Olsen, A. et al. (1999) *The pros and cons of using SDL for creation of distributed services.* Lecture Notes in Computer Science, 1597 . pp. 342-354. ISSN 0302-9743

http://eprints.gla.ac.uk/7236/

Deposited on: 03 September 2009

Enlighten – Research publications by members of the University of Glasgow
http://eprints.gla.ac.uk

The Pros and Cons of Using SDL for Creation of Distributed Services

A. Olsen*, D. Demany*, E. Cardoso*, F. Lodge^{+}, M. Kolberg^{+}, M. Björkander^{+}, R. Sinnott^{+}

*SCREEN project, $^{+}$ TOSCA project

In a competeative market for the creation of complex distributed services, time to market, development cost, maintenance and flexibility are key issues. Optimizing the development process is very much a matter of optimizing the technologies used during service creation.

This paper reports on the experience gained in the Service Creation projects SCREEN and TOSCA on use of the language SDL for efficient service creation.

1. Introduction

The area of Service Creation, while far from new, grows ever more complex and challenging due to the constant increase in requirements placed on it by consumer demands, competition and new technologies. For example, JAVA, CORBA, DCOM ,etc, . present us with exciting new opportunities for software distribution in terms of flexibility, portability and re-use of existing software components. However this, in return, adds to the complexity of the service creation process, and introduces many new factors which need to be considered to ensure that all general requirements on service creation are met. Examples of these requirements include:

- Reductions in time to market,
- Reductions in development and operating costs,
- The need for open solutions to service creation, i.e. methods and processes which are equally applicable in many software environments,
- The facility to prototype services, for the purposes of quality assurance and validation of user requirements,
- Re-use of existing software/components,
- Intelligent distribution of the objects comprising the service, in order to meet non-functional requirements such as performance and fault tolerance.

Classic software engineering methodologies are insufficient to meet these challenges. Therefore, new and innovative service creation processes are required to address these issues in the realms of distributed services, in order to facilitate the full exploitation of the potential of these new technologies.

The ACTS projects TOSCA and SCREEN are both currently working in the area of distributed service creation. SCREEN is primarily focused on defining methodologies which are based on the use of industry standard modeling notation and specification techniques, while TOSCA is focused on the development of a tool-chain to support the rapid (automatic) creation and validation of distributed services.

A fundamental part of addressing many of the requirements stated above is the identification of a language which facilitates both specification and design of a new system, which simplifies the validation process and which allows flexible implementation, so that it can target different environments (e.g. TMN, CORBA) with different real-time requirements. Both SCREEN and TOSCA have been investigating the suitability of using SDL (Specification and Description Language) for these various service creation phases. SCREEN has used SDL for the design, implementation and validation of distributed services, while TOSCA has used SDL for service implementation and validation. Both projects have had much practical experience developing distributed services in SDL and are therefore in a good position to offer a pragmatic analysis of the advantages and disadvantages of using SDL in the various phases of service creation. This paper presents the conclusions of the projects in this area and makes a number of recommendations based on these conclusions.

The format of the paper is as follows: in Section 2, the principle features of SDL are outlined. Section 3 describes the applicability of SDL in service analysis, design, implementation, simulation, validation, testing and deployment, based on the experiences of both TOSCA and SCREEN. Section 4 presents the recommendations of the projects as to when and for what SDL should be used for service creation.

2. Introduction to SDL

SDL (Specification and Description Language) is standardised by ITU (the International Telecommunication Union) for describing event driven systems. It is based on the experience of describing telecommunications systems as communicating state machines in the telecommunications industry. Since 1976, SDL has evolved from an informal drawing technique to a fully fledged object oriented language. Today, it is also utilised in areas that are not related to telecommunications systems (for example embedded systems).

Due to its simple conceptual basis and its choice of graphical or textual representation, SDL has maintained its original informal flavour, even though today SDL is a formal language implying that every concept and construct in the language has a precise meaning and can be executed.

