Frameworks by means of virtual types - exemplified by SDL

Rolv Bræk
SINTEF Telecom and Informatics, N-7034 Trondheim, Norway
and Institute for Telematics, Norwegian University for Science and Technology

Birger Møller-Pedersen
Ericsson AS, Applied Research Center,
Software Engineering Laboratory,
P.O. Box 34, N-1361 Billingstad, Norway
and Institute of Informatics, University of Oslo

Abstract: Frameworks have emerged as a very effective way to achieve reuse. A framework provides the basic structure and behaviour of a family of applications, and they simplify the task of application development. Traditionally the framework idea has not been applied to FDTs. This paper contributes to the use of SDL for building frameworks. SDL offers the opportunity to encapsulate object structures and default behaviour in frameworks defined by SDL system types. Specific types of applications are obtained by defining subtypes of framework system types and redefining virtual types. The mechanism is not specific to SDL, but can be applied to any language that supports virtual types.

Keywords: Reusable architectures, Frameworks, SDL

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35394-4_29

S. Budkowski et al. (eds.), *Formal Description Techniques and Protocol Specification, Testing and Verification*

1 WHAT IS A FRAMEWORK?

A framework is a reusable, "semi-complete" application type that can be specialized to produce custom applications. The main property that distinguishes a library from a framework is that a framework embodies a design of a type of systems, while a library just is a collection of related classes. An elaboration of this distinction is given in Table 1.

Table 1: Library and Framework

Library	*Framework*
The application uses library classes, but the library knows nothing about the application	*The framework knows about the application and uses application classes*
No predefined system structure. The system is entirely defined in the application	*Provides structure. The system is (partially) defined in the framework*
No predefined interaction	*Defines object interaction*
No default behaviour	*Provides default behaviour*

Classical frameworks like window systems are defined as a set of related classes. The structure of these frameworks results from dynamically created objects that are kept together by object references. The event loop that dispatches mouse and keyboard events to window objects will use a list of currently active window objects. This list will typically contain different window objects, with the common property that they react on these events. The default behaviour of the framework is to call virtual procedures or callback procedures. The user of the framework redefines these procedures to what is special for the application.

The paper does not address the definition of frameworks by means of composition of components with so-called "hot-spots" where components may be exchanged dynamically. Component-based frameworks rely on a black-box approach: the user of a component only knows its interface and cannot apply specialisation in order to cover specific needs.

2 WHY MAKE FRAMEWORKS USING SDL?

Experience from SDL based development has shown that SDL systems often contains an *application specific* part and an *infrastructure specific* part (implementation specific) which are not clearly separated. The reason is that an abstract (application specific) system has to be supplemented by a large infrastructure part in order to be executable on a given platform. These two parts need to be combined for the purposes of complete system simulation and automatic code generation, but they serve different purposes and follow separate evolution patterns. Therefore an approach is needed that will allow these parts to be efficiently combined into complete systems, while keeping the parts identifiable for evolution and maintenance purposes. A solution to this *divide and combine* problem is to make a framework in SDL. Instead of making the infrastructure part again and again for each new system with the same infrastructure on the same platform, a framework that embodies both the application part and the infrastructure part is useful. The idea is that if a specific system can be made as an instance of a framework, with much of the general properties of the framework isolated in the infrastructure, then the framework will have a potential for being reused as a design. Furthermore application development may concentrate on the central application issues and disregard the infrastructure, while infrastructure development may concentrate on the central infrastructure issues and disregard the application.

In an initial development, the infrastructure aspect may not be obvious. Frameworks will often come as a result of a (successful) initial development, which is to be used as a basis for a new system. If e.g. distribution has been considered and isolated in an infrastructure part, the next system with the same infrastructure, but with a different application part can reuse this framework.The basic idea is that the infrastructure shall provide some general support to the application, and that application development may concentrate on the application parts.

3 HOW TO MAKE FRAMEWORKS IN SDL

The distinction between a library and a framework is in SDL directly reflected by the language concepts *Package* and *System Type*. An SDL specification can be either a package specification or a system specification. A Package in SDL defines a set of types, while a System (Type) defines both local types *and* a set of related instances. The interpretation of a system

specification yields a system instance that is the outmost container of all instances in a system.

