19

Combining Formal Methods: An Exercise in Integration

J.-Ch. Grégoire and M. Ferguson
Telecommunications Software Group, INRS-Telecommunications
{gregoire,mike}@inrs-telecom.uquebec.ca
Lou Pino[1]
Stentor

Abstract

Formal methods can rarely capture all the dimensions of a software project. Different aspects of a project are thus typically formalized separately, with little or no integration. We study here the integration of a formal specification methods and a validation method. The methods that we use are LARCH and PROMELA/SPIN. LARCH is an algebraic specification method, specialized in the specification of abstract data types and their properties. PROMELA/SPIN is a modeling and verification package for concurrent systems, based on a process/communication channel abstraction.
We show how LARCH provides a natural integration path for the two methods, describe the features of an integration language, and discuss the problems we have encountered.

Keywords

SPIN, LARCH, integration of formal methods.

1 Introduction

Formal methods are mathematical languages with proof systems and associated tools. They are used in software projects to bring *precision*, not only in the expression of the requirements, but also at the design level in the expression and the verification of the analytical properties that the system is supposed to present [Cooke92].

We describe the integration of two formal methods to specify and validate a distributed application. Indeed, formal methods tend to excel in a specific dimension of software, e.g. data or functions, but seldom across all dimensions. It is thus interesting to try to combine different methods to take advantage of their strong points.

We are interested in formalizing the properties of distributed administration applications, such as those encountered in a workstation environment. In a related activity [Grégoire93], we have developed an implementation tool based on a remote programming concept to support distributed administration. Our goal is to develop formal tools to help with the conceptual development and verification of applications in this context.

We have integrated a formal method based on algebraic specification of abstract data

[1]work done while this author was a student at INRS-Télécommunications.

types together with another focusing on expressions and verification of concurrency. In this integration process, the prime focus is given to data abstractions. As these abstractions are refined, interactions are identified and formalized in the other formalism.

In the following sections, we present the benefits of using formal methods and introduce the two we have selected, LARCH and PROMELA/SPIN. We then describe how the integration is realized and illustrate the result with an example. We then discuss our achievement and draw some conclusions.

At this stage, we should stress the pragmatic nature of this work. We have not tried to introduce a new formal method, but rather to study the problem of integrating two well supported methods with the long term view of the development of a methodology.

2 Formal Methods

Formal methods are used to capture abstractions in a precise mathematical language, express their properties, and verify them. They can also be used to validate an implementation with regard to a formal description.

Formal methods bring *precision* to a specification. This means not only a clear, unambiguous terminology, but also verification of the *desired properties* of the system. It is thus possible to relate directly design decisions to their intent.

We focus here on the formalization of functional, data and control specifications. In an object-oriented, or rather object-based framework, such as the one provided by abstract data types, data and functions are encapsulated together. Control is defined as patterns of methods (functions) invocation, depending on the internal states of the objects. Reasoning on the control part requires breaking up the encapsulation and a different verification framework.

We therefore require here two formal methods. The first one, LARCH, deals with encapsulated abstractions, while the other one, PROMELA/SPIN focuses on control.

2.1 LARCH

LARCH [Guttag et al.93] is a definitional specification method that formalizes abstract data type (ADT)-like abstractions. LARCH has a *two-tiered* approach to specification where, on the one hand, one captures the state-independent properties of program abstractions in the LARCH SHARED LANGUAGE (LSL), and, on the other hand, a LARCH INTERFACE LANGUAGE (LIL) captures the implications of the operations on the ADT in terms of the state model of some implementation language.

In LARCH, an ADT is defined in a *trait*, consisting of *sorts*, *signatures* for a set of *operators* and *axioms*. These latter define *equalities* between terms composed of the operators. A trait also defines *derived axioms*, which should be a logical consequence of the axioms. A LARCH trait forms a theory in equational logic. Extensions and refinements are done simply by adding new axioms, with the only condition that the set must remain consistent. This is known as the *loose semantics* model, as opposed to initial algebra semantics used in languages such as ACT-ONE [Ehrig et al.85]. A trait is written in the LSL.

Traits form theories which should ideally be ***consistent***, ***contained*** and ***complete***. Consistency, means that is is not possible to derive the equation $true == false$ from the theories; in LARCH, this can be checked with the help of a proof assistant. Containment is the property that the invariants can be derived from the theories. Completeness means that an operator is fully defined by the theory and the invariant. In LARCH, for practical purposes, this is not explicitly required of a trait. It thus becomes possible to support incremental refinement of specifications and the formalization of reusable, incomplete abstractions, such as generic components.

The other tier is an ***interface***, expressed in a LIL, that describes state-dependent effects of the program abstractions, in terms of features of a programming language [Chen89]. Such "effects" can be state transformation, exceptions, iterators or also concurrency. Interfaces are explicitly related to LSL traits. Each procedure of the interface is characterized by the requirements on the state space before it is invoked, which variables may be modified and what must be changed when the procedure returns [2].

LARCH supports notions of reuse through parameterized abstract data types. It is also ***constructive***, since ADT can be built up from other ADTs through an inclusion mechanism, and inherit their theories. Since theories do not have to be complete, they can simply capture assumptions or implications, or general mathematical properties (e.g. monoid, group).

2.2 PROMELA/SPIN

PROMELA [Holzmann91] is an operational specification language, tailored to the expression and analysis of control behaviour as communicating finite-state machines. The finite-state machines are executed by ***processes***, which exchange information over bounded communication channels, either synchronously or asynchronously.

Processes and channels are reusable abstractions. The language also provides other elements useful to express control, namely, boolean and integer variables, and array structures. There is however no other form of data structure, nor a procedure abstraction. It is thus rather difficult, and inadequate, to express data transformation. The language also has nondeterministic selection and iteration control statement, modeled as guarded commands analogous to CSP's [Hoare85].

A PROMELA program is semantically equivalent to the traces of all possible executions. SPIN, an associated verification tool computes the traces and checks that they verify properties such as reachability, local or global assertions, some temporal logic claims, absence of deadlocks and livelocks.

2.3 Justification

LARCH and PROMELA/SPIN are complementary. The first one focuses on the description of abstract properties of system components, while the other allows the expression and analysis of the behaviour of interactions in their operations.

Both LARCH and PROMELA have associated tools, such as syntax checkers, proof

[2]the notions are analogous to pre- and post-conditions, and invariants

assistants and property checkers. The existence of tools differentiates formal methods from formal languages, and really makes their use attractive. Rather than being simply a precise way to express requirements, they become a practical mean to verify them.

There are alternatives to either of them. However, both LARCH and PROMELA/SPIN have the advantage of having well-documented, although still evolving, methodological steps governing their use, as well as more robust tools.

3 Integration

Combining different formal methods into a single framework is not an easy task. They usually have different semantic models which cannot be integrated. One can only hope to separate orthogonal concerns and treat each of them with an appropriate tool. When the orthogonality hypothesis is not entirely valid, interactions must be formalized and analyzed.

LARCH's two-tiered model gives us an interesting framework to work with. Whereas the shared language describes the operators and the abstract properties of the data, an interface language formalizes the effects of the operations on the variables of the target language. In the interface language, we find the operators of the trait in a more concrete (i.e. implementation language dependent) form, with a formalization of their behaviour in terms of the state model of a specific programming language. This formalization typically takes the form of an axiomatization describing pre and postconditions, as well as invariants.

It thus seems natural to integrate PROMELA/SPIN as an interface language where not only sequential, but also concurrent aspects of interaction are formalized [3].

3.1 LA/PRO: the LARCH/PROMELA interface language

One of the key notions of an interface language is the state space. States are mappings from a memory location to an object. The kind of objects we have here are:

- 1, