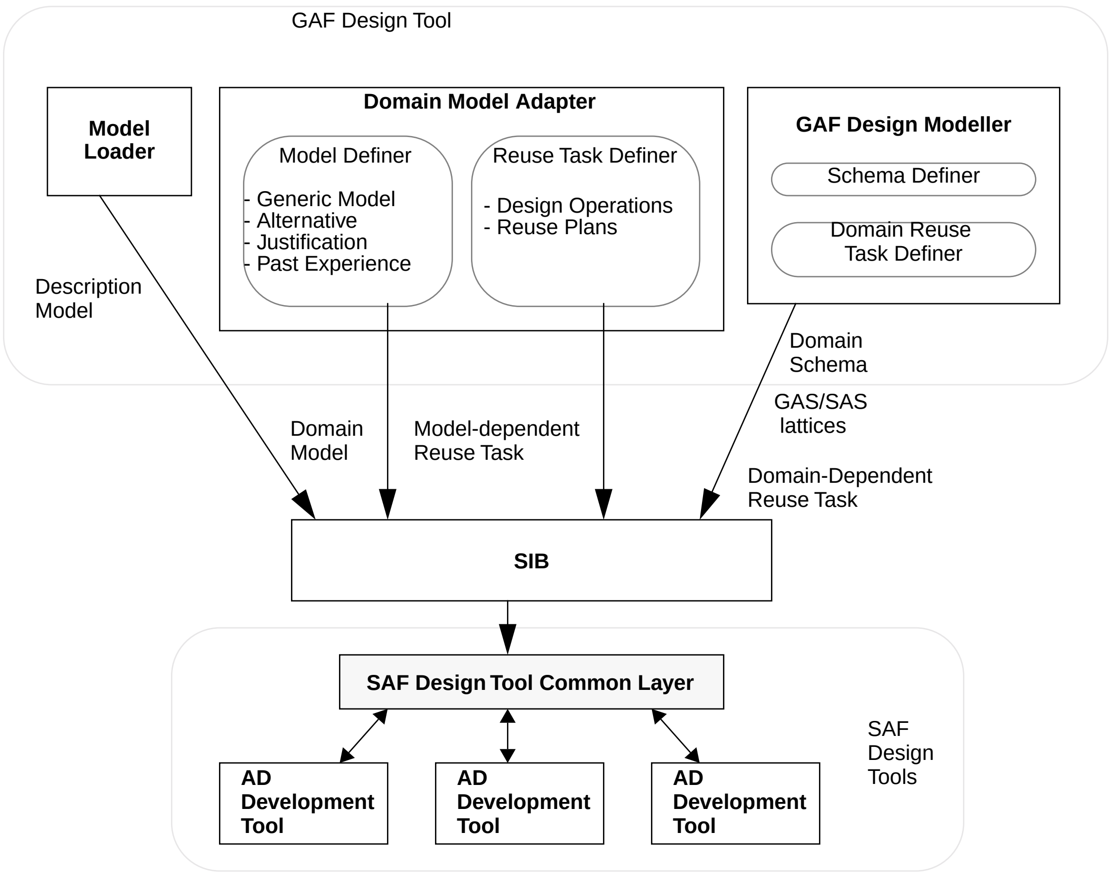

Figure 15 — GAF Design Tool Architecture

domain knowledge, in which the development of a new application would be assisted by the domain information containing intensional knowledge as well as extensional supporting evidences. Finally we've also sketched a method for GAF design.

Many issues are still to be addressed. One of the problems is to effectively manage and support class evolution [7], which is needed to maintain domain knowledge. Another concern is computer supported cooperative work [21], necessary to coordinate the interaction among application engineers, application developers and users. A third issue is the relationship between application domains, since they do not present sharp boundaries. One can develop a new system by also exploring another available intermingled domains, leading to the problem of how to face broad scope domains.

Creating a comprehensive Generic Application Frame is by no means an easy task. It involves doing a lot of problem domain analysis and it is more complex than application development, an activity that despite all the research and pragmatic efforts expended so far is yet not really completely understood. We cannot wait for further advances in application development technologies before starting to tackle the GAF design problem. Rather, we argue that approaching a software development process with reusability issues in mind will contribute to improve quality by the simple fact that it will force a more inclusive and better supported perception of the subject. Also, a reuse-based software development paradigm can turn out to be the solution to the software crisis, as many people working on this problem believe.

Acknowledgements

The authors are thankful to Jean Bell for her useful comments about this paper.

References

[1] G.F. Arango, “Domain Engineering for Software Reuse, Ph.D. thesis, University of California, Irvine, T.R 88-27, 1988.

[2] P.G.Basset, “Frame-Based software Engineering”, IEEE Software, july 1987, page 9-16.

[3] C.Batini, M.Lenzerini, S.B.Navathe, “A Comparative Analysis of Methodologies for Database Schema Integration”, ACM Computing Surveys, Vol.18, No.4, December 1986, pp 323-364.

[4] C. Bernard, D. Mellgren, “RMT specification”, ITHACA.CUI.90.E.#1 technical report.

[5] T.J.Biggerstaff, A.J.Perlis, “Software Reusability”, Frontier Series, ACM Press, 1989

[6] G.Booch, “Object Oriented Design with Applications” Benjamim/Cummings Publishing Company, 1991, ISBN 0-80-53-0091-0

[7] E.Casais, “Managing Evolution in Object Oriented Environments: An Algorithmic Approach”, Ph.D. thesis, Université de Genève, 1991.

[8] P. Coad, E. Yourdon, “Object-Oriented Analysis”, Youdoon Press, Prentice Hall, Englewood Cliffs, 1990, ISBN 0-13-629122-8

[9] P. Constantopoulos, M.Doerr, E.Pataki, E.Petra, G.Spanoudakis, Y. Vassiliou, “The Software Information Base-Selection Tool integrated prototype, ITHACA.FORTH.91.E2.#3, 12/1/91

[10] V. de Antonellis, B.Pernici, “ITHACA OBJECT-ORIENTED METHODOLOGY MANUAL INTRODUCTION and APPLICATION DEVELOPER MANUAL (IOOM/AD)” version 0.0, october 1991

[11] V. de Antonellis, B.Pernici, P. Samarati, “F-ORM METHOD: A F-ORM Methodology for Reusing Specifications”, ITHACA.POLIMI.90.E2.7#3 technical report.

[12] V.de Mey, B.Junod, S.Renfer, M.Stadelmann, I.Simitsek, “The Implementation of VISTA — A Visual Scripting Tool”, in Composition d’Objects, edited by D. C. Tsichritzis, Centre Universitaire d’Informatique, 1991, pp 31-56.

[13] P. Freeman, “Software Reusability” Tutorial, IEEE Computer Society Press, 1987.

[14] P.Freeman, “A Conceptual Analysis of the Draco Approach to Constructing Software Systems, IEEE Transactions on Software Engineering, Vol. SE-13, No.7, july 1987, pp 830-843.

[15] M.Fugini, S.Faustle, M.Theodoridou, D.Vista, D.Nastos, “Technical Description of the Selection Tool”, ITHACA.POLIMI-FORTH.90.E3.5.#2, 12/1/90

[16] M.Fugini, M.Guggino, B. Pernici “Reusing Requirements through a Modeling and Composition Support Tool”, ITHACA.POLIMI-.90.E2.7.#1, december, 1990.

[17] M.G.Fuigini, O.Nierstrasz, B.Pernici, “Application Development through Reuse: the Ithaca Tools Environment” in Composition d’Objects, edited by D. C. Tsichritzis, Centre Universitaire d’Informatique, 1991, pp 9-114.

[18] S.Gibbs, G.Kappel, “The ITHACA Application Development Environment Process Model and Tools Scenario”, technical report, ITHACA.CUI.89.E.#9,june/1989

[19] S.Gibbs, D.Tsichritzis, E.Casais, O.Nierstrasz, X.Pintado, “Class Management for Software Communities”, Communications of the ACM Vol. 3 No.9, Sept. 1990.

[20] A.Goldberg, “Smalltalk-80: The Interactive Programming Environment”, Addison-Wesley, Reading, Massachusetts, 1984.

[21] U.Hahn, M.Jarke, T.Rose, “Teamwork Support in a Knowledge-Based Information Systems Environment”, IEEE Transactions on Software Engineering, vol.17, no.5, may 1991, pp 467-482.

[22] M.Jarke, M. Jeusfeld, T.Rose, “A Software Process Data Model for Knowledge Engineering in Information Systems”, Information Systems Vol.15, No.1, pp 85-116, 1990.

[23] G.E.Kaiser, D.Garlan, “Melding Software Systems from Reusable Building Blocks”, IEEE Software, July 1987, pages 17-24

[24] T. Korson, J. D. McGregor, “Understanding Object-Oriented: A Unifying paradigm” Communications of the ACM Vol. 3 No.9, Sept. 1990.

[25] K.J.Lieberherr, I.Holland and A.Riel, “Object-Oriented Programming: An Objective sense of Style”, Proceedings of OOPSLA 88, ACM SIGPLAN Notices, vol.23, no.11, pp 323-334, November 1988.

[26] B. Meyer, “Reusability, The Case for Object-Oriented Design”, IEEE Software, March 1987, pages 50-64.

[27] J. Mylopoulos, A.Borgida, M.Jarke,M. Koubarakis, “Telos: Representing Knowledge About Information Systems”, ACM Transactions on Informations Systems, Vol. 8, No 4, October 1990, pp 325-362.

[28] J.M.Neighbors, “Draco: A Method for Engineering Reusable Software Systems”, in Software Reusability, ed. T.J.Biggerstaff and A.J.Perlis, ACM press, 1989, pp 295-319.

[29] O. Nierstrasz, “The ITHACA Application Development Environment — Rationale and Approach”, TWG Interim Report, ITHACA.CUI.89.E#8, june, 1989.

[30] O. Nierstrasz, “A Survey of Object-Oriented Concepts” In Object-Oriented Concepts, Databases and Application, (Ed. W. Kim and F. Lochovsky) Addison-Wesley/ACM Press, 1989, pp 3-21.

[31] O. Nierstrasz, “The ITHACA Application Development Environment — Overview of Technical Report TR.E.1”, ITHACA.CUI.90.E#1, june, 1989.

[32] O. Nierstrasz, “The ITHACA Application Development Environment — Overview and Abstracts of the TWG Interim Report”, ITHACA.CUI.89.E#10, 1990.

[33] O. Nierstrasz, D. Tsichritzis, V. de Mey and M. Stadelmann, “Objects + Scripts = Applications,” in *Proceedings, Esprit 1991 Conference*, Kluwer Academic Publishers, Dordrecht, NL, 1991, pp. 534-552.

[34] H.Partsch and R. Steinbruggen, “Program Transformation Systems”, ACM Computing Surveys, Vol. 15, No. 3, September 1983, pages 199-236.

[35] B. Pernici, “Objects with Roles”, Conference on Office Information Systems, Cambridge, Massachusetts, April 1990 pp 205-215

[36] B. Pernici, “Class Design and Meta-Design” in Object Management, edited by D. C. Tsichritzis, Centre Universitaire d’Informatique, 1990.

[37] E. Petra, C. Vezerides, “SIB Contents’ Manual”, draft of version 1.0, january 1991, ITHACA.FORTH.91.E.2.#4 technical report.

[38] C. Potts, “A Generic Model for Representing Design Methods”, in Proceedings of the 11th International Conference on Software Engineering, pp 217-226, Pennsylvania, May 1989

[39] R. Prieto-Diaz, “Domain Analysis for Reusability”, Proceedings of COMPSAC’87, 1987, pp 23-29.

[40] R. Prieto-Diaz, “Domain Analysis: An Introduction”, ACM Software Engineering Notes, vol. 15 no 2, Apr. 1990, pp 47-54.

[41] A.Profrock, D.Tsichritzis, G.Muller, M.Ader, “ITHACA: An Overview” In Proc. European Unix Users’ Group (EUUG) Conference, Spring 1990, 99-105.

[42] B. Henderson-Sellers, J.M.Edwards, ‘The Object-Oriented Systems Life Cycle”, Communications of the ACM Vol. 3 No.9, Sept. 1990.

[43] W.Tracz, “Software Reuse: Emerging Technology” Tutorial, IEEE Computer Society Press, 1988.

[44] Y.Vassiliou, M.Jarke, E.Petra, T.Topaloglou, G.Spanoudakis, C.Vezerides, “Technical Description of the SIB”, ITHACA.FORTH.90.E2.#2,12/1/90

[45] R. Wirfs-Brock, B. Wilkerson, “Object-Oriented Design: A Responsibility-Driven Approach”, OOPSLA 89, pp 71-75.

[46] R. Wirfs-Brock, Object-Oriented Software Development, Prentice Hall, Englewood Cliffs, 1991.

Appendix 1: An example of structuring past experience using TELOS

This appendix defines a method to structure past experience using TELOS knowledge-based language. Adopting this approach would cause a need to increase by two levels the meta classes framework depicted in figure 13.

Rationale:

Following are the necessary steps for structuring past experience using the TELOS Class format.

Step 1

Construct a generalization/specialization hierarchy of *M2_Classes*, that will serve to represent the generic concepts together with generic relationships between them. As a concept can be related to different concepts in different applications, a specialization of a concept correspond to adding new TELOS attributes that refer to additional generic relationships.The generalization/ specialization hierarchy is useful because it represents incremental differences among specific applications. These Classes are labelled:

concept_GACidentification_Class

These are abstract classes, i.e, they do not instantiate and serve to be the generalizations of *concept_SACidentification_Class.*

concept_SACidentification_Class

These are the leaves of the generalization/specialization hierarchy that will be instantiated in the next step.

The word *Concept* in the classes' label is the name of the concept, as it is known in the domain vocabulary. *GAC* stands for Generic Application Concept while *SAC* stands for Specific Application Concept (Note that in the paper we have used GAS and SAS to avoid introducing further detailing). The *identification* correspond to a specific level of generalization (if it is a *concept_GACidentification_Class*) or to identify a specific application (if it is a *concept_SACidentification_Class*.
This Modelling Approach requires that every individual concept of a specific application has a corresponding *concept_SACidentification_Class*. So it is possible to introduce classes that do not add new attributes, i.e., classes that are not essential to represent the model, but that are helpful to the understanding of the similarities among applications. This happens with *concept_SACidentification_Class* as well as with *concept_GACidentification_Class*.
The set of M2_Classes can be seen as a set of generic concepts and their relationships. The generalization/ specialization hierarchy existing in this level represents the similarities among applications.

Step 2

Construct a set of *M1_Classes* that are instances of the corresponding *M2_Classes* of type *concept__SACidentification_Class*. They are labelled *concept_SACidentification*. This set of classes has the purpose of instantiating the generic relationships between the generic concepts which are defined in the *M2_Classes* hierarchy, guaranteeing the type consistency. As a consequence of the requirement stated above, there will be a *concept_SACidentification* class for each concept in a specific application.

The set of M1_Classes can be seen as instantiations of the generic concepts (and instances of generic relationships), as they appear in a specific application. Also, these classes will also be instances of the concrete description model, (for example instances of class Smalltalk_Class in figure 13), while classes defined in the step above will be classes of the generic model (for example, instances of class Gen_Smalltalk_Class in figure 13).

Step 3

The set of *concept_SACidentification* classes with the same *identification* represents the description network of a Specific Application at a given abstraction level. To represent an application, instantiate all the *concept_SACidentification* that have the same *identification* in their labels. This will finally correspond to a real built application. This is necessary because we are defining a general model so, for example, the set of *M1_Classes* defined above can represent a requirements description. Two real applications can have the same requirements description (i.e., the same set of specific concepts and relationships), but different implementations. These two real applications will be represented by two different sets of classes in this level.

The set of classes in this level (*S_Classes*) represents a concrete application which is based in the specific concepts (and their relationships) defined in the above level.

Example

The following example is drawn from figure 1.

Definition of GAS1

```
TELL IndividualClass A_GAC1_Class IN M2_class
WITH
attribute
link1: B_GAC1_Class
END

TELL IndividualClass B_GAC1_Class IN M2_class
WITH
attribute
link1: A_GAC1_Class
END
```

Definition of SAS3

(M2_Classes)

```
TELL IndividualClass A_SAC3_Class IN M2_class
ISA A_GAC1_Class
END

TELL IndividualClass B_SAC3_Class IN M2_class
ISA B_GAC1_Class WITH
attribute
link2: D_SAC3_Class
END

TELL IndividualClass D_SAC3_Class IN M2_class
WITH
attribute
link1: B_SAC3_Class
END
```

(M1_Classes)

```
TELL IndividualClass A_SAC3 IN M1_class,
A_SAC3_Class WITH
link1
l2: B_SAC3
END

TELL IndividualClass B_SAC3 IN M1_class,
B_SAC3_Class WITH
link1
l1: A_SAC3
link2
l2: D_SAC3
END

TELL IndividualClass D_SAC3 IN M1_class,
D_SAC3_Class WITH
link1
l1: B_SAC3
END
```

Definition of GAS2

```
TELL IndividualClass A_GAC2_Class IN M2_class
ISA A_GAC1_Class
END

TELL IndividualClass B_GAC2_Class IN M2_class
ISA B_GAC1_Class
END

TELL IndividualClass C_GAC2_Class IN M2_class
END
```

Definition of SAS1

(M2_Classes)

```
TELL IndividualClass A_SAC1_Class IN M2_class
ISA A_GAC2_Class
END

TELL IndividualClass B_SAC1_Class IN M2_class
ISA B_GAC2_Class WITH
attribute
link2: C_SAC1_Class
END

TELL IndividualClass C_SAC1_Class IN M2_class
ISA C_GAC2_Class WITH
attribute
link1: B_SAC1_Class
END
```

(M1_Classes)

```
TELL IndividualClass A_SAC1 IN M1_class,
A_SAC1_Class WITH
link1
l2: B_SAC1
END

TELL IndividualClass B_SAC1 IN M1_class,
B_SAC1_Class WITH
link1
l1: A_SAC1
link2
l2: C_SAC1
END
```

TELL IndividualClass C_SAC1 IN M1_class,
C_SAC1_Class WITH
link1
l1: B_SAC1
END

Definition of SAS2

(M2_Classes)

TELL IndividualClass A_SAC2_Class IN M2_class
ISA A_GAC2_Class WITH
attribute
link3: C_SAC2_Class
END

TELL IndividualClass B_SAC2_Class IN M2_class
ISA B_GAC2_Class
END

TELL IndividualClass C_SAC2_Class IN M2_class
ISA C_GAC2_Class WITH
attribute
link1: A_SAC2_Class
END

(M1_Classes)

TELL IndividualClass A_SAC2 IN M1_class,
A_SAC2_Class WITH
link1
l2: B_SAC2
link2
l3: C_SAC2
END
TELL IndividualClass B_SAC2 IN M1_class,
B_SAC2_Class WITH
link1
l1: A_SAC2
END

TELL IndividualClass C_SAC2 IN M1_class,
C_SAC2_Class WITH
link1
l1: A_SAC2
END

An Individual Instantiation of Application X using SAS2

Several implementations are possible, given a requirements specification. Suppose the requirements are given by the set of classes composing SAS2, i.e all the class that appear as *concept_SAC2* above. If you have two (or more) applications that use the same set of requirements, they will be distinguished in the environment by creating two sets of instances. For example A_X will point to a different implementation than A_Y.

TELL IndividualClass A_X IN S_class, A_SAC2
WITH
l1
: B_X
l2
: C_X
END

TELL IndividualClass B_X IN S_class, B_SAC2
WITH
l1
: A_X
END

TELL IndividualClass C_X IN S_class, C_SAC2
WITH
l1
: A_X
END

Domain Information Evolution

SAS4 is a new application which will be introduced (see figure 4). It is very similar to SAS2, so the concepts corresponding to *concept_SAC2_Class* will be replaced by new generic concepts, i.e, *concept_GAC3* concepts. These are the changes:

Definition of SAS2

(M2_Classes)

(The following 4 TELL commands correspond to the old definition of SAS2)

TELL IndividualClass A_GAC3_Class IN M2_class
ISA A_GAC2_Class WITH
attribute
link3: C_GAC3_Class
END

TELL IndividualClass B_GAC3_Class IN M2_class
ISA B_GAC2_Class
END

TELL IndividualClass C_GAC3_Class IN M2_class
ISA C_GAC2_Class WITH
attribute
link1: A_GAC3_Class
END