Telephone Feature Verification: Translating SDL to TLA$^+$

P. Gibson and D. Méry*[a]

[a]CRIN-CNRS URA 262 et Université Henri Poincaré - Nancy 1 , Bâtiment LORIA, BP239, 54506 Vandœuvre-lès-Nancy (FRANCE)
Email: {gibson,mery}@loria.fr

SDL is commonly used in the early stages of software development. It provides mechanisms for the specification of data structure, data flow, control flow, encapsulation, information hiding and abstract dependencies, through its support for concurrent objects. We propose a mechanism for translating SDL into a TLA$^+$specification, in order to provide a proof-theoretical framework. The preservation of properties through the translation is examined within the framework of a simple state-sequence semantics. We identify the strengths and weaknesses of such an approach, and introduce the translation which binds the two different semantics together. We apply the translation in the verification of two telephone features, and their interaction.

1. Introduction

This paper reports on the research that arose in response to a need for more formal means of verifying telecom feature systems. We believe that TLA$^+$holds many attractions for rigorous development and so aim to utilise it as our mathematical basis. We also believe that object oriented concepts offer real benefits at all stages of software development. Our strategy is based on combining object oriented and temporal logic models in a coherent and complementary manner. This provides us with a compositional approach to verifying systems of interacting telephone features. In this paper we examine the process of generating the first formal design models through a translation from SDL to TLA$^+$.

1.1. SDL

SDL92 [15], which we will now refer to as SDL, is an ITU (formally CCITT) standard language which provides a well accepted means of constructing reactive systems. Using SDL, it is possible to construct models which satisfy certain safety constraints. The validation techniques associated with the language provide automatic simulation environments and, in some cases, automatic code generation. These techniques are useful but not sufficiently rigorous for proving properties about systems: i.e. verifying that the defined behaviour is *correct*. SDL is particularly popular in telephone feature development [2]. The feature interaction problem is difficult, if not impossible, to solve without some form

*This work is partially supported by the contract n°96 1B CNET-FRANCE-TELECOM & CRIN-CNRS-URA262 and by the Institut Universitaire de France

of formal reasoning. SDL has been used to specify features but has not proved useful in the verification of systems of interacting features.

1.1.1. Introducing language concepts

In SDL, behaviour is defined by communicating processes. These communicate by exchanging (parameterised) signals via channels. Processes describe extended finite state machines (EFSMs) which change state only when a signal arrives. A signal may also trigger the sending of other signals and update some local variables. Processes may be grouped into blocks, the most important structural construct in SDL. A block may contain other blocks, connected by channels, or it may contain a set of processes, also connected by channels. The system is finally composed of several blocks, and is, of course, itself a block. An SDL system communicates with its environment by sending and receiving signals to/from that environment. These signals are, like internal communications, transferred by channels. Communication in SDL is asynchronous: each process has a message queue which buffers the incoming messages.

1.1.2. SDL is Object-Based

SDL provides object based conceptualisation [21], since we can view processes as concurrent objects. It provides the following facilities:
Instantiation: Processes, blocks and procedures are defined by types. Multiple instances of these types are indeed distinguishable by unique identifier.
Encapsulation: In SDL, the only way to change both state and local variables is to send a message to the appropriate process according to some well defined protocol.
Genericity: The ability to create generic behaviour expressions is very useful but not fundamentally object oriented.
Inheritance: All SDL types may be specialised by 'inheriting' all features from a supertype. This is one of the fundamental aspects of object oriented languages. However, the SDL inheritance semantics is not powerful enough to cope with the notion of class as type and the need for polymorphism. Inheritance, in SDL (and most other object oriented specification languages), is more like a code re-use mechanism [12]. Thus, we say that SDL is object-based rather than object oriented.

We choose to promote an object oriented interpretation of SDL code (wherever possible), and thus promote an object oriented style of expression which utilises our interpretation mechanisms. The underlying semantic framework is illustrated in figure 1. The object-labelled state transition semantics (O-LSTS) are taken directly from [12] and we are currently working on their implementation in TLA$^+$. They provide the structural consistency as we move from SDL to TLA$^+$, and, from our experience, they help both requirements modellers and designers to communicate.

1.1.3. Validation techniques

Message sequence charts (MSCs) provide one means of validating SDL specifications are [4]. MSCs can be used to formulate requirements of an SDL specification in terms of signal sequences. These can then be validated using an appropriate tool, like the SDL validator [7]. Such a tool is able only to validate MSCs which observe all messages between participants and thus it is limited in its application. Animators and code generators can

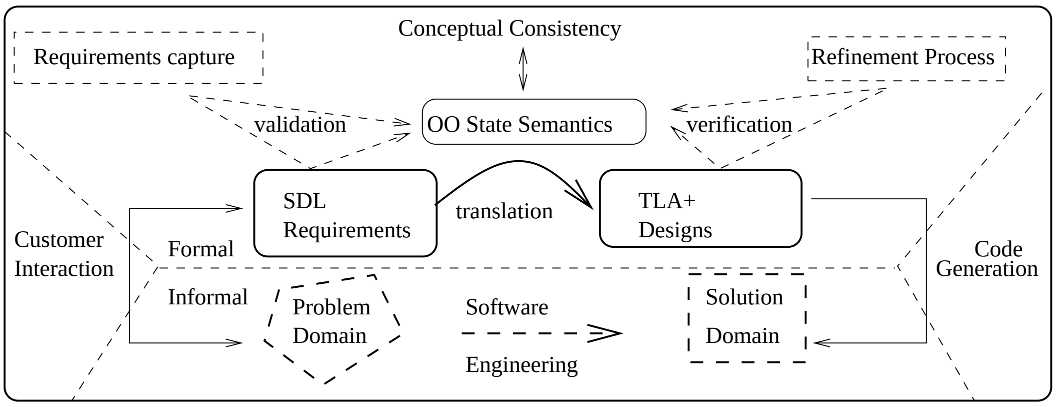

Figure 1: The Semantic Structure

also aid the validation process, which is always going to be informal[2] because of the need for the customer to be satisfied that the behaviour expressions being validated do *correctly* express their needs.

1.2. TLA$^+$

TLA$^+$is a specification language developed by Lamport [19] and based on his temporal logic of actions [20,18], extended by notations of set theory and syntactic structuring mechanisms. It has proved itself to be a formal, abstract and usable specification language [17]. It is founded on simple syntax and semantics which, in our opinion, can be exploited by anyone with basic mathematical ability. It provides a means of temporal reasoning which does not require a large knowledge of logic. In TLA$^+$, a refinement relation between two specifications is simply defined as the deduction relationship in the underlying logic. The proof of a property in TLA$^+$is simply reduced to a proof of implication: *A specification satisfies a property, if a specification implies that property.* TLA$^+$is well supported by the theorem prover TLP [10].

1.3. Translation: A General Motivation

The translation from one language $\mathcal{L}_1$, say, into another language $\mathcal{L}_2$, say, is common in computing science. The utility of the translation is based on the preservation of the properties defined in the behavioural expression being transformed [3,22]. The difficulties of formalising such an approach must be out-weighed by the potential advantages:

Degree of Theoretical Foundation — $\mathcal{L}_1$ may not be formally defined. As such, expressions written in $\mathcal{L}_1$ are not amenable to the type of mathematical analysis (and transformation) which is becoming ever more important in safety critical systems development. By defining a mechanism for translating any given *correctly* defined $\mathcal{L}_1$ expression into an $\mathcal{L}_2$ expression, where $\mathcal{L}_2$ is a formally defined language, one effectively provides $\mathcal{L}_1$ with a rigorous semantics. Furthermore, there is potential for re-use of well-defined concepts in the framework of $\mathcal{L}_2$.

Expressiveness — Different languages have different expressional capabilities. Consequently, some problem domains are more easily modelled in one language than in another.

[2]Informality does not necessarily imply a lack of rigour.

Furthermore, different languages are better at working at different levels of abstraction. Having two different languages obviously supports a wider range of expression.
Practical Support — There are many different software development tools which generally provide the following types of functionality: synthesis, analysis, validation, verification and transformation. For any given language, there are different degrees of tool support for each of these primary functions. Translating from $\mathcal{L}_1$ to $\mathcal{L}_2$ offers the possibility of utilising the best tools in both language domains.
Expanding user-base — Different languages have different user-bases. $\mathcal{L}_1$ may be widely used in a problem domain for which $\mathcal{L}_2$ is not commonly used. To promote the use of $\mathcal{L}_2$, one can translate already existing $\mathcal{L}_1$ behaviour expressions into $\mathcal{L}_2$.
Improving understanding — Having different semantic models may also facilitate a better understanding of the problem domain being considered and of the strengths and weaknesses of each of the languages.
Re-use — An important aspect of all modelling is the ability to re-use previous work. This re-use may come in many different forms [13,16,14,11,24]. Translation between languages facilitates the transfer of all types of re-use from one domain to the other.

1.4. Translation: SDL to TLA$^+$

It is difficult to rigorously relate or transform SDL models. Furthermore, SDL, unlike TLA$^+$is not well suited to the verification of rigorously formulated properties. SDL cannot express properties such as liveness and fairness, which are becoming ever more important in telephone feature specifications. SDL tool support is strongest for model synthesis and validation. TLA$^+$tool support is strongest in the areas of refinement and verification. The translation from SDL to TLA$^+$should introduce many more developers to the advantages of formal languages, in general, and temporal logics, in particular. Through translation we understand much better the feature interactions which are due to fairness problems. Such interactions are difficult to address in the SDL framework. There is much potential for re-use of SDL code in a constructive, object-based manner. Within TLA$^+$we also promote re-use of proof components as a means of facilitating compositional verification. Translating SDL to TLA$^+$permits exploitation of the strengths of both languages in a complementary manner.

1.5. Object Oriented Methods

1.5.1. Advantages

Object oriented concepts have been shown to aid model development at all levels of abstraction and in many different problem domains [1,5,9]. They also lend themselves to a natural state-based conceptualisation [12], incorporating the notions of abstraction, composition, delegation and subclassing in a formal framework. Object oriented methods encourage different types of re-use, facilitate a better understanding of the systems being developed, aid validation and, given a formal semantics, are a step towards the type of constructive rigorous verification which is necessary for *correct* system development.

1.5.2. Objects and Classes

There are two distinct conceptualisations of the term *object* within the semantic framework which we propose. Each class has a set of *member objects*, each of which represents a potential state of an *object instance* of that class. An *object instance* is a state machine whose current state is defined by a reference to a particular *object member* of the class to which it belongs. Meyer states in [25] that 'a class is an executable entity in its own right'. We consider each class member to be an executable finite state machine (FSM), as defined in [12]. State transitions occur only in response to service requests. This model can be realised, after a degree of conceptual manipulation, in the SDL framework by the notions of block, process, channel and signal.

1.5.3. Objects and Fairness

Temporal requirements, such as liveness and fairness, must be included informally in any SDL model. These informal aspects must be formalised by hand as we translate to TLA$^+$: in our telephone feature case studies, the need for fairness is evident in our formal models. The O-LSTS semantics provide formal definitions for object, class, service, subclassing, polymorphism, genericity, and composition. The O-LSTS models complement our TLA$^+$specifications because the underlying model is that of a class as a state transition machine. The two methods share this interpretation, but at different levels of abstraction.

2. An Operational Semantics for SDL

2.1. An Overview

We are required to preserve the semantics of the given SDL models in the TLA$^+$specifications which result from the translation. We must ensure that the semantics which we attribute to SDL (through our translation to TLA$^+$) coincide with those as expressed in the SDL standard. Let us denote any given SDL specification as *sdl* and the semantics of such a specification, expressed as a set of infinite sequences of states closed by stuttering, as $S(sdl)$. Similarly, let us denote any given TLA$^+$specification and its semantics as *tla* and $S(tla)$ respectively. Now, given a translation function T: SDL $\rightarrow$ TLA$^+$, we have the mathematical structure as shown in figure 2:

> Given *sdl*, and $tla \equiv T(sdl)$ then: $S(sdl)$ and $S(tla)$ are two sets of infinite sequences of states related by the following relationship — $S(sdl) \subseteq S(tla)$. Now, we ensure that any property holding for *sdl* also holds for *tla* : if a property Φ holds for *tla*, then $S(tla) \subseteq S(\Phi)$, where $S(\Phi)$ is the semantics of the property expressed as a set of infinite sequences of states, and if $S(sdl) \subseteq S(tla)$, then $S(sdl) \subseteq S(\Phi)$. The preservation guarantees that any property derived from *tla* is also a property of *sdl*.

Now, after generating the *tla* specification we can verify our design steps as refinements. These refinements guarantee that as we add new features to our system we do not compromise the requirement properties of features already developed.

2.2. The SDL definition

The SDL syntax is represented in two different ways: textually and graphically. Informal semantic descriptions are added to these representations in the form of comments. Our method consists of studying SDL syntax defined in the ITU (CCITT) standards document for the purpose of understanding and defining an appropriate subset, which we call CROCOS [23]. A subset of SDL syntax has been examined for translation. (Work is currently being carried out to expand this subset and to incorporate the object based syntactic constructs of SDL-92. At the moment the size of the subset is constrained by

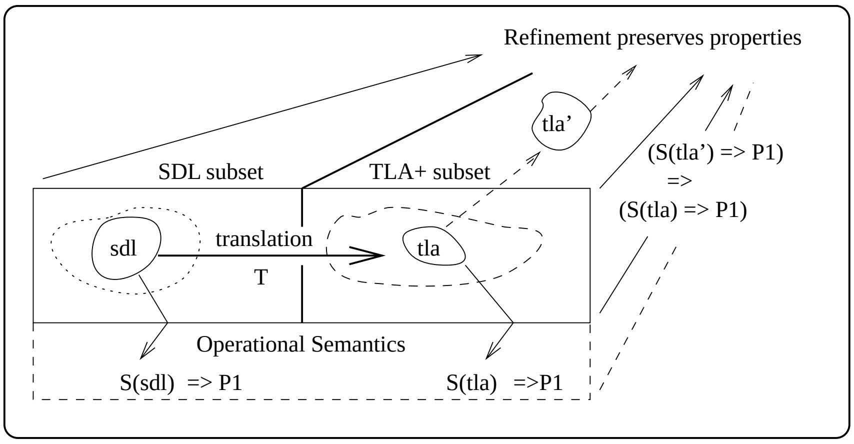

Figure 2: Mathematical Structure

the semantic framework underlying the translation process. Furthermore, we need to formalise the relation between the O-LSTS object oriented concepts and those found in the standard before the translation can be extended.)

2.3. Introducing the language constructs

An SDL text specifies a system as a set of blocks communicating through channels and using signals.

$$\texttt{System}\ S \triangleq \left\{ \begin{array}{ll} \texttt{Signal} & sig_1, \ldots, sig_p; \\ \texttt{Channel} & chan_1, \ldots, chan_q; \\ \texttt{Block} & B_1 \\ \ldots & \ldots \\ \texttt{Block} & B_n \end{array} \right.$$

A block B is either a set of blocks communicating via channels and using signals:

$$\texttt{Block}\ B \triangleq \left\{ \begin{array}{ll} \texttt{Signal} & B \bullet sig_1, \ldots, B \bullet sig_p; \\ \texttt{Channel} & B \bullet chan_1, \ldots, B \bullet chan_q; \\ \texttt{Block} & B \bullet B_1 \\ \ldots & \ldots \\ \texttt{Block} & B \bullet B_n \end{array} \right.$$

or a set of processes communicating via routes:

$$\texttt{Block}\ B \triangleq \left\{ \begin{array}{ll} \texttt{SignalRoute} & B \bullet r_1, \ldots, B \bullet r_p; \\ \texttt{Process} & B \bullet P_1 \\ \ldots & \ldots \\ \texttt{Block} & B \bullet P_n \end{array} \right.$$

An SDL process P is defined by a name, a list of parameters and a set of transitions. A transition is characterized by a starting label and final labels corresponding to the execution of different basic actions as assignment, receipt of a message, sending a message.

$$\texttt{Process } P \triangleq \left\{ \begin{array}{lll} \texttt{Process} & < name > (min, max) & \\ \texttt{Fpar} & < formal\ parameters > & \\ \dots & \dots & \\ \texttt{state } start : & < transition_0 > & \\ & & \\ \texttt{state } label_1 : & < transition_1 > & \\ \dots & \dots & \dots \\ \texttt{state } label_n : & < transition_n > & \end{array} \right.$$

The translation of an SDL system into a TLA text uses the hiding of variable as channel. The mapping is relatively obvious and it allows us to give a TLA-based semantics for SDL. The fact that SDL requires the definition of algebraic datatypes does not suggest any problem because we suppose that users can utilise the abstract data type of SDL as set-theoretical elements. In sections 2.4 to 2.7, we provide a semi-formal overview of the way in which the translation maintains the semantics and structure of SDL systems. (As an overview, these sections can act only as a brief justification of our approach: more details can be obtained directly from the authors.)

