Supporting evolution of SDL-based systems: industrial experience

B. Gulla
Norwegian Institute of Technology
N-7034 Trondheim, Norway
Tel +47 73594479, Fax +47 73594466
bjorngu@idt.unit.no

J. Gorman
SINTEF DELAB
N-7034 Trondheim, Norway
Tel +47 73597085, Fax +47 73532586
gorman@delab.sintef.no

Abstract

Stentofon, a Norwegian company producing customized internal communication systems, uses SDL and automatic code generation for software development. A crucial issue for the success of the approach in this industrial context turns out to be efficient handling of system variants and evolution.

The paper starts with a description of the approach to software development currently used at Stentofon, and highlights why evolution support is an important problem area in their product development. The paper then describes the PROTEUS* approach to supporting system evolution, and assesses the extent to which this has been successful in its practical application to SDL-based product development at Stentofon. The paper concludes with a summary of the lessons learned from this work.

Keywords

SDL, industrialization, FDT tool support, system evolution, configuration language

1 INTRODUCTION

This paper deals with practical issues concerning the use of SDL in industrial software development. It reports on an approach used by Stentofon, a Norwegian company producing customized internal communications systems, and on their experiences using this approach.

The underlying concepts of SDL (processes, defined as finite state machines, communicating via signals) are well-suited to the kind of software required by Stentofon in their products. The level of abstraction provided by SDL allows designers to work on application related concepts rather than on coding details. In addition, by using automated code generation, it is possible to produce software satisfying the constraints on time and space efficiency required in industrial software. This enables rapid production of reliable, efficient software. It is for these reasons that Stentofon chose to use SDL as the basis for software development in their prod-

* PROTEUS is project no. 6086 in the European research programme ESPRIT. The goal of the project is to provide methodological and tool support for system evolution.

G. V. Bochmann et al. (eds.), *Formal Description Techniques VIII*
© IFIP International Federation for Information Processing 1996

ucts.

There are, however, some important practical issues which need to addressed to ensure that SDL can be used effectively in an industrial setting. Discussions about SDL (and other notations for describing software) are almost always centred round describing the behaviour of "the system". But for a company such as Stentofon, business centres round a whole *family* of systems. All members of the system family need to be maintained and developed, and there is substantial sharing of components amongst them. On the other hand, each member of the system family is in some way unique, filling the specific needs of a particular execution platform or customer. The family must also *evolve* over time, adding new members to take account of new requirements as they arise. Effective product development requires some infrastructure for dealing with the family as a unit *and* taking care of the details of individual members: support is needed for handling versions and variants of components, and for handling interdependencies amongst them.

Another issue is that of code generation. It is not enough that code generation is automatic and fast; it must also be *customizable*. This is necessary in order to be able to generate optimal code according to the particular constraints of different members of the system family. Thus, a customizable code generator is also part of the required infrastructure.

Not all parts of a system are amenable to description using SDL. From a purist point of view, it would be best to describe all parts of a system using SDL - even parts where SDL was not the most natural notation to use. But it would be unwise to do so in practice: it would lead to "artificial" SDL descriptions, less effective code and dissatisfied designers. It is far better to adopt a compromise, and use traditional programming languages (such as C) - or even assembly code - where this is most appropriate. This is the approach adopted by Stentofon, and the positive effect of it has been to *facilitate* acceptance for using SDL.

This paper describes Stentofon's approach to these practical software engineering issues. This involves the use of ProgGen (a customizable code generator), and PCL (a configuration language developed in ESPRIT project PROTEUS). PCL is used to describe system models which include information on all sources of variability within a system. By using the associated toolset, system models expressed in PCL can be used as a basis for system generation. PCL provides a more formalised approach to system building than that provided in commonly used tools such as *Make*.

In order to demonstrate the practical applicability of the PROTEUS approach with SDL, Stentofon's intercom product family was used as a "trial application" in the project. The paper starts by describing this industrial context (Section 2) and by listing the specific areas where support was needed (Section 3). This is followed by a description of the PROTEUS approach (Section 4), and a discussion of how this was integrated with existing practices at Stentofon (Section 5). Finally, Section 6 provides an assessment of how successful the approach has proven to be in its practical application at Stentofon.

2 INDUSTRIAL CONTEXT: THE STENTOFON INTERCOM PRODUCT

2.1 Stentofon

Stentofon is an industrial company based in Trondheim, Norway. Their principal business is the production of customized internal intercom systems. They design and supply both the hard-

ware and the software elements of these systems, and are a market leader in this area.

Stentofon has been in this business for many years, and has a significant share of the world market for this type of product. As a result, they have a large number of installations around the world - and many of these are different variants of the intercom product. Good hardware reliability also means that many older systems are still in use. In addition, it has always been - and continues to be - an important element of company strategy that Stentofon provides *customized* solutions for the special needs of individual customers. Combined with the large number of installations, this means that the company is responsible for the maintenance and further development of a very large variety of systems.

The large variety of systems produced by Stentofon, combined with the fact that new product development is always necessary to maintain a competitive edge, means that support for evolving systems is very important at Stentofon. Although evolution applies to both the hardware and software parts of the product, the issues discussed in this paper will be restricted to the software components.

2.2 Tool chain for software development

The tool chain used at Stentofon is summarised in Figure 1. It is mainly based on use of the design language SDL (ITU, 1993), supported by a methodology (Bræk and Haugen, 1993) developed in the SISU* programme. Software development is carried out on a host environment consisting of a network of workstations; code developed on that platform is then re-generated for various target platforms consisting of the hardware systems developed by Stentofon.

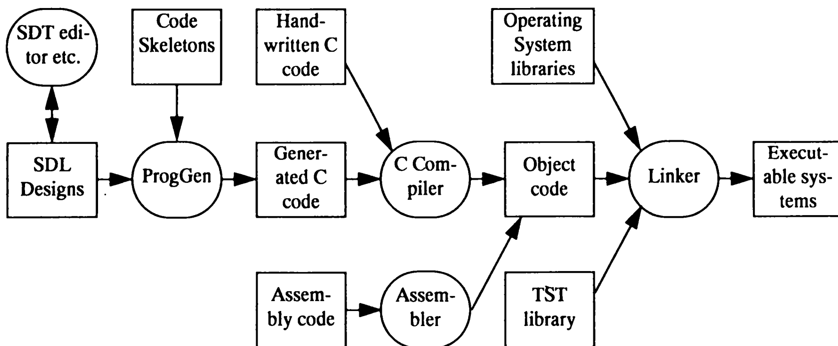


Figure 1 SDL tool chain used at Stentofon.

SDL is a high-level specification language with a graphical syntax which facilitates communication amongst system designers and between them and the customer. It is highly suitable for expressing the functional design of real-time software systems of the type developed by Stentofon, and provides a clear picture of system structure and behaviour. For editing and analysing SDL descriptions, Stentofon use the SDT toolset (Telelogic, 1995). The support offered by this toolset is satisfactory, but there is very limited support for handling variants of systems.

SDL is not a 'programming' language, and there are many different strategies one might

* SISU is a Norwegian national research and development programme (1988-1996). Its main goal is to increase the effectiveness of software production for the real-time software industry in Norway.

choose to implement designs expressed in SDL (Gorman and Johansen, 1991). There is no single, obvious strategy for producing implementations from SDL descriptions. Nevertheless - to support evolution - it is best to work at as high an abstraction level as possible, without being too concerned about implementation strategies. The Stentofon approach to this problem is to *transform* SDL designs to C code, in such a flexible way that the system designer can describe/select the transformations that are appropriate as the system evolves. For this purpose, the ProgGen tool (Floch, 1995) from SINTEF DELAB is used. ProgGen has two inputs: the SDL description to be transformed and a set of *code skeletons* which allow the designer to describe the exact strategy for transformation to C to be used in a particular case. By using this automatic but flexible code generation, the advantages of working at the SDL level can be maintained, while still maintaining flexibility at the programming language level. Using this approach, about 75% of the system functionality in the Stentofon systems is maintained at the SDL level. However, various practical considerations mean that some parts of the software have to be directly coded in C or (in a few cases) in assembler.

SDL is based on the concept of finite state machines (FSMs) which execute in parallel, and communicate by exchanging signals. Thus, efficient implementations of SDL systems are usually based on use of an SDL *run-time system* providing low-level implementation support for the high-level SDL concepts. Stentofon uses the TST product (Telox, 1995) for this purpose. TST provides an SDL run-time system consisting of a library of support routines, together with extensive test facilities. In order to use TST, it is necessary to provide a set of *configuration files* which include information like which SDL processes will execute on which physical processors.

2.3 Variability

The tool chain described above has proven to be effective at Stentofon, and the advantages of working as much as possible at the SDL level are appreciated by all members of the software development team. However, there are many different types of 'source code' that have to be carefully managed:

- SDL designs
- ProgGen skeletons for describing the process of C code generation
- C code and Assembly code, hand-written
- TST configuration description files
- declaration of the system building process, currently represented by a set of shell script files and makefiles
- 'foreign' libraries, e.g. TST, AMX etc.

As the system evolves, variants of all of these item types are produced, and dependencies between them also increase in complexity. To make effective use of the various tools of the development platform, it is vital that support be provided to control these sources of variability.

3 SUPPORT NEEDED BY STENTOFON

This section describes the problems concerning evolution that Stentofon identified *before* the

PROTEUS project started. In each sub-section, we describe the nature of the problem, the requirements for improved support and 'Today's approach' i.e. how the problems are currently being handled *without* support from PROTEUS. In Section 5 we assess the extent to which these problems have been solved or alleviated by using the PROTEUS approach.

3.1 General requirements

Stentofon require support for two main **types** of system evolution: evolution over *time* (corrections, product enhancements, adaptations for new platforms etc.) and evolution arising from *customizations* carried out for specific customers. The support provided must handle *revisions and variants* of all the item types involved in the tool chain described in Section 2.2 on page 3. In particular, it has to be able to deal with:

- stable system versions: systems already delivered to customers
- versions under development (including test and debug variants)
- variants for different platforms (host/target)
- variants produced to fulfil the needs of specific customers.

A basic requirement is to provide *visibility* of the overall system structure, clearly indicating which parts of the system are common and which vary. For variants it must be possible to express that the choice for one item might depend on the choice of variant for some other items.

Another key issue is support for the *system building* process. Since this process is to a large extent determined by the system structure, consistency between these must be ensured. The build process should:

- be completely automated, through tool support
- result in executable code that is at least as time and space efficient as that previously produced using a less automated generation process
- be reliable: *all* necessary re-compilations etc. needed for a new variant must be carried out using the correct versions of files, compiler flags etc.
- be fast: unnecessary build operations must be avoided.

Today's approach: Existing support for evolution falls far short of the requirements described above. There is no globally documented, detailed overview of existing system variants, so 'visibility' is poor. RCS (Tichy, 1985) is used for version control of the various types of source files, and Make (Feldman, 1979) is used for partial automation of the system building process.

3.2 SDL composition structure

Systems described in SDL are essentially built up of *processes* and *procedures* which are composed in a hierarchical *block* structure. In developing evolving systems, Stentofon often wish to reuse particular processes, procedures and block hierarchies; these elements need to be composed to produce a given system variant. However, the version of the SDL language used by Stentofon lacks support for modularisation*, and composition must be carried out using vari-

* Stentofon currently use SDL/88. Section 6.1 discusses possible consequences of converting to SDL/92.

ous ad hoc facilities provided by the SDT editor tool.

This composition problem is particularly important because the SDL level is the 'top' level of the system, and many other components used to generate an executable system are dependent on the SDL composition structure.

Stentofon require tool support which facilitates composition of SDL processes etc. into complete SDL systems. Systems composed in this way must then be able to be used as the basis for the automated system generation process, in full confidence that all dependencies with other item types (C code, ProgGen skeletons etc.) will be taken into account.

Today's approach: SDL composition is carried out using the SDL editor provided in the SDT tool; this involves a considerable degree of user interaction, and makes it very difficult to see which variants of which SDL components are composed together in different system variants.

3.3 Complexity of system generation

The system building process used at Stentofon involves many complex dependencies between the various item types that are involved in the process. These have to be taken into account when deciding which tool invocations are necessary, as do other sources of variability such as the correct compiler flags etc. that are needed when generating a particular system instance.

The process is made more complex because it is necessary to translate from SDL to C: there is a large number of ProgGen skeletons (several hundred), and it is essential that the skeletons used during system generation are appropriate for the system variant being produced. Skeletons must be selected according to which translation strategies are required to fulfil the functional and non-functional requirements of a particular delivery.

Given the general requirement that system generation should be completely automated, tool support is required which takes account of these complexities and ensures consistent choices.

Today's approach: System building is performed using a mixture of makefiles and shell scripts. The 'implicit' rules provided by Make have proven not to be sufficiently powerful to handle the complexities of the system generation process, and a considerable amount of manual maintenance of makefiles is necessary. This is both time-consuming and potentially error prone. It has also led to a situation where the flexibility offered by the Proggen skeleton mechanism has not been fully exploited because of the complexities of selection of skeletons during system generation.

3.4 Run-time system and operating system considerations

Some important sources of variability are dealt with at the run-time support and operating system level. For particular system variants, customizations are needed in order to make most effective use of the basic hardware/software platform on which the system will run.

A case in point is the *distribution* of the software: which SDL processes will run on which physical processors? This is dealt with by producing appropriate low-level *configuration* files for the TST run-time support system.

Tool support is required which will remove the need to maintain such low level files, and allow such considerations as software distribution to be dealt with at a higher abstraction level.

Today's approach: No tool support: all dependencies are dealt with by manual procedures.

4 THE PROTEUS APPROACH

The objective of the PROTEUS project is to provide support for system evolution. The project is developing methods and tools for (1) domain analysis, (2) adapting existing design methods (SDL, HOOD, MD) to support evolving systems, and (3) modelling system structure and software system building. Stentofon participates as an application company in PROTEUS.

4.1 PROTEUS Configuration Language (PCL)

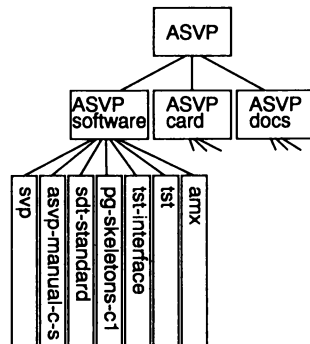
PCL (PROTEUS consortium, 1994) is a formalism for system modelling and software system building. As systems evolve, large numbers of system and component versions with slightly different properties are created. A *system model* is a description of the items in a system and the relationships between them. Such a model uniquely identifies the comprising components, their properties and structure, and tracks their evolution. Variability should be represented, making it clear what is common and what differs between system variants. Its purpose is to capture knowledge about a system and its domain in an understandable and concise manner.

The system model in PCL is based on the *family* notion. A family represents a logical entity which may occur in different variations in particular systems. The family description encompasses all potential variability of the entity. A specific member (version) of the family is determined by removing ambiguity in the family description. We call this operation *binding*, and it is one of the core functions of the tool set supporting PCL usage.

4.2 System modelling facilities

In PCL the family construct is used for modelling the logical entities in the system. A model is organized as a layered composition structure at the logical level. Entities may be part of other entities, and may also have sub-components. In the parts section the logical composition structure is declared. The following example shows ASVP* – one of the sub-systems in the Stentofon application.

```
family ASVP
   parts
      software =>       ASVP-software;
      hardware =>       ASVP-card;
      documents =>      ASVP-docs;
   end
end
family ASVP-software
   parts
      sdl =>            svp;
      manual =>         asvp-manual-c-s;
      sdl-support =>    sdt-standard;
      sdl-trans =>      pg-skeletons-c1;
      run-time =>       (tst-interface, tst);
      oper-sys =>       amx;
   end
end
```



* ASVP is an abbreviation for AICE Stored Voice Playback.

In the example above, the ASVP entity is composed of ASVP-software, ASVP-card and ASVP-docs. The decomposition of ASVP-software is further indicated. Parts may of course be shared among subsystems, relationships which must be kept track of during change impact analysis.

A family may represent any kind of entity: hardware objects, software artifacts or even combinations. PCL facilitates multi-dimensional classification in user-defined term spaces. There is also a relationships section for declaring other kinds of relationships between entities and a general relation definition facility.

Attributes are used to characterize a family and its potential variability. Attributes are typed and may be of type integer, string, or a user-defined enumeration. There are two kinds of attributes, information attributes and variability control attributes. Information attributes state properties common to all members of the family. They are declared by using the '=' assignment operator.

```
family hwif-manual-c
   attributes
      language: string = "C";
   end
end
```

Variability control attributes indicate possible variability among the members of a family. Default values may be assigned to such attributes with the ':=' operator, but these may be overridden at binding time.

```
family ASVP-software
   attributes
      target : target_type;
      speed : speed_type := fast;
   end
end
```

Particular members of the family are identified by determining values for all variability control attributes. For example, if binding attribute target to emulator-stripped and speed to fast, a unique member of ASVP-software is established. Each variability control attribute defines a dimension of variability as illustrated in Figure 2. Variability control attributes can in principle be selected independently, although there might be some disallowed combinations.

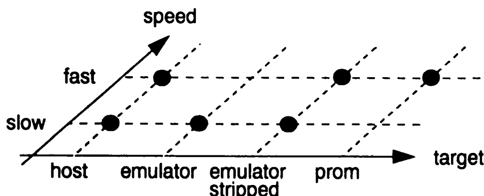


Figure 2 Two dimensions of variability and some possible family members.