3.1 System types

An SDL *system* consist of *blocks*. A block contains *processes* that are extended finite state machines communicating by means of asynchronous signal exchange and remote procedure calls. Blocks are connected by *channels*, while processes are connected by *signal routes*. The connection points are called *gates*, and they are defined as part of the block- and process types. In object oriented terms, *process types* correspond to classes of "active" objects, while *block types* correspond to classes of "container" objects. Gates define simple interfaces in terms of which signals and remote procedures that are allowed. SDL have more entity kinds than system, blocks and processes, but for the purpose of this presentation only these are used. The notion of *specialisation* by means *subtyping* applies to most of the entity kinds of SDL, and also to system. A framework can therefore be defined by a *system type* in SDL, and a specific application of the framework by a *subtype* of the system type.

The framework system type will have a structure where some components may be classified as infrastructure- and some as application components, see Figure 1. The framework will have *virtual types* for those components that should be adapted to specific applications of the system type. These virtual types will be *redefined* when the framework is used as a supertype when defining the system type for a given application. In Figure 1 the system type is defined to consist of a number of blocks. The types of two of these are defined as *virtual block types,* and can as such be redefined in a subtype of the system type. The redefinitions imply that the instances (A1 and I1) specified as part of the framework system (super)type will have redefined properties. The structure specified in the supertype is inherited, while the "contents" of the instances are redefined.

The dashed boxes in Figure 1 are not part of the SDL specification, but indicates the distinction between application- and infrastructure components. The actual borderline between these may not be as sharp as indicated.

Virtual types do not distinguish between being infrastructure or application component types, so infrastructure component types can also be defined as virtual type in order to allow the infrastructure to be adapted to the specific application.

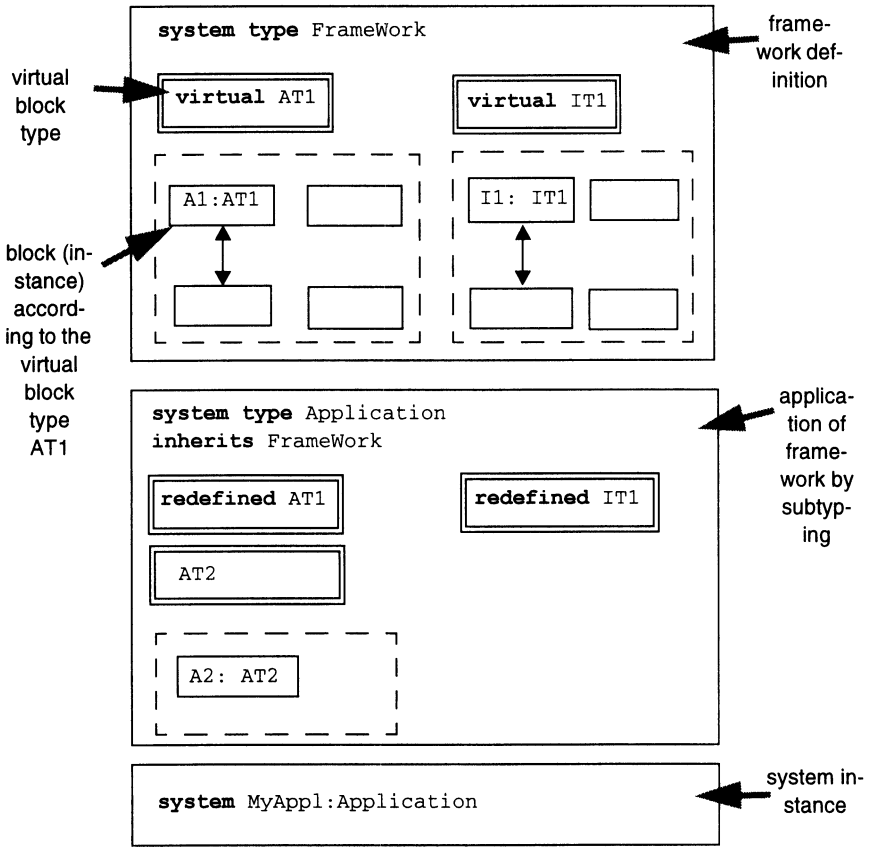

Figure 1: A framework as a System Type

3.2 Applications

In many cases the application components are blocks with some infrastructure specific parts. The block type AT1, see Figure 1, may consist of an infrastructure and an application specific part, each represented by a virtual process type. This is illustrated in Figure 2. This also illustrates that the borderline between application and infrastructure parts is not so sharp as indicated in Figure 1.

AT1_p and IT1_p are two *virtual process types*, and Appl and Infra are processes of these types. When redefining the virtual block type AT1, the virtual process types in this may also be redefined. If the application specific part is to be adapted, then the virtual process type AT1_p is redefined. If the infrastructure part shall be adapted, then IT1_p is redefined.

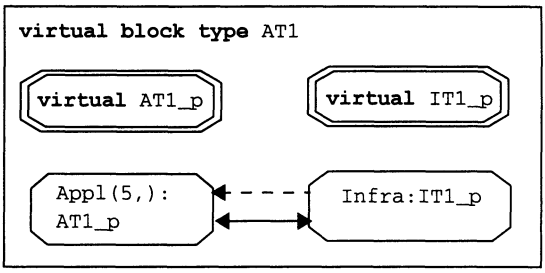

Figure 2: A virtual block type as part of framework, with application specific instances and infrastructure instances

3.3 Instances as part of frameworks

As described above, the way a framework is adapted to specific needs is by redefining virtual types. Redefining virtual types will, however, have no effect if the virtual types are not used to specify instances. These instances may either be specified as part of the framework or be specified as part of the specific use of the framework.

3.3.1 Application specific instances specified as part of the framework

In this case the framework defines the structure of instances with their connections in terms of channels and signal routes. This is the approach used in Figure 1 and Figure 2. SDL is special in the sense that a static structure of instances and instance sets can be specified - instances are not just created dynamically. In a specific application, the properties of the instances are provided by redefining the virtual types, while the structure and connection of instances are inherited from the framework.

The structure of instances and their connections in the framework are obtained by connecting gates of the instances with channels and signal routes. The properties of the instances may be redefined when a specific application is made, but the interconnections between gates cannot be redefined. Gates are defined as part of the virtual types, and as gates are used for connecting instances, the redefinition of these must be constrained. It is important that the framework specification can be analyzed (e.g. that the

connections are valid), and that this analysis is valid for any application of the framework.

This requirement is fulfilled by the notion of *virtual type constraint.* A virtual type in SDL has a constraint (in terms of a type of the same kind), and a redefinition of the virtual type must be a subtype of the constraint. The default is that the constraint of the virtual type is the type definition itself. The gates necessary for the connections - specified as part of the framework - are properties of the virtual type constraints, and are as such guaranteed to be properties of any redefinitions.

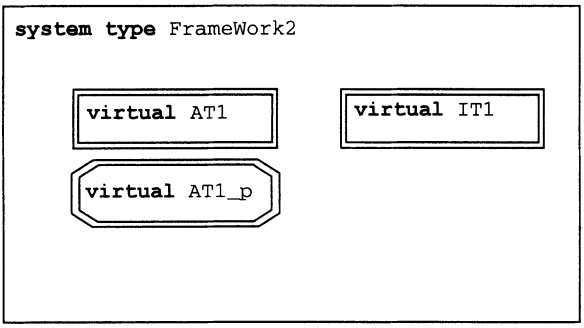

Figure 3: A framework where an application process type is common to many parts of the framework

If the structure is so that the same virtual process type is used to make process sets in many parts of the system structure, then the virtual process type should be moved to the system type, see Figure 3. In this way it can be redefined at *one* place as part of the system subtype and have implication on many parts of the system.

Application specific instances may be added either statically as part of the subtype system definition, as illustrated in Figure 1 (block A2), or they may be created dynamically. Dynamic creation of application instances are anticipated in Figure 2: the Appl specifies a set of maximum 5 processes of type AT1_p. The dashed arrow specifies that the Infra process creates processes in the Appl process set. If new process sets are added in redefinitions of the enclosing virtual type (here AT1), then the infrastructure process must be redefined accordingly to take care of their creation.

3.3.2 Application specific instances not specified as part of the framework

In this case the framework only consists of general types that may be used for the construction of the application part. The infrastructure parts may either contain instances or just be represented by types.