Commercial SDL tools are available today (e.g. [GEODE], [SDT], [CIND]) which support various phases of development, such as combined use with UML for analysis, editing, static checking and simulation for design. Fast prototyping, verification (model checking) and code generation for implementation and combined use with MSC and TTCN for testing are also supported by SDL.

An SDL document defines the behaviour of an SDL system. To actually obtain the behaviour of the system the SDL document is processed, which is equivalent to concurrently executing concurrently a number of processes that exchange signals with each other and with the environment of the system.

A process is the unit of concurrency. It is described as a state machine which can react to signals received from other processes or from the system environment. In the transition to the next state, the process can perform various actions such as send signals, assign to variables, branch, call procedures, create new processes and set timers. Each process instance has an unlimited input queue where signals are stored until the process reaches a state where consumption of a given signal is possible.
As an alternative to defining a process as a state machine, it can be defined as a container for a set of alternating services (i.e. threads) each of which is a state machine taking care of specific signals of the process input signal set. Whether a process is split into services is completely transparent to other processes.

The processes are grouped into blocks. Thus, each block contains either sets of processes or a number of (sub)blocks. Blocks are used for grouping conceptual related processes. Shown below is an example of a block type and a process type.

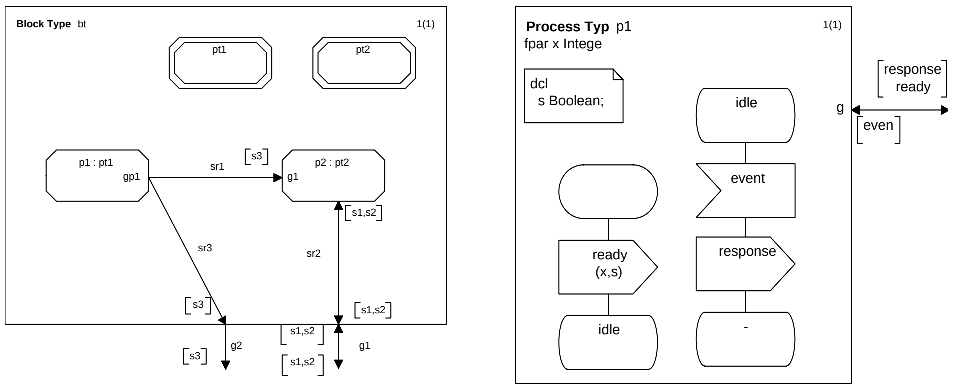

Figure 1: Example of an SDL block and process type.

If a system contains only one block, it is not necessary to specify the system level. Likewise, if that block contains only one process, it is not necessary to specify the block level. Hence, an SDL description may consist of a single process.

To express the communication interface between blocks, processes and services, these are connected by channels. Channels are optional and if they are omitted it means that there are no constrains on who can communicate with who. Therefore, SDL can be used to describe systems with tighly (statically) coupled components and systems with loosely coupled components where a communication link is set up dynamically (e.g. client/server based systems).

Channels convey signals, remote procedures and remote variables. Remote procedures and remote variables are based on signal exchange so the signal is the basic communication primitive. Comparing to CORBA, signals resemble one-way operations, remote procedures resemble two-way operations and remote variables resemble read-only attributes.

Some powerful concepts for object orientation are available for handling types:

- A type may be a *specialization* of another type. Conceptually, the types in a specification are grouped in a *type hierarchy* (i.e. a tree) where the nearest parent node of a type in the hierarchy denotes the *supertype* from which the type has been specialized. All parent nodes of a *subtype* are *supertypes* of the type. Likewise, all child nodes of a type are *subtypes* of the type. Often, the upper layers in the hierarchy are only introduced for the purpose of classifying types, i.e. some types may be 'abstract' implying that it does not make sense to use them for instance creation.

- When a type is specialized, some of its locally defined types may be redefined. The supertype determines which types the subtype may redefine. Those types are called *virtual types*.For types which define state machines, i.e. process types, service types and procedures, also the start transitions and the transitions attached to states, may be redefined by a subtype. The parts of a state machine which a supertype allows its subtypes to redefine, are called *virtual transitions*.
- A *generic type* is an incomplete type in the sense that it may refer to entities which are not bound to a definition until the type is specialized. Such entities are known as *formal context parameters* attached to the type. Inside the type, only those properties of the formal context parameters are known (the *formal constraints*) which are important for using the context parameters. When a type with formal context parameters is used, *actual context parameters* are supplied. These are the identifiers replacing the formal context parameters inside the type. A generic type can thus be used to build several similar types. Below is shown an example of a process type with a procedure context parameter (insert) and a signal context parameter context parameter (get_key)

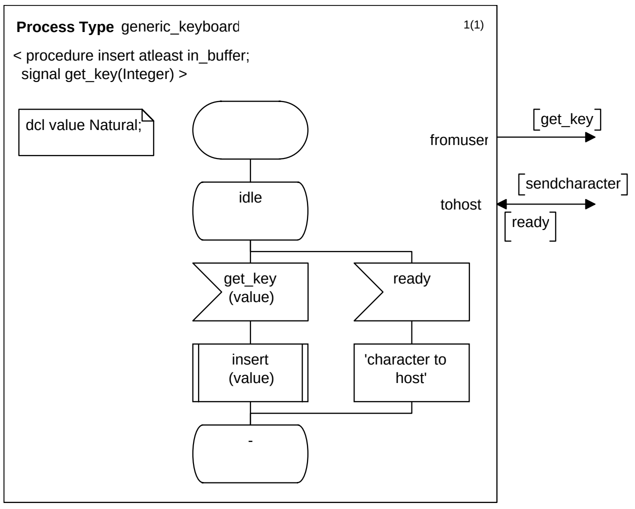

Figure 2: Example of formal context parameters

The data concept of SDL resembles the data concept found in other programming languages since there are expression evaluating to values and values that belong to a certain data type. Also, there are variables which are of a certain data type, used for storing values for later use.

One specific advantage of the SDL data concept is the standardised (in [Z.105]) relation to ASN.1 (a language suitable for description of protocol data).

However, there are two important limitations on the use of data: There is no pointer/reference concept (data is always handled by value) and the data concept does not have full support of object oriented features (e.g. no polymorphism). Therefore, SDL may be less suitable in application areas which have substantial data handling. This problem has been taken up by ITU implying that the data concepts of SDL will be substantially improved in the next version of the standard to be issued in year 2000.

3. Applying SDL to service creation

3.1 Using SDL for Requirements Capture

The purpose of the Requirements Capture phase is to record the service requirements in dialog with the customer or user. During this dialog, it is essential that notations are used which can be easily perceived by non-technical people (such as UML Use Cases) and which focus on the essence of the service as seen from the user. An example would be the user interaction with the service. However, there are two situations where SDL is useful:

1. If there are requirements to the underlying service architecture, SDL may be used informally to sketch the system components.
2. If it is required that a 'works-like' demo is built, SDL may be used to draft a rudimentary service behaviour (a prototype)

If a 'looks-like' demo is sufficient, there is only need for a user-interface tool like Visual Basic or Tcl/Tk.

3.2 Using SDL for Analysis

The purpose of the service analysis phase is to represent the information obtained from the consolidated requirements in such a way that a software design can be directly derived. This means creating a high-level but application-oriented information and dynamic model of the new service. Service Analysis develops models in OMT/UML according to the *Information Viewpoint* of ODP.

OMT include features for making both static object models and dynamic interaction models. But only the static object model is utilised with SDL because MSCs (Message Sequence Charts, another ITU standard) usually is part of the SDL tool and they are well suited for supporting the dynamic model.

UML enhances the OMT static object model by adding useful concepts such as *interface* and *dependency* and making more clear some aspects such as the multiplicity of roles in associations between more than two classes. The dependency relationship is the right concept for specifying dependencies between services or between the service under creation and low level services such as CORBA services.

MSCs are well suited for specifying dynamic properties during the service analysis phase. Traditionally, MSCs have the advantage of being simple to use and understand. Basically, an MSC represents an exchange of messages between object (or SDL process) instances.

3.3 Using SDL for Service Design

The purpose of the Service Design is to provide a complete and non-ambiguous definition of the component/service, taking into account architecture and performance constraints, that can easily be transformed into code (semi-)automatically. Service design develops models according to the *Computational Viewpoint* of ODP.

During the service design, relevant UML modelling items such as classes, associations and roles are translated to corresponding SDL items such as block types, process types, channels and signal routes. OMT/UML tools provide mechanism such as copy/paste from an object diagram to a SDL interconnection diagram in order to facilitate the translation; however an automated translation of an entire diagram is not very helpful because each class and association of the diagram may be translated in a different way. Considering UML, the following table gives an overview of the mapping possibilities from UML to SDL.

UML $\longrightarrow$	**SDL**
Package	block type
DataType	ADT (syntype)
Class	ADT (newtype)
Class { active }	process type
Class { complex }	block type
Association	channel, signal route
Interface	process type
Operation of a class	remote procedure
Operation of an interface	virtual exported procedure
Operation { oneway }	signal
Exception raised by operation	–

Table 1: Translation from UML to SDL.

The left part of the table refers to an UML model element, sometimes specialised by a property (or tagged value) which is displayed inside a pair of braces ({}). These properties are boolean properties with implicit *true* value. For example, *active* is a predefined tag which, applied to a class, means it has its own thread of control.

Active objects (classes with *active* property) are mapped to process types while passive objects (classes without property) are mapped to data types.

Some UML concepts have no corresponding concept in SDL96. For example, translating an UML interface to an SDL process type amounts to adding an *interface class* to the object diagram. SDL2000 is expected to fulfil these missings by introducing an interface concept and exception handling.

3.4 Implementation Strategy: Using SDL as a Behaviour Language for IDL/ODL

In the ODP Reference Model the requirements on a language used for computational specification of a system are defined. The Object Definition Language (ODL) is a language for specifying computational objects according to these rules. ODL is a superset of OMG's IDL that allows for IDL interfaces to be grouped to object structures. Also ODL is specified by ITU-T as ITU-ODL and widely used within the telecommunications community.

Given the advantages of SDL as described in section 2 and [Sinnott], and the importance of ODL in the syntactic specification of objects in object-oriented distributed systems, a mapping from ODL to SDL is necessary. In fact, SDL is one of the few formal languages for which an ODL mapping has been made [Born97].

The following table summarises some of the main features of an ODL to SDL mapping.

ODL Structure	SDL Mapping
Group type	Block type
Object type	Block type
Interface type	Process type
Object Reference	Pid
Oneway (asynchronous) Operation	Signal prefixed with pCALL_
Operation (synchronous)	Signal pair. The first signal is prefixed with pCALL_, the second signal prefixed with pREPLY_ or pRAISE_ (if exception raised)
Exception	Signal prefixed with pRAISE_
Basic IDL types, e.g. long, char, float,…	Syntype
Any	not supported
Enum	Newtype with corresponding literals
Typedef	Syntype
Struct	Newtype with corresponding structure
Constant	Synonym

Table 2: Summary of the ODL to SDL Mapping.

Other mappings have been made as well [Björk], however these are based largely around the remote procedure call concept of SDL. The remote procedure concept in SDL is a shorthand notation and is based on a substitution model using signals and states. More precisely, remote procedures are decomposed into two signals. The first carries the outgoing parameters (**in** or **in/out**) and the second the return value of the procedure and all **in/out** parameters. These signals are sent via implicit channels and signalroutes. There are several problems with mapping ODL operations to remote procedures. For example, they prohibit the raising of exceptions – an essential feature in realistic distributed systems. Also, the client side of the remote procedure call is blocked until the server side returns.

The mapping generates client stubs and server skeletons. These act as templates whose behaviour is to be filled in through inheritance. These stubs and skeletons are placed in packages containing the interface specifications in the form of data types, signals, remote procedures, signallists, etc, and the structure information that is inherent in the ODL description in the form of modules, objects and interfaces, as well as a system type representing the ODL description.

Having a mapping defined from ODL to SDL is only a basis on which specifications can be developed. However, tool support is essential to allow for SDL to be used in service creation. Fortunately, SDL has arguably the most developed toolsets of all formal techniques used today.

In order to perform the mapping the Y.SCE [Y.SCE] tool can be used; it allows for ODL and IDL descriptions to be developed (or to be imported) and subsequently mapped to SDL in PR format. The generated SDL stubs can then be imported into the Telelogic TAU toolset [TAU]. This toolset consists of a collection of tools that allow SDL specifications to be both specified, checked, e.g. simulated and validated (using SDT) and subsequently tested.

After having imported the SDL stubs into the SDT tool, the behaviour for the generated processes (ODL interfaces) and procedures (ODL operations) can be specified (cf. Discussion in [SinnFMDS]).

3.5 Implementation Strategy: Targeting SDL to a CORBA platform

A complementary approach to using SDL for defining behaviour to CORBA-IDL/ODL is to define the whole service in SDL and use CORBA as the execution architecture for the SDL processes. This is known as the *SDL-oriented approach* as opposed to the *CORBA-oriented approach* as described above.

In the SDL-oriented approach, IDL and corresponding C++ or Java is generated from an SDL description whereas the CORBA-oriented approach takes IDL as the starting point and generates SDL skeletons which are then completed using the SDL tool.

One of the strong points of the SDL-oriented approach is that you can take full advantage of the capabilities of the SDL tool. Debugging, fast prototyping, validation and testing are especially more cumbersome using the CORBA-oriented approach. A disadvantage of the SDL-oriented approach is that it is more difficult to interface with existing or none-SDL components because IDL in the SDL-oriented approach has the role of an intermediate language. To interface with such external components, wrappers must be developed.

In the SDL-oriented approach, a signal corresponds to an IDL interface, since the signal is known to both the sender and receiver as opposed to the type of the destination process. Thus, each SDL process has an interface for each signal in its input signal set. But it should be noted that the next version of SDL (SDL 2000) will include an interface concept implying that the relationship between SDL concepts and CORBA concepts is more straightforward and, as a side effect, there will be less need for wrappers when communicating with external components.

3.6 Simulation

Using the extensive toolset provided, services specified in SDL can easily be simulated. This allows for carrying out a basic behaviour check. The service designer can verify if the modelled behaviour matches the behaviour expected. Also while simulating a service, MSCs can be created. These can be used at a later stage in the validation activity, e.g. to guide a state space exploration or to support test case generation. In general, the SDL tools provide a good interface to simulate specifications. For instance the SDT simulator tool allows even for changing the interface by adding new buttons with additional behaviour assigned. These buttons can respond by sending signals into the simulation. Clearly, this allows for designing the interface in a more intuitive way and can shorten the time spent on simulating a specification considerably.

A further feature worth noting here is the possibility of interworking of a number of simulators, that is specifications running in different simulators can exchange information by sending signals to processes in the other domain. Here one simulator is part of the *environment* of the other one. Thus the behaviour of the environment in the SDL model can be represented by a specification. In fact, registration is not limited to specifications but also external applications can communicate with a SDL specification being simulated in a simulator. The necessary interface is provided for C++ and also for Java applications.

Furthermore, there is also the possibility of interworking with the CORBA world. However, this is only possible if the mapping rules implemented in the TAU tool were used to generate the SDL stubs from the IDL specification in the first place. Unfortunately, the mapping in the TAU tool is translating IDL operations to SDL remote procedures. This prohibits the use of exceptions in SDL. Hence using this feature is somewhat limited as exceptions are an inherent feature in distributed systems generally, and in CORBA based systems in particular.

Nevertheless, having this interface offered by the postmaster allows for significantly extending simple simulation of a specification as described initially. For instance real graphical user interfaces (GUIs) can be used in conjunction of a simulation. That is the GUIs are used to actually drive a simulation of a SDL model and thus the impression of using the real service is given. This activity is often referred to as animation.

The animation of service specifications might soon be an important factor as new service creation approaches trying to widen participation in service creation are being developed. These approaches mainly focus on being accessible by potentially non-technical people [SinnFMDS]. Hence proper mechanisms to validate the created services are needed also. An example of service animation can be found in [SinnOOPSLA].

In short, the possibilities related to extending the 'old-styled' simulation are considerable and offer new extended ways of checking the behaviour of a service model. Also a simulation does not have to necessarily be driven by a technical person.

3.7 Service Validation

Service validation aims to answer whether the service conforms to the known requirements, which were translated during the service analysis and design to a set of MSCs. The design MSCs may be compared to MSCs generated from simulation of the SDL computational model of the service. Generating MSCs is also useful as a means for error reporting following a simulation of the SDL model. The exhaustive simulation - or only random simulation if the combinatory explosion of states prevents exhaustive simulation - aims to verify the dynamic semantics of the model. The tool detects dynamic errors such as deadlocks, livelocks, and unspecified message reception. For each error found the tool automatically generates a simulation scenario containing the whole sequence of transitions, starting from the initial state of the model and ending up in the error state. Note that to do validation, the system must be closed and therefore the environment (e.g. GUIs) must be modelled. Also, for large systems (e.g. implementations), validation takes a long time and the probability of state space explosions is not negligible.

3.8 Test Case Generation

Testing is an especially challenging area in the formal methods community. Since systems, may be very complex, it is impossible to completely test all possible behaviours, in particular in a distributed service. Instead, testing can be done by identifying certain important or essential scenarios of a given system.

Since the specification and implementation should be based on the same IDL, the nature of the tests should not change. For example, an isolated test of the specification is likely to involve interacting at an operation in an interface of a given computational object and ensuring that when invoked, the correct response is eventually received. This same test should be applicable to the implementation assuming that they are based on the same IDL, i.e. the interface and accompanying operation name will be the same and the parameters should be the same. Typically, such simple tests are not the normal scenarios in distributed systems, where isolated invocations on a given interface require other remote interfaces to be monitored (so called points of control and observation) to ensure that the invocation was as planned. The IDL basis for the testing and observation is the same though.

It is quite possible that given tests can be represented directly in the specification language, i.e. in the form of SDL processes to test the behaviour of the specification, or MSCs. These MSCs can be used by implementers to ensure that the implementation allows for the same sequence of events as given in the MSC. However, instead of relying upon an interpretation of some notation, e.g. SDL or MSCs, a standardised testing language exists: the Tree and Tabular Combined Notation (TTCN). This can then be used to generate code to test the implementation through an appropriate gateway [SchiefEtal].

With TTCN an Abstract Test Suite (ATS) is specified which is independent of test system, hardware and software. Test suites consist of a collections of test cases, where each test case typically consists of sending messages to the implementation under test (IUT) and observing the responses from the IUT until some form of verdict can be assigned, e.g. whether the result was a pass, fail or inconclusive.

We note that the IUT itself is treated as a black box, i.e. only its observable interfaces are considered. The points at which it is tested are termed points of control and observation (PCO). Once an ATS is complete it is converted to an Executable Test Suite (ETS) which can then be used by the system performing the test to see whether the implementation passes or fails the tests.

Several tools exist within the Telelogic TAU toolset [TAUref] that allow for the derivation of tests from SDL specifications. The TTCNlink tool provides an environment that links the SDL specification world represented by the Specification Design Tool (SDT) with the testing world represented by the Interactive TTCN Editor and eXecutor (ITEX) tool. Once a TTCN link executable is generated from the specification it may be opened with ITEX and used to generate the code for testing the system.

Having generated the static parts of the tests, the dynamic parts and the constraint parts associated with the test case can be developed through synchronising the TTCN test case with the SDL system. Once synchronised, the messages to be sent and received can be selected.

Unfortunately, for complex systems the problem of state space exploration as discussed in section 3.7 arises. As well as making the validation activity more complex, this problem also impinges upon the testing activity. Thus the idea of interactively developing test cases is appealing, but is often not possible. This might be manifest through the apparent deadlock that can occurs when an TTCN test case is synchronised with a complex SDL system.

It should be pointed out that conformance testing provides a basis for ensuring that the system being developed has the intended functionality. It is however only a basis. Checking that an implementation is free from certain functional errors does not guarantee that it will not contain other errors not specified in the testing process. In addition, testing non-functional aspects of systems, e.g. their performance in certain run time scenarios, makes the testing process difficult.

3.9 Deploying SDL Systems

Currently deployment concepts are not defined in SDL and there is a need for a flexible, formal and standard process to partition the SDL service design for distributed applications. Therefore, there are initiatives [SCRD28] to define a Distribution Configuration Language (DCL) to assist the deployment and configuration of component-based distributed applications. Given a selected service deployment strategy or, preferably, the optimal deployment scheme, DCL provides the means to enforce this strategy over the network. The optimal logical strategy for component partitioning over network nodes is provided by Optimisation of the Service Logic Deployment Scheme (OSLDS) tool [SCRD38]. The partitioning configuration of the service is expressed in DCL independently of the functionality of each component, i.e., the structure of a component-based application is defined independently of the location of its components.

The DCL partitioning process is comprised of three steps:

(1) Definition of a Configuration Computational model automatically generated from the application’s SDL design model according to the SDL→DCL mapping. This model defines the static configuration architecture, specifying the component hierarchy and their interconnections.
(2) Definition of a Configuration Engineering model specifying node structure and component deployment according to the deployment strategy.
(3) Automatic code generation of the configuration design from the DCL specification. Two code generation strategies are foreseen: either generation of SDL code or direct generation of code in the service target language, C++ or Java.

One of the main issues when using SDL as an implementation language for developing CORBA-based services is to make sure that an SDL system can work appropriately together with its environment.

Instead of using an ORB to provide the interconnection between nodes in a distributed system, it is possible to make use of the internet interoperability protocol (IIOP) which is defined as part of the CORBA standard [OMG]. When doing so, most of the issues normally handled by the ORB, such as marshalling and object adaptation, is maintained internally by the code generated for the SDL system.

There are several advantages with providing a direct connection over IIOP instead of using an ORB, as each part of the SDL system (if it is split up) is linked with a much smaller library (the IIOP library is much smaller than a complete ORB library), giving a smaller footprint. Depending on how the integration between SDL and CORBA is performed, it is also the case that using IIOP directly gives a very distinct performance boost, as the code can be optimized for use with SDL. Furthermore, by providing CORBA support through IIOP, the system becomes ORB independent, thereby allowing any ORB to be used together with the (potentially distributed) SDL system.

Messages that are sent between SDL processes in the same address space are handled internally by the SDL system, but messages to other SDL processes, or to and from CORBA objects in general, are all handled through IIOP, in which message formats have been defined to handle connections, requests, object lookup, etc. All messages sent are encoded on the client side to the transfer syntax and then decoded on the server side to the appropriate language.

The direct use of IIOP is believed to give a much more efficient solution when defining distributed SDL systems than if an ORB is used. If the SDL system does not have any external communication with other CORBA objects the use of IIOP removes the need for an ORB altogether.

4. Recommendations

SDL has certain advantages and disadvantages when used to create and subsequently validate distributed services. It is one of the few formal languages to have evolved with current technologies and approaches to software engineering, e.g. its support for object-orientation and re-use through the package construct, as well as its support for CORBA IDL and TINA-C ODL mappings [YSCEref].

The tool support for SDL is perhaps the most advanced of all the formal languages existing today. Not only do these tools support development environments where analysis and design can be made in a controlled and documented manner, they also allow for thorough investigations of the behaviour of the modelled systems. This might be in the form of generating MSCs to highlight aspects of the behaviour of the system, or through exploring the behaviour of the system in a less constrained way, e.g. looking for situations where signals are dropped by processes or where no processes are able to receive certain signals. Further, these tool environments often allow for the development of test cases from specifications which can subsequently be executed against the implementation of those systems.

In terms of making formal models of distributed systems and being able to reason about those models, SDL is arguably the language of choice. Having an SDL model of a given system is a useful activity in understanding that system However, service creation is not only about making models. It is about making real software that is able to run on real hardware. The SDL tools do allow for real systems to be built from SDL models, however, the main problem with producing such models are their level of abstraction. Modelling systems at a very low level of abstraction, e.g. down to the low-level IDL of objects, results in complex systems being built. Often, formal languages allow for the overall picture of the system functionality to be developed. Basing a model on low level IDL lessens the ability to see this big-picture.

As well as this, the associated problems of complexity and scalability of tools are also impacted by the level of abstraction. Realistic distributed service models are likely to have many objects interacting in non-trivial ways. Loosely speaking, the more complex the system, the more difficult it is to check the behaviour of those systems via tools. The problem of state space explosion arises when systems have large numbers of states with many inputs possible from their environment where these inputs can carry complex parameter structures. Whilst it is possible to model systems whereby the state space explosion problem is lessened, e.g. through reducing the number of interactions with the environment, the problem of state space explosion and scalability of formal techniques more generally remains.

From this it has to be identified that adopting an SDL-based approach to service creation cannot guarantee that the developed services will be perfect. As argued by [Hall] this is one of the common myths of formal methods more generally, i.e. that they can guarantee perfect software. Instead he argues that they are a powerful technique for aiding the understanding of the software development process. It is this pragmatic approach that we endorse. Using SDL and its associated tools will not ensure that services are completely free from errors, however, they will expedite the production of services through removing many of the errors that might exist in the services as early as possible.

5. References

[Björk] M. Björkander, Mapping IDL to SDL, Telelogic AB, 1997.

[Born97] M. Born, A. Hoffmann, M. Winkler, J. Fischer, N. Fischbeck, *Towards a Behavioural Description of ODL*, Proceedings of TINA 97 Conference, Chile.

[CIND] The Cinderella SDL Tool, see www.cinderella.dk

[Hall] J.A. Hall, *The Seven Myths of Formal Methods*, IEEE Software, volume 7(5), pages 11-19, September 1990.

[GEODE] The ObjectGEODE SDL tool, see www.verilogusa.com//home.htm

[OMG] OMG, The Common Object Request Broker: Architecture and Specification, revision 2.0.

[SchiefEtAl] I. Schieferdecker, M. Li, A. Hoffmann, *Conformance Testing of TINA Service Components - the TTCN/CORBA Gateway*, Proceedings of the Intelligence in Services & Networks Conference 1998, Antwerp, May 1998.

[SCRD28] ACTS project SCREEN (AC227), CEC Deliverable D28, *“SCREEN Engineering*

Practices for Component-based Service Creation", Dec 1998.

[SCRD38] ACTS project SCREEN (AC227), CEC Deliverable D38, "*Advanced Service Creation Environment*", Oct 1998.

[SDL] SDL home page, see www.sdl-forum.org

[SDT] The SDT SDL tool, see www.telelogic.se

[Sinnott] R. Sinnott, *Frameworks: The Future of Formal Software Development*, Semantics of Specifications, Journal of Computer Standards and Interfaces, August 1998.

[SinnFMDS] R. Sinnott, M. Kolberg, *Engineering Telecommunication Services with SDL*, accepted for International Conference of Formal Methods for Open Object-Based Distributed Systems, Florence Italy, February 1999.

[SinnOOPSLA] R. Sinnott, M. Kolberg, *Business-Oriented Development of Telecommunication Services*, Proceedings of Seventh OOPSLA Workshop on Behavioral Semantics of Object-Oriented Business and System Specifications, OOPSLA'98, Vancouver, Canada, 1998

[TAU] Telelogic AB, *Getting Started Part 1 – Tutorials on SDT Tools*, Telelogic AB, 1997.

[TinaSA] TINA-C, *Service Architecture*, version 5.0, 16 June 1997.

[TinaODL] TINA-C, TINA Object Definition Language MANUAL, version 2.3, July 1996.

[YSCE] For more information see http://www.fokus.gmd.de/minos/y.sce