Note that attributes denote *conceptual* variability – they do not say anything about how that variability is realized. PCL covers a wide range of types of variability, for example structural variability, version selection of associated physical objects, and differences in processing tool parameters. The example below illustrates *structural* variability, i.e. a family whose members have different composition structures. This is expressed by embedding if-then-else phrases in the description. Such phrases use expressions based on attribute values. Alternatives can also be viewed graphically as illustrated in Figure 3.

```
family ASVP
   attributes
      target : target_type;
   end
   parts
      software =>  ASVP-software;
      hardware =>  if target = host then
                       Workstation
                   else ASVP-card
                   endif;
      documents =>ASVP-docs;
   end
end
family ASVP-software
   attributes
      target : target_type;
      speed : speed_type;
   end
   parts
      sdl =>        svp;
      manual =>     asvp-manual-c-s;
      sdl-support =>sdt-standard;
      sdl-trans =>  if speed = fast then pg-skeletons-c1
                    else pg-skeletons-c2
                    endif;
      run-time =>   (tst-interface, tst);
      oper-sys =>   if target <> host then amx endif;
   end
end
```

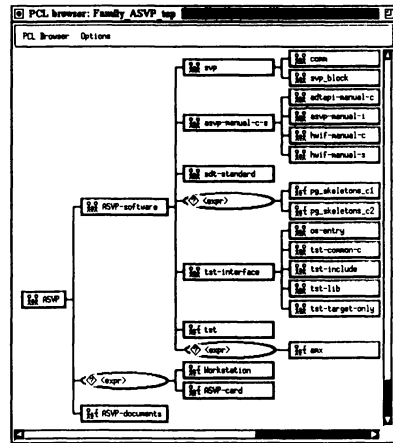


Figure 3 PCL Browser view of top levels of ASVP.

A set of *physical objects* may be associated with a family. Physical objects is the PCL term for tangible objects existing in the real world and for software artefacts making up a system. They are called physical objects to distinguish them from the logical notions which may exist only within the PCL model.

```
family AICE_COMM
   attributes
      IMPL_DIR : string;
      manual : boolean := true;
   end
   physical
      p1 =>    "aice_comm.spr";
      p2 =>    "AICE_COMM_def.h" attributes workspace := IMPL_DIR; end;
      p3 =>    if manual = true then
                   "AICE_COMM_proc.c" attributes workspace := IMPL_DIR; end
               endif;
   end
end
```

Three physical objects named aice_comm.spr, AICE_COMM_def.h and AICE_COMM_-proc.c are associated with the AICE_COMM entity representing an SDL process. Attributes and classifications may be declared for physical objects just as for families. All physical objects here are software objects, i.e. files, which is the default unless classified otherwise. The workspace attribute is a special attribute recognized by the PCL tools which indicates where in the file system this file should reside.

In descriptions of large systems, it often happens that several family descriptions share a similar structure and content. In Stentofon's ASVP system, for example, one family description is required for each SDL process in the system. It would be tedious to have to write (basically) the same PCL description once for each SDL process; it would also be difficult to maintain all the descriptions. To avoid such problems, PCL provides a comprehensive *inheritance* mechanism. Thus, for Stentofon's ASVP system, one can capture the *common* information in an sdl-process family, and highlight what is *unique* to each individual process in separate family descriptions all of which inherit from sdl-process.

As mentioned previously, a particular member of a family is determined by assigning values to the variability control attributes. Such attribute assignments are specified in a *version descriptor*. These will override the default expressions occurring in the family model.

```
version v-ASVP-host of ASVP-software
   attributes
      target := host;
      speed := slow;
      HOME := "/users/arvid/";
   end
end
```

During binding the version descriptor is applied to the family and sub-families recursively, producing a bound family hierarchy. If a version descriptor is incomplete (i.e. a value is not specified for all variability control attributes), a partially bound family will result. This corresponds to a partially bound configuration.

To handle the problem of file versions, we have chosen a two-tier repository approach. The contents and the descriptions of file versions are managed by a special tool called the Repository. Otherwise, descriptions of file versions would soon make the system model impractically voluminous and hard to use.

Version selection is the process of determining a consistent set of versions for elements of a system. PCL supports intensional version selection (Belkhatir and Estublier, 1986). Based on a query stating desired properties, the best matching version (if any) for each file is computed by the Repository by inspecting the available versions and their characteristics. This process is controlled by the PCL system model, since the submitted query is constructed by including certain attributes from the PCL model. Details of this are outside the scope of this paper.

4.3 System building support

For modelling software system building processes, PCL includes the tool construct. It allows declarative specifications of steps in the building process by defining the signature and behaviour of software tools. A C compiler may be modelled as follows:

```
tool cc
   attributes
      CC : string default "cc ";
      CFLAGS : string default " ";
      CINCL : string default " ";
   end
   inputs in => c-source; end
   outputs out =>obj-68k; end
   scripts
      build :=   CC ++ CFLAGS ++ CINCL ++ "-o " ++ out ++ " -c " ++ in;
   end
```

end

The input and output sections specify that this derivation step template transforms a file classified as c-source into a obj-68k file. The build script specifies the actual command line for tool invocation. It is also possible to describe more complex tools, with multiple inputs and outputs.

System building support in PROTEUS is currently realized by generating standard makefiles, which are interpreted by the Make program (Feldman, 1979) for actual system (re)generation. Figure 4 shows a part of the derivation graph for an ASVP-software variant. The generated makefiles may optionally include check-out rukes of correct versions of files.

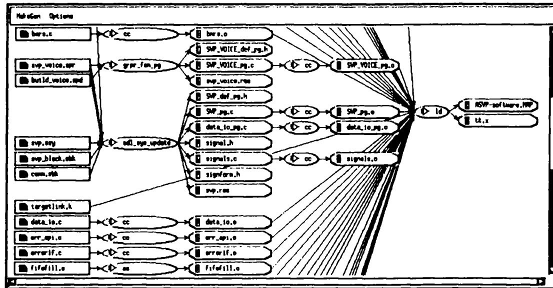


Figure 4 Derivation graph produced during Makefile generation.

4.4 Tool support

A comprehensive toolset to support the creation and use of PCL models has been developed. Figure 5 presents an overview of the core PCL tool set.

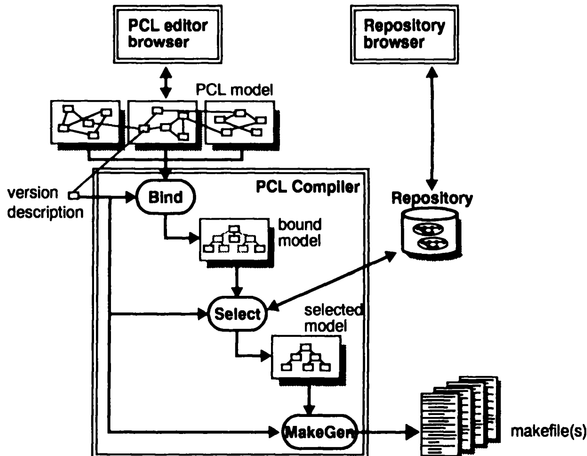


Figure 5 Tool overview.

- *PCL compiler* is an interactive tool for management and analysis of PCL models. Primarily it supports the three fundamental operations for PCL models, namely binding, version selec-

tion, and makefile generation. Partial and interactive binding are both supported. Some other functionality is available as well, e.g. parsing and unparsing of textual PCL descriptions, check-in and check-out of software subsystems to/from the Repository.

- *PCL editor/browser* is a graphical structural editor for entering and browsing PCL models.
- The *Repository* manages the contents and descriptions (attribute annotations) of versions of software objects.
- *Repository browser* is a graphical browser for inspecting and manipulating the contents of the Repository.
- The *PCL reverse* tool allows automatic construction of rudimentary PCL models for existing software systems. It also includes features to perform consistency checks between the workspace, the Repository and a PCL model for the software parts of a system.

5 THE PROTEUS APPROACH APPLIED IN STENTOFON

In this section we discuss how the PROTEUS approach may be integrated into Stentofon's development process. The discussion is organized according to the requirements for evolution support listed in Section 3.

5.1 General requirements

Overall system *visibility* is provided by the PCL system model. Descriptions of all relevant parts and their inter-dependencies are possible, for e.g. SDL parts, hand-coded software, documentation, hardware elements. Elements are classified in a company-defined schema. The system building process is controlled from the same descriptions, ensuring consistency between the model and the actually generated system.

The PCL system model allows representation of all potential variability in a system, highlighting which parts are common and which differ between variants. In Stentofon a number of different mechanisms for achieving software variability are used, e.g.:

- varying system composition
- component selection, i.e. different files
- component version selection (revisions and branches)
- conditional text inclusion (*conditional compilation*)
- tool selection and ordering of invocation
- tool parameters/switches, e.g. for selection of code generation skeletons, target platform, debug support, optimization
- configuration files for the run-time system.

PCL allows expressing and controlling variability of all these kinds with a single mechanism, namely attributes. Dependencies between variability selections may also be expressed.