2.4. Naming elements of SDL text

The naming of SDL constructs leads to a simple partition of sets as variable parameters in the corresponding TLA$^+$specification.

`System`	:	set of names for system
`Block`	:	set of names for blocks
`Process`	:	set of names for processes
`Channel`	:	set of names for channels
`Signal`	:	set of names for signals
`Variable`	:	set of names for variables
`State`	:	set of names for states

module *Basic-Notations*

variable

$P, S, B, Chan, Sig$

2.5. Transforming transitions into TLA actions

A transition is written as follows:

$$T \triangleq \left\{ \begin{array}{lllll} STATE & label_1 & S_1 & NEXTSTATE & label'_1 \\ & \dots & \dots & NEXTSTATE & \dots \\ & label_n & S_n & NEXTSTATE & label'_n \end{array} \right.$$

The control state is defined as a special variable attached to every process :

- $P \bullet L$ means that L is a label variable for P of the process P.
- $P \bullet L$ is the current value of the control of P, and $P \bullet L'$ will be the next value of L.
- the process P has variables that may be modified by `TASK` and P can send or receive messages. Labels are located at the beginning of every state and before every operation. A global assertion, namely an invariant on control, is defined to handle the relationship between labels.

A control invariant for P is defined from the text of the process P : $Icontrol(P)$.
A control invariant for a block B is defined from components of B:

$$Icontrol(B) \triangleq \bigwedge_{B' \in B \bullet Block} Icontrol(B')$$

A control invariant for a system S is defined from components of S:

$$Icontrol(S) \triangleq \bigwedge_{B \in S \bullet Block} Icontrol(B)$$

The transformation of T into an action of TLA is as follows:

- T is decomposed into $T_1; T_2; \ldots; T_n$
- for any i in $\{1, \ldots, n\}$, T_i becomes $\mathcal{A}_i(T)(x, x')$
- $\mathcal{A}$ is $\mathcal{A}(T_1)(x, x') \vee \mathcal{A}(T_n)(x, x')$
- for any i in $\{1, \ldots, n\}$, $\mathcal{A}(T_i)$ is $\mathcal{A}(T_{i1})(x, x') \vee \mathcal{A}(T_{ij_n})(x, x')$

2.6. The Next relation for a system S

We require the following notation in our definition of *Next*:

- For any process P, $\mathcal{N}(P)$ is the disjunction of actions in P.
- For any block B, $\mathcal{B} \triangleq \bigwedge_{B' \in B.Block} \mathcal{N}(B')$
- For any system S, $\mathcal{N}(S) \triangleq \bigwedge_{B \in S.Block} \mathcal{N}(S)$

Finally, the global next relation for S is simply defined by a TLA$^+$module *Next*:

module *Next*

extends . . .

```
Other modules are required for the definition of Next and
are imported
```

$$\mathcal{GN}(S) \triangleq \mathcal{N}(S) \wedge Icontrol(S)$$

2.7. A semantics for S

The definition of a semantics for S leads us to characterize a set of behaviors for S with the help of a TLA formula; in the style of Lamport we write:

module *semantics-of-S*

extends ...

```
Other modules are required for the definition of 𝒮(S) and
are imported
```

$$\mathcal{S}(S) \triangleq Init(S) \land \mathcal{GN}(S) \land Fairness(S)$$

$Fairness(S)$ expresses a fairness condition for S. $Fairness(S)$ can be defined with the weak fairness (WF) or strong fairness (SF) predicates. WF$x\mathcal{A}$ states that, if $\mathcal{A}$ is continuously enabled from a given state, then $\mathcal{A}$ will be executed, while there is a possible stuttering on x; SF$x\mathcal{A}$ states that, if $\mathcal{A}$ is infinitly often enabled from a given state, then $\mathcal{A}$ will be executed.

2.8. Translation: The Formalisation

The main problem is to define the soundness of the translation with respect to a semantics of system. Intuitively, a SDL system is semantically defined as a set of traces: a trace of states allows us to observe the transformation of data with actions. A notion of action can be clearly extracted from the ITU (CCITT) documentation on SDL.

Definition 1 *Semantics of a system*
A system S over a set of variables V is modelled by a set of behaviours Behaviour(S, V,) over V: $\sigma \in Behaviour(S,V,)$ *and* $\sigma : \sigma_0 \xrightarrow{\mathcal{A}_0} \sigma_1 \ldots \xrightarrow{\mathcal{A}_{i-1}} \sigma_{i-1} \xrightarrow{\mathcal{A}_i} \sigma_i \xrightarrow{\mathcal{A}_{i+1}} \ldots \sigma_0, \sigma_1, \sigma_{i-1}, \sigma_i$ *are states of the system S over V and are defined as mapping from V to the memory values* $\mathcal{A}_0, \ldots, \mathcal{A}_{i-1}, \mathcal{A}_i, \mathcal{A}_{i+1} \ldots$ *are actions over V : an action* $\mathcal{A}$ *is defined as a set of couples of states.*

A system is characterized by a set of infinite traces over a set of variables. However, SDL uses infinite communication channels that leads to a first observation that automata in SDL are generally not finite state. A second observation is that we need to use a variable for every channel. Now we turn to the translation of an SDL specification into a TLA formula and the definition of the semantics of a system S written as an SDL specification:

- Action is [TASK X := E]: $(x' = e(x) \land inv(x))$, where $inv(x)$ is an invariant of x.
- Action is [Start Choice Task1 Task2 Task3 ... Taskn End]: $T(Task1) \lor \ldots \lor T(Taskn)$

The question of the soundness of the translation is crucial. The result is based on equivalence by stuttering, which simply states that two traces are equivalent by stuttering, if they are equal when you forget idle actions[20].

2.8.1. Translation: work to date

The translation of SDL into a formal framework has been done in our CROCOS environment [23], and we have translated a subset of SDL using the *wp* operator for atomic actions. We are currently considering translation of the full SDL language: the additional structure in the TLA$^+$logic, together with our O-LSTS semantic integration, provides a means to facilitate a more complete object-based conceptualisation of SDL. An important thing to note in the translation is the introduction of fairness assumptions which could not be expressed in the original SDL systems. These fairness assumptions reflect informal assumptions that we have made about the objects in our models. In particular, they reflect the fact that a system of concurrent objects is actually modelled using interleaving and a *nondeterministic scheduling*. The TLA$^+$semantics can be used to formally state our informal assumptions about the fairness of concurrent objects in the SDL models. This is important in the domain of telephone feature development. The behaviour specified by the resulting TLA$^+$models can be viewed as a refinment of the originating SDL models because of the way in which the nondeterminism is treated at a different level of abstraction.

2.9. Verification of SDL programs

Our technique has been developed to improve the way to prove properties about programs written in SDL. Several techniques can be applied to handle a SDL specification translated into a TLA specification. A model checking technique is applied when the TLA specification is finite-state[3] A direct consequence is that we need to translate a TLA specification into an equivalent finite automaton in order to check properties on the SDL program. Theorem proving is a more powerful approach which requires theorem prover tool. We have such a tool, namely TLP [10], for TLA. Hence, when we have translated an SDL program into an equivalent TLA specification, we prove properties of the SDL program by proving properties of the TLA$^+$produced in the translation. Then during design, each refinement of the TLA$^+$specification maintains the properties as required by the SDL model.

3. A Simple Case Study: Telephone Features

The feature interaction problem is concerned with what happens when we try to put a wide range of telephone features together in one system. An interaction is said to occur if the complete system does not fulfil the requirements of each individual component feature. Feature interaction is one of the most difficult problems in this problem domain [6,8] and must be considered in the initial stages of specification. Finite state automata have been shown to play a role in the analysis of safety properties in systems in which feature interactions can occur. However, it is now becoming clear that liveness and fairness properties can also be important in such interactions. In this simple study we show the problems that can occur when defining new features (call waiting and three-way calling are used as examples) which extend standard telephone functionality. We compare an *SDL-alone* approach with one based on a translation to TLA$^+$.

3.1. POTS

The Plain Old Telephone Service (POTS) requirements is represented, in figure 3, by our O-LSTS diagram of the simplest of telephones. (In the diagram we have not represented null state transitions which return some value to the external interface, through services hook and signal, but do not change the internal state of the system. Their intended semantics should be evident from the state labels.)

[3]If a TLA specification has only finite domains, then the generated automaton is finite.