In this situation the creation must be anticipated by the infrastructure. SDL is special in the sense that processes are part of process sets and that creation of processes is done by referring to the name of the set and not to the name of the process type. It is therefore not enough that the infrastructure specific process types are defined in the same scope (e.g. a package or a system) as the application specific types and thereby know these types - they must have means for referring to the process sets that will be part of the specific system.

SDL provides two means for this: *context parameters* and *virtual procedures*.

Context parameters

The general types of the framework that have to create process instances according to application specific types do this through process context parameters. The actual process context parameter is the process set being part of the specific application.

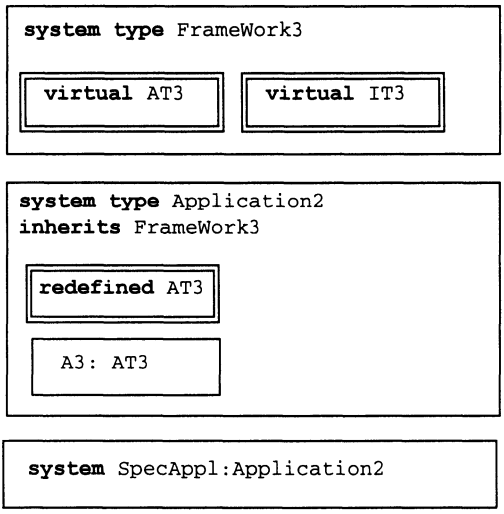

Figure 4: A framework with no instances

The scheme is illustrated in Figure 4, Figure 5, Figure 6 and Figure 7. Within the block types, there will be general process types (infPT and applicationPT) that are used as supertypes in specific systems inheriting from FrameWork3.

The infPT process type shall create instances of type applicationPT, and to this purpose it will have a process context parameter that is constrained by applicationPT, see Figure 7. The idea is that a particular system will pro-

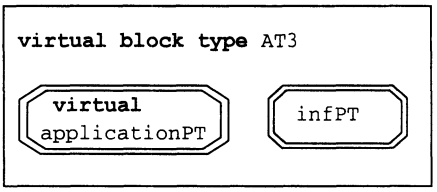

Figure 5: A virtual Application block type without instances.

vide its redefinition of applicationPT *and* its process set. By means of the context parameter, the infPT processes will be able to create instances of the redefined applicationPT without knowing the final process set.

The final system type will introduce the redefined block types with appropriate process sets, see Figure 6, and provide these as the actual context parameters.

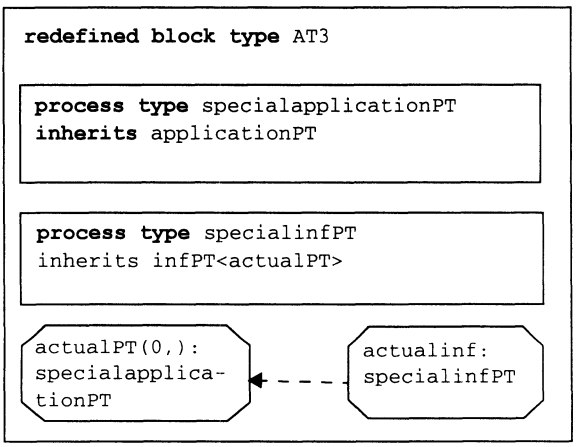

Figure 6: Redefined block type with process set and actual context parameter

The difference from the case with application specific instances as part of the framework is not so big: the process set must be foreseen (the context parameter must be defined - and correspond to a process set), but the name of it and its position in the application part is not determined. Its position will, however, be constrained by the fact that it shall be visible from the place where the actual context parameter is provided, and that it shall be created by a process in the same block.

```
process type infPT
<process cp atleast applicationPT>



 . ...create cp( . . .)
```

Figure 7: Creation of application specific processes

Another constraint with this approach is that it is not possible to specify instance sets of the infPT in the framework itself. The reason is that this process type has context parameters and therefore cannot be used for instance specification.

This approach should only be used in cases where it is important that the framework specific and application specific process types can be defined within the same enclosing block type and where the framework specific types must specify the creation of application specific processes.