5.2 SDL composition structure

The SDL decomposition structure is reflected in the PCL model. Each SDL system, block, process and procedure is described by a PCL family construct, which includes references to the

SDL source and associated handwritten-code (if any). Initially this PCL fragment is created by running ProgGen on the initial SDL model with an appropriate set of skeletons. After subsequent *structural* changes at the SDL level, these must be incorporated into the system model by a manual merge step using the PCL editor. This task typically involves inserting conditional expressions over attributes and possibly introducing new attributes. For structural changes in the hand-written code, i.e. creation or removal of files, a similar procedure is followed, but now PCL reverse may be used to produce the PCL fragment to merge into the existing model.

To generate a particular system instance, a user states the desired properties related to functional characteristics, non-functional aspects, implementation choices etc. in the form of a version descriptor. During the instantiation process the best matching SDL component versions are selected, possibly forming an original combination of versions. The SDT tool (Telelogic, 1995) reports any inconsistencies, and allows manual editing to resolve conflicts if necessary.

5.3 Complexity of system generation

PCL allows all knowledge related to the system building process to be stated in one formalism and be represented in one place. Complex derivation steps far exceeding the expressiveness of implicit Make rules (Feldman, 1985) are handled. Manual maintenance of makefiles and shell scripts is avoided by automatically generating appropriate makefiles for each particular system instance.

To ease maintenance of the system model itself, *derived* relationships are supported in PCL. Dependencies which are possible to derive from files, e.g. #include dependencies between C source code files, need not be represented in the system model. By rather declaring depend scripts for tool entities, such dependencies may be automatically extracted during the system building process.

Variability in the SDL-to-C transformation is solved by including a model of the ProgGen skeleton set as a subsystem of the system model. Skeleton files are versioned and managed by the Repository. Variability in this subsystem is then characterized and disambiguated just as any other variability in the system model.

5.4 Run-time system and operating system considerations

The run-time organization of an application is possible to model in PCL. PCL includes predefined relations which are taken into account during system analysis, allowing a high-level declaration of distribution aspects and usage of the underlying platform. Since hardware elements are already included in the model, this fits naturally into the framework.

However, the ultimate goal of automatically generating appropriate TST low-level configuration files from such a PCL model has not yet been achieved. We hope to be able to express this generation step using the general tool construct, but some technical problems still remain.

6 EVALUATION

This section reports on experiences from applying the PROTEUS approach at Stentofon. Table 1 shows the number of files, total number of lines of code, and number of versions per. file. Totally there were 721 manually maintained files and over 8000 versions of these. Note

that all *generated* files (C files generated from SDL, Make dependency files, etc.) are omitted.

Table 1 Size metrics for the Stentofon AICE application

Type of source	*No. of files*	*LOC*	*Versions pr. file*
SDL	141	181421[a]	5-300
C	238	93500	10-100
Assembly	14	5266	10-100
Configuration files	11	918	5-50
ProgGen skeletons	288	3481	2-5
Foreign libraries	2		3-5
Makefiles, shell scripts	27	17224[b]	10-100

a. LOC for SDL source is approximated by the number of bytes in GR files divided by 30.
b. Five of these makefiles (14.263 LOC) are for recreation of old configurations.

Table 2 gives the size of the PCL model of the AICE application. When using the PROTEUS approach 'Makefiles/shell scripts' and possibly 'Configuration files' in Table 1 can be eliminated, since they will be generated automatically from the PCL model.

Table 2 PCL model

Type of source	*No. of files*	*LOC*
Generic[a]	3	500
System specific[b]	7	1700

a. Common to all SDL-based systems in Stentofon
b. Of this ca. 75% was initially generated by ProgGen.

The evaluation has been based on three main criteria: (1) support for evolutionary development, (2) support for design-oriented devleopment, and (3) flexibility of approach. Among the more advantageous experiences we can mention:

- PCL allows a number of tasks to be done at higher abstraction level than before. Many highly inter-dependent details around the system and in the associated system buidling process are now automatically taken care of based on intensional specifications.
- Constructing a *total* system model covering all aspects of a system has been a favourable experience. Knowledge previously distributed and unavailable (person dependent) is now made visible and represented in a formal model. This information is also useful e.g. for internal communication and training. Modelling of system variability is however still somewhat incomplete.
- The choice of building the Repository on top of an existing version management system has been quite fortunate at Stentofon. Several years of evolution history are instantly available, presented in a graphical user interface.
- 'High' expressiveness of PCL is a vital factor for the success of PCL. It allows concise mod-

els, which are easy to understand and maintain as the system evolves. Intensional mechanisms are essential to handle the size and complexity of real systems.

Problem areas and deficiencies observed are:

- A crucial problem is to ensure consistency between the system model and the actual system. Structural and even some non-structural changes require that the system model is updated. More tight integration between the design tool and the PCL model is one possibility, currently being explored by other partners in PROTEUS.
- A more practical obstacle in the experimentation has been the lack of availability, stability and maturity of the PCL toolset. The tools were originally released for a platform not available at Stentofon's development department.
- Fully automatic generation of run-time system configuration files not yet achieved.

6.1 Looking ahead: SDL/92

This evaluation is based on use of the SDL/88 version of SDL, which is the one currently in use at Stentofon. However, as tool support for SDL/92 is now becoming available, Stentofon will almost certainly start using SDL/92 in the near future. While we obviously cannot here report on experiences of using ProgGen and the PCL tools in conjunction with SDL/92, it is worthwhile considering the impact that the change to SDL/92 is likely to have. From a software engineering perspective, SDL/92 offers the following main advantages over SDL/88:

- easier to modularise, using *types* and *packages*
- variants can be described using *inheritance*
- possible to produce more generic descriptions, using *context parameters*

These improvements in the language make it easier to structure system descriptions, and to handle variability as systems evolve. However, we do not believe that these new language facilities are sufficient to provide the support for evolution required in an industrial context (see section 3). In fact, the use of inheritance can actually complicate management of the system descriptions through the extra dependencies that are introduced, and the wider range of translation strategies that become possible.

In conclusion, our expectation is that use of SDL/92 will improve our software development process, but that the support provided by the ProgGen translation tool and the PCL toolset will, in total, be no less beneficial when using SDL/92 than when using SDL/88.

7 CONCLUSIONS

Software is a valuable organizational asset. As organizations find themselves having to respond more and more quickly to business and environmental changes, it is important that the collective expertise and knowledge about a system can be pooled, formalized and recorded. Although formal description techniques as SDL significantly reduces the size of the system representations, the large number of source elements to manage and the complex building processes in industrial systems necessitates method and tool support.

Using PCL to support evolution through the use of complete system models is a promising approach to this problem. Experience at Stentofon shows that it requires a significant effort to build such models, but that the benefits in terms of improved visibility and automation are also substantial.

The initial experience reported here is sufficiently positive that Stentofon intends to continue this evaluation work, and it is quite likely that they will eventually decide to adopt PCL in all their software development projects. The potential *economic* benefit of the evolution support provided by PROTEUS has not been assessed by the work reported here. It is an important issue, and future work based on longer-term evaluations should be planned.

ACKNOWLEDGMENTS

The work described in this paper is carried out in the scope of PROTEUS, project number 6086 in the European Commission ESPRIT research and development programme. The work carried out in Norway has been partly funded by the Norwegian Research Council (NFR). We would like to acknowledge the contributions made by current and previous members of the project, and especially Arvid Strømme of Stentofon, Professor Reidar Conradi of NTH, Jacqueline Floch and Richard Sanders (DELAB) for many valuable comments.

REFERENCES

Belkhatir, N. and Estublier, J. (1986) Experience with a Data Base of Programs. *ACM SIGPLAN Notices*, **22(1)**, 84-91, January 1987.

Bræk, R. and Haugen, Ø. (1993) Engineering Real Time Systems. Prentice Hall, ISBN 0-13-034448-6.

Feldman, S. I. (1979) Make, A Program for Maintaining Computer Programs. *Software - Practice and Experience*, **9(4)**, 255-265, April 1979.

Floch, J. (1995) Supporting evolution and maintenance by using a flexible automatic code generator. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, April 23-30, 1995. IEEE CS Press, pp. 211-219.

Gorman, J. and Johansen, U. (1991) Engineering the Implementation of SDL Specifications. In *Proceedings of the Fifth SDL Forum*, Glasgow, North Holland, ISBN 0 444 88976 0.

ITU (1993) Specification and Description Language SDL. Recommendation Z.100, 237 pp., CCITT ITU, Geneva, Switzerland.

PROTEUS consortium. (1994) PCL-V2 Reference Manual. Technical Report P-DEL-3.4.D-1.9, 85 pp., September 2, 1994.

Telelogic. (1995) SDT User Manual, TeleLOGIC Malmö AB, Box 4128, S-203 13 Malmö, Sweden.

Telox. (1995) TST User Manual, Telox, Norway.

Tryggeseth, E., Gulla, B. and Conradi, R. (1995) Modelling Systems with Variability using the PROTEUS Configuration Language. In *Proceedings of the Fifth International Workshop on Software Configuration Management*, Seattle, WA, April 24-25, 1995, Springer Verlag.

Tichy, W. F. (1985) RCS - A System for Version Control, *Software - Practice and Experience*, **15(7)**, 637-654, July 1985.