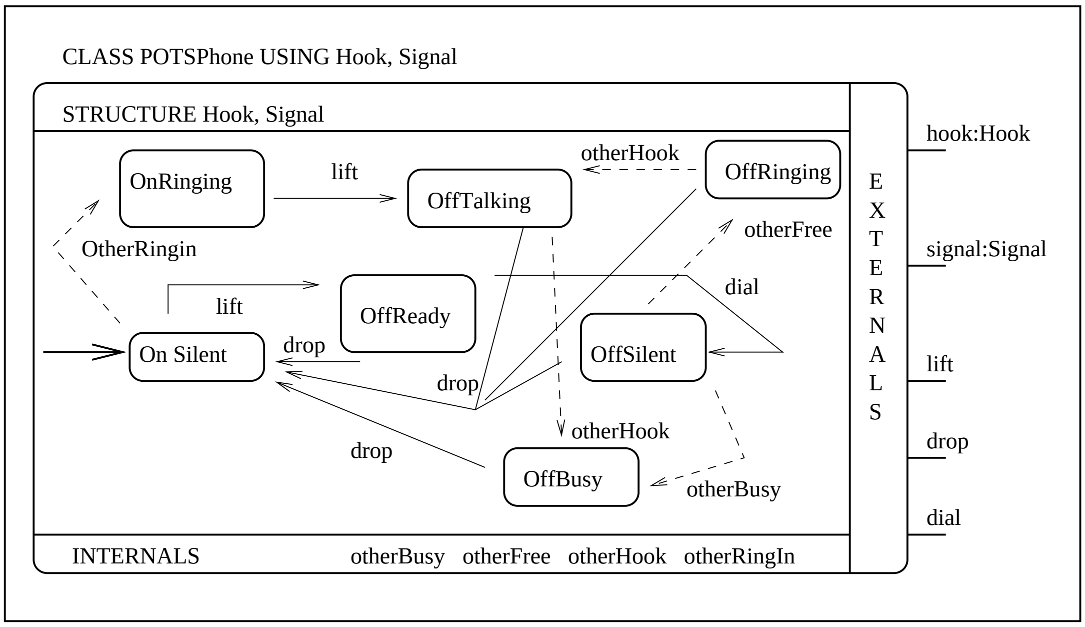

Figure 3: A POTS Phone Requirements Model

3.2. Call Waiting (CW) and POTS

CW functionality is enabled only when talking to someone else, and a third person tries to call. With CW, they will not get a busy tone (as before) but will hear a ringing tone until they are held. CW permits the talking to one person concurrently with the holding of another. It also permits the switching of the talking and held persons. Either of the two other users can hang up to leave the CW user in the standard POTS state where they are talking to one person. The CW subscriber can also hang up leaving them in the standard POTS `OnSilent` state. This is illustrated in the *partial* state transition diagram for the extended telephone (we do not show all the telephone states and transitions as they are preserved by the extension mechanism), in figure 4.

3.3. Three Way Calling (TWC) and POTS

If I subscribe to TWC then the feature is enabled only when I am talking to someone and I receive a call from a third party. I then have a new service (connect) which permits me to talk with both callers at the same time. In this new state I can also disconnect either one of the two callers. This is illustrated in the *partial* state transition diagram for the extended telephone, in figure 5.

3.4. CW and TWC: A feature interaction?

Traditionally, the argument for these two features interacting is as follows:

> What happens if both features are activated and we are talking to someone when we receive an incoming call? There is an ambiguity because the system doesn't know which feature to execute: the CW or the TWC.

We must ask whether there is really a problem here. The question is best addressed within a formal framework: below, we examine the SDL-TLA^{+}models of the behaviour. The *partial* O-LSTS diagram, in figure 6, is common to both models.

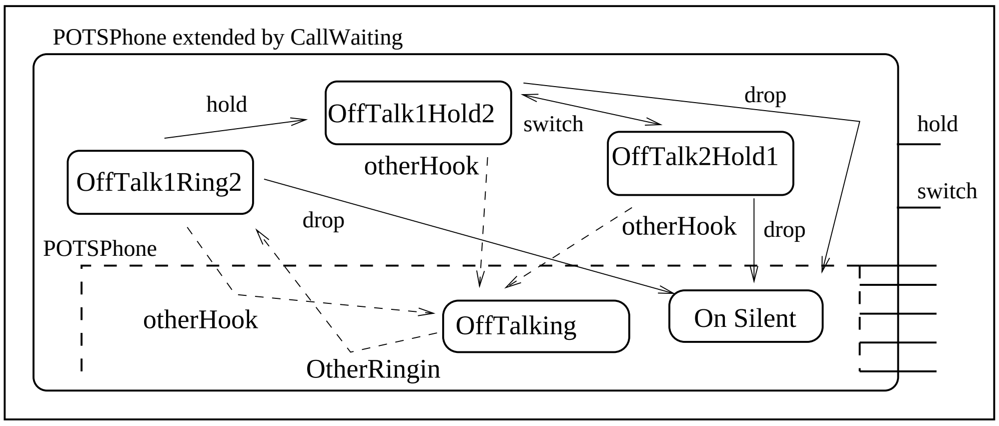

Figure 4: A POTS Call Waiting Extension

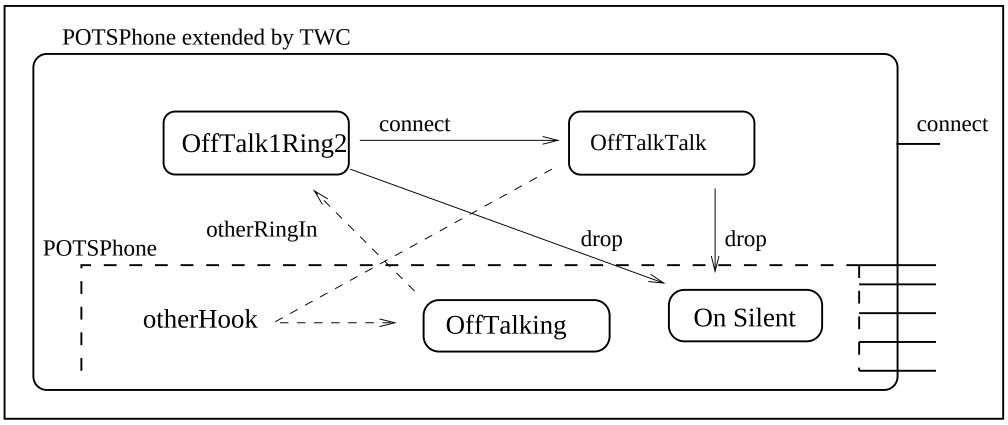

Figure 5: A POTS Three Way Calling Extension

3.5. Standard SDL Approachs

3.5.1. Informal Validation: Animation

The SDL state transition model can be animated for validation. A black-box validation is done when the animator shows only execution behaviour as a trace of actions. White-box testing also permits the user to see the internal state representation of the system during execution. Both types of validation are useful for interacting with the customer, but they are not sufficient when we consider systems with large numbers of states.

3.5.2. Rigorous Validation: Using MSCs

The SDT validator of MSCs can perform an exhaustive search on the state space of the finite state machines representing the SDL specifications. MSCs help to structure the validation process. The user no longer needs to interact directly with a large, complex machine; instead, they specify sets of requirements as MSCs and these are automatially verified against the underlying model. There are limitations to this approach:

- It is sometimes necessary to construct huge MSCs which are difficult to understand

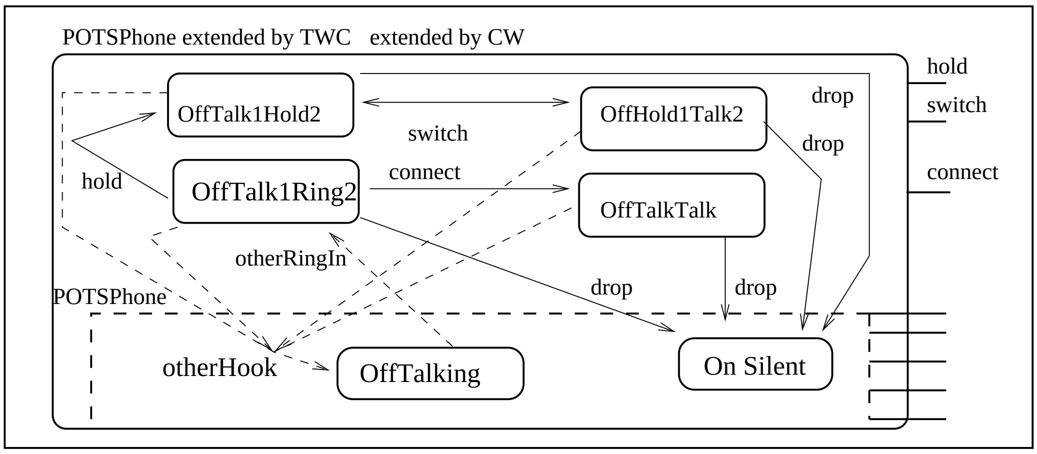

Figure 6: A Feature Composition

and ultimately may not even correctly define the requirements properties that are to be verified.

- Safety properties can be proved in this way but were found, in practice, to be too complicated to be expressed using MSCs
- Liveness properties cannot be considered
- As systems are refined or extended, the validation process must be carried out again. In other words, the approach is not compositional.

3.5.3. Verifying our Feature Composition

The process of validation does not aid us with the feature interaction problem. Consider the following situation:

The POTS telephone is validated against MSCs $p1$ and $p2$
The CW-telephone is validated against MSCs $cw1$ and $cw2$

We would like to be able to formally verify each extension of the original POTS system in such a way that the properties are preserved automatically, for example:

CW-telephone is *verified against* a POTS telephone $\Rightarrow$
CW-telephone automatically fulfils properties $p1$ and $p2$

In the standard SDL approach $p1$ and $p2$ must be separately verified against the CW-telephone specification. The state space of a simple telephone and a few features is relatively small and so is amenable to exhaustive testing. Consequently, we can verify the incremental feature extensions of the POTS phone using the standard SDL tools. However, the state space of the POTS system (where we have a set of telephones and their features) was too large for automatic verification. It was possible to validate subsets of the state space but there was no means of constructing a larger validation (or verification) of the whole system in a rigorous fashion.

3.6. Verification in TLA$^+$

TLA$^+$is not as useful as SDL for the valdiation of our requirements models Using TLA$^+$for verification, however, is more powerful than the standard SDL approach:
* **Liveness Conditions** can be verified. For example, using TLA$^+$we are able to prove the following requirements which we could not find a means of verifying using just the SDL and MSC formalisms:

- I can *always* hang up my phone, lift it and start a new call.
- If I am talking with two people at the same time and I hang up then the two people will *always* end up talking to each other.
- There is an absence of circular holds in the system: i.e. we cannot have a situation where: $P1$ holds $P2$ holds ... holds $P1$.
- When I dial a number I will *eventually* get a busy or ringing signal.

These properties are formalised in TLA$^+$as follows:

module *SPECIFICATION*

extends ...

theorems

`I can always hang up my phone, lift it and start a new call.`

$$DEFINITION1 \triangleq \Box(\textsc{Enabled}\ \neg Hang\_Up\_My\_Phone \bullet Lift\_It \bullet Start\_New\_Call)$$

`If I am talking with two people at the same time and I hang up then the two people will always end up talking to each other.`

$$DEFINITION2 \triangleq \Box(\forall p1, p2. Talking(I, p1, p2) \land Hang\_On(I) \Rightarrow Talking(p1, p2))$$

`There is an absence of circular holds in the system: i.e. we can never have a situation where: P1 holds P2 holds ... holds P1.`

$$DEFINITION3 \triangleq \Box(\neg(\exists P_1, P_2, \ldots, P_n : \land \forall i \in \{1 \ldots n-1\} : Hold(P_i, P_{i+1}))) \\ \land Hold(P_n, P_1)$$

`When I dial a number I will eventually get a busy or ringing signal.`

$$DEFINITION4 \triangleq Dial(I, Number) \leadsto (Busy\_Toning(I.Number) \lor Ringing(I, Number))$$

* **Compositional verification** can be mechanised because verification is represented by a logical implication which can be handled directly by the TLA$^+$theorem prover TLP [10].
* **Power of expression** is better when using TLA$^+$rather than MSCs for specifying requirements properties for verification.

3.7. CW and TWC: Is there an interaction?

The two features *do* interact when the `hold` and `connect` actions are obliged (by each of their feature requirements) to occur when in the state `OffTalk1Ring2` — this is the standard model. Clearly, these two actions cannot both be performed and, in this case, an interaction is said to occur because both features cannot meet their obligations. With the TLA$^+$notion of liveness, we can guarantee that we do not stay in state `OffTalk1Ring2` whilst obliging neither of the two actions individually. Thus, we have no contradictory requirements and the two features do not interact. It is the the user who chooses which feature to execute; and if such a choice is not made liveness can be used to gaurantee that a nondeterministic action *eventually* changes the state of the system.

4. Conclusions and Future Work

This paper reports on a continuing experiment into the use of formal methods in showing the *correctness* of telephone feature requirements models (written in SDL). It is clear, even at this early stage, that there is much complexity involved. Some of this complexity is presently being 'factored out' in the development of appropriate translation tools. One of the most difficult aspects of verification is the sheer size of the task. What is clear is that the complexities of verification increase much more than linearly with respect to the size of the models to be verified (this is particularly true for our telephone feature interaction case study). At worst, the problem seems to be almost exponential in nature. What is needed is a constructive approach to verification in the same vein as the constructive approaches to validation which are evident in many of the modern object oriented analysis and requirements capture methods. However, the problem of constructive verification hinges on the definition of constructive proof operators which can be used in conjunction with the operators used in the synthesis of requirements models. This is our current area of research.

We believe that there is much to be achieved through the integration of object oriented concepts, simple operational semantics and temporal logics. This work is a small step towards achieving such an integration. SDL is useful in telephone feature development during analysis and requirements modelling. TLA$^+$is useful for formal verification of design steps and system extensions, and it also provides us with the means of reasoning about fairness aspects of telephone services. The object oriented state-transition based semantics provide us with a good basis on which to connect these two languages in a complementary fashion. We have illustrated how a subset of SDL can be translated; this subset will be expanded upon as we integrate the O-LSTS semantics with the extended finite state semantics in the SDL standard. This will also facilitate a more compositional approach to TLA$^+$verification.

REFERENCES

1. G. Booch. *Object Oriented Development.* IEE Software Engineering, February 1986.
2. L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunications Systems.* IOS Press, 1994.
3. N. Brown and D. Méry. A proof environment for concurrent programs. In J. C. P. Woodcock, editor, *FME'93: Industrial-Strength Formal Methods*, pages 196–215. IFAD, Springer-Verlag, 1993. LNCS 670.
4. M. Broy. Towards a semantics for sdl. Technical report, PASSAU University, 1989.
5. P. Coad and E. Yourdon. *Object oriented analysis.* Prentice-Hall (Yourdon Press), 1990.
6. P. Combes, M. Michel, W. Bouma, and H. Velthuijsen. Formalisation of properties for feature interaction detection. In *ISN Conference*, 1993.
7. P. Combes, M. Michel, and B. Renard. Formalisation verification of telecommunications service interactions using SDL methods and tools. In *6th SDL Forum.* North Holland, 1993.
8. P. Combes and S. Pickin. Formalisation of a user view of network and services for feature interaction detection. In L. G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Software System*, pages 120–135. IOS Press, 1994.
9. Brad Cox. *Object oriented programming: an evolutionary approach.* Addison-Wesley, 1986.
10. U. Engberg. *TLP Manual-(release 2. 5a)*-PRELIMINARY. Department of Computer Science, Aarhus University, May 1994.
11. R. Fairley. *Software Engineering Concepts.* McGraw Hill, New York, 1985.
12. J. Paul Gibson. *Formal Object Oriented Development of Software Systems Using LOTOS.* Technical report CSM-114, Stirling University, September 1993.
13. Joseph Gougen. Reusing and interconnecting software components. *Computer*, 20, February 1986.
14. Raymonde Guindon. Knowledge exploited by experts during software system design. *International Journal of Man-Machine Studies*, 33(3):279–304, 1990.
15. ITU. *Specification and description language (SDL) ITU-T Recommendation Z.100*, revision 1 edition, 1994.
16. R. E. Johnstone and B. Foote. Designing re-usable classes. *Journal of Object Oriented Programming JOOP*, pages 22–35, June 1988.
17. L. Lamport. Hybrid systems in TLA$^+$. In Grossman, Nerode, Ravn, and Rischel, editors, *Workshop on Theory of Hybrid Systems.* Springer-Verlag, 1992. LNCS 736.
18. L. Lamport. A temporal logic of actions. *Transactions On Programming Languages and Systems*, 21(9):768–775, September 1995.
19. L. Lamport. TLA$^+$. Technical report, Digital Equipment Corporation, 5th july 1995.
20. L. Lamport. TLA in pictures. *IEEE Trans. on SE*, 16(3):872–923, May 1995.
21. T. Lindner and C. (editors) Lewerentz. *Case Study Production Cell A Comparitive Study in Formal Software Development.* FZI-Publication, 1994.
22. D. Méry. *Une Méthode de Raffinement et de Développement pour la Programmation Parallèle.* PhD thesis, Université de Nancy 1,UFR STMIA, DFD Informatique, February 1993. Doctorat d'Etat.
23. D. Méry and A. Mokkedem. CROCOS: An integrated environment for interactive verification of SDL specifications. In G. Bochmann, editor, *Computer-Aided Verification Proceedings.* Springer Verlag, 1992.
24. B. Meyer. Genericity versus inheritance. In *Object Oriented Programming Languages Systems and Applications (OOPSLA 86) As ACM SIGPLAN 21*, November 1986.
25. B. Meyer. *Eiffel: The Language.* Prentice Hall International Ltd., 1992.