Virtual creation procedures

This approach is more general in the sense that it does not have to be decided if there is one or several process sets in the final system type. In the general types where there is a need to create application specific processes, this is represented by corresponding virtual procedures. In the final system type these are redefined to create processes in the actual process sets.

This approach also has the property that instance sets of the application specific types are first introduced in the final system subtype.

In order to provide an example on this way of making frameworks, we use Figure 8. Suppose that it is a requirement that the system shall start by creating processes for each of the actual applications and that changes to the number of application processes shall be reflected while the system is running. Still we would like to define the system as a framework in the sense that it will consist of a common infrastructure and some dynamically created application instances. We assume that the division of responsibility between the two are determined, the communication is fixed and that the functionality of the infrastructure is specified - the only thing that is not specified is the types and numbers of application processes. The configuration of the system is initiated by a new signal (setUp, with appropriate parameters) coming to the infrastructure part of the system.

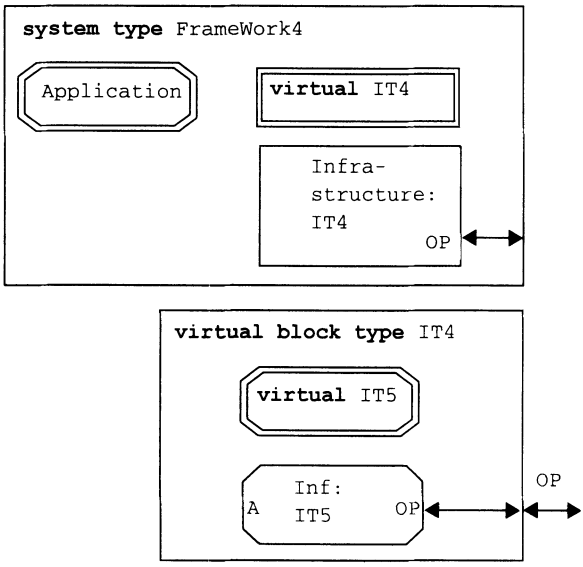

Figure 8: Framework with virtual creation procedures

The setUp signal is supposed to come to the inf process and imply the creation of application processes in the right process sets. Depending on the desired number and types of application processes, the signal will carry enough parameters for the infrastructure to create the right instances.

The infrastructure types can now be defined as before, the only difference being that they will have a virtual procedure setUp and will communicate with possible application processes via a gate that is constrained by Application - that is only process sets of Application or subtypes of Application can be connected to the gate, see Figure 9.

In addition to the normal behaviour and the creation of application processes, the Infrastructure may have behaviour that contributes to the definition of the framework. As an example there may be a limit on the total number of application processes, independent of type of application. The behaviour that ensures this will either be part of the infrastructure, e.g. some action executed each time setUp is executed, or it may be a constraint on setUp which all redefinitions will inherit.

An actual system consisting of two types of application processes is specified as a subtype of the system type Framework4, redefining the setUp procedure to cater for this and introduce the two process sets, see Figure 10.

The names of the process sets are used in the redefined setUp procedure for the specification of the creation of process instances.

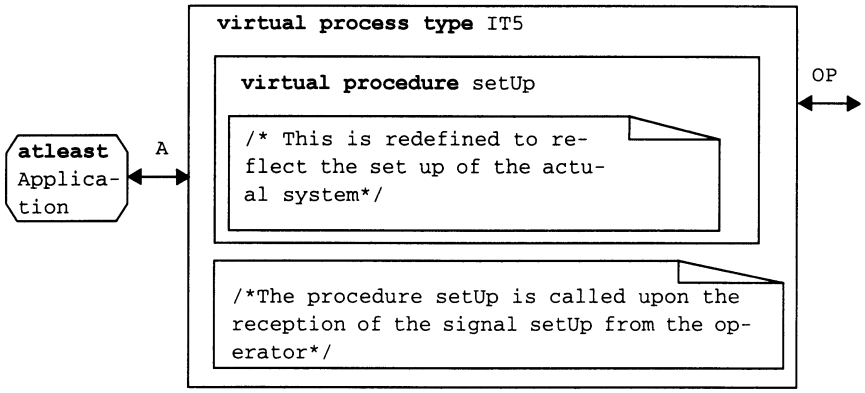

Figure 9: Setup as a virtual procedure of IT5

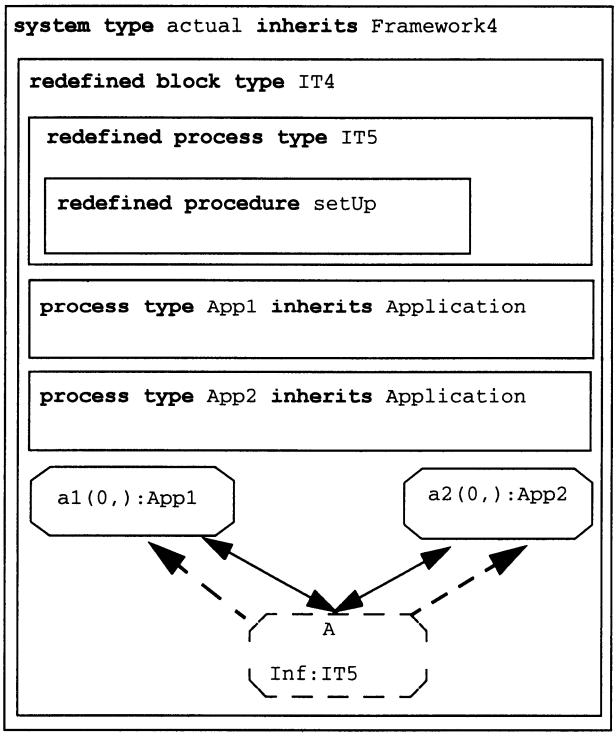

Figure 10: Dynamically created application part

4 RELATED WORK

4.1 Virtual classes/design patterns

The notion of virtual types with constraints where first introduced in BETA (Madsen, Møller-Pedersen & Nygaard, 1993) in terms of virtual patterns. Patterns in BETA is a generalisation of language concepts like class, type, procedure, etc. Virtual procedure patterns provides what is common in most object oriented languages: virtual member functions or methods that can be redefined. The use of virtual patterns to obtain virtual classes is described in (Madsen & Møller-Pedersen 1989).

In (Agerbo, 1998) virtual classes are used to express some design patterns by means of language constructs. It is argued that a design pattern should not be covered by a language construct, in order to be a design pattern, and they show that the Factory Method design pattern can be expressed by a virtual class for the class that varies with the different applications of the design pattern.

Virtual block types and virtual process types as described above for SDL corresponds to virtual classes. A requirement for using this approach to framework definition is therefore that the language has virtual classes. Most object oriented languages do not have classes within classes, with Java (Arnold &Gosling, 1996) and BETA as exceptions.

Component based software development, with solutions based on e.g. CORBA, COM or Java Beans, represents another approach to the definition of frameworks. While the approach described above relies on access to the source so that specialisation can be used to express adaptation to specific needs, component based frameworks relies on applications with so-called "hot-spots", that is components that can be exchanged with components with the same interface.

4.2 Frameworks in The Integrated Method - TIMe

TIMe is a comprehensive development methodology that uses UML , MSC and SDL as its primary languages for analysis and design. It emphasises that SDL is used to make models that are both readable, formal and sufficiently complete for extensive simulation and automatic code generation. TIME recommends that design is carried out in three main steps: application design, architecture design and framework design, where application design and framework design are abstract models expressed primarily in

SDL and MSC (possibly using UML for parts where SDL is not well suited).

4.2.1 Application design: where the service functionality is designed

The first purpose of an application design model is to describe the system behaviour at an abstraction level, where it can be understood and analysed independently of a particular implementation.

The second purpose is to be a firm foundation for designing an optimum implementation satisfying both the functional and non-functional requirements.

4.2.2 Architecture Design: where the implementation architecture is decided

The purpose of architecture design is to design an implementation architecture that will behave as defined in the application design model and satisfy the non-functional properties. The purpose of architecture design is to answer *how* the system is going to be realised. This is expressed using Architecture descriptions that show:

- the overall architecture of hardware and software;
- how frameworks and applications are mapped to the architecture.

While the application and the framework has focus on functional properties and behaviour, the architecture has focus on non-functional properties and physical structures. The purpose is to give a unified overview over the implementation and to document the major implementation design decisions.

Architecture design determines critical architectural issues such as physical distribution, global addressing schemes and fault handling. Some of these may subsequently be reflected in the framework model in order to describe the complete system behaviour.

During normal application evolution, the architecture will be stable, and system evolution can take place mainly at the Application level.

In an initial development, architecture design will come before framework/infrastructure design.

4.2.3 Framework Design: from Infrastructure to Framework

The purpose of framework design is to describe the complete system behaviour taking the underlying implementation as defined in the architecture design into account. In this step the implementation dependent infrastructure functionality is taken into account, e.g. distribution support, error handling and configuration. The infrastructure part of a system contains additional behaviour needed to fully understand what the system does (i.e. the complete system behaviour). Here we find objects and parts of objects that support distribution, system administration and other facilities not directly related to user services (applications). Whenever practical, the application and the infrastructure should be put together in a framework that serves to simplify the definition of new systems and separate the evolution of applications from the evolution of infrastructures. The framework model defines a system type or a block type, with predefined structure so that a specific application system only has to provide the specific "contents" of part of this structure using the approaches explained earlier.

An application system designed before the infrastructure was known,

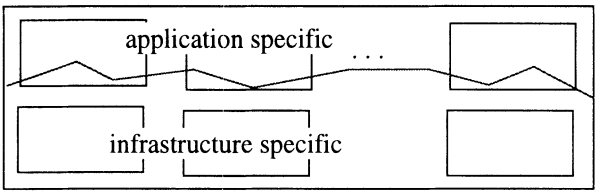

Figure 11: A framework with application and infrastructure specific parts of systems

may have to be redesigned somewhat in order to satisfy the infrastructure interfaces. Restructuring does not mean that everything has to be redefined, only those parts that depend on the infrastructure. When the infrastructure is known, new application types may be defined right away to comply with the infrastructure interfaces.

In general it will be an advantage if the application design has been done by means of types that are as general as possible.

When the framework has been defined, making a specific system instance is a matter of exchanging the application types of the framework with either improved versions or new application types with e.g. new functionality.

5 CONCLUSIONS

The notion of a framework has a potential to be very useful when making abstract system models using FDTs, especially when the models are complete and used as basis for system evolution and code generation. This paper has only considered the possibility of making frameworks using SDL. However, the underlying needs that are addressed and the potential benefits of using frameworks are independent of the FDT actually used and by no means restricted to SDL. The important thing is that the language of the FDT is able to support frameworks in an efficient way. As has been demonstrated in this paper, SDL provide some basic features that make frameworks possible. Using these, a divide-and-combine approach to application and infrastructure development is possible that enable a high degree of reuse, and simplify system evolution.

6 REFERENCES

Steve Sparks, Kevin Benner, Chris Faris; *Managing Object-Oriented Framework Reuse;* IEEE Computer, September 1996.

Mohammed E. Fayad, Douglas C. Schmidt; *Object-Oriented Application Frameworks*; Communications of the ACM, October 1997.

Z.100; *CCITT Specification and Description Language (SDL)*, ITU-T, June 1994

ITU (1993d) Z.120 Message Sequence Charts (MSC), ITU-T, September 1994, 36 p ("MSC-92")

Rolv Bræk and Øystein Haugen (1993); *Engineering Real Time Systems*. Hemel Hempstead: Prentice Hall, 1993.

TIMe - The Integrated Method. SINTEF Telecom and informatics 1997, see http://www.sintef.no/time.

Booch, G., Jacobsen, I. and Rumbaugh, J. (1997b). *The Unified Modeling Language*, Version 1.1, Rational Software Corporation, http://www.rational.com (September 1997)

Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard; *Object Oriented Programming in the BETA Programming Language*, Addidon-Wesley 1993

Ole Lehrmann Madsen and Birger Møller-Pedersen; *Virtual Classes - A powerful Mechanism in Object-Oriented Programming*, OOPSLA 1989.

Ellen Agerbo and Aino Cornils: *How to preserve benefits of Design Patterns*, OOPSLA 1998.

K. Arnold and J. Gosling: *The Java Programming Language*, Addison-Wesley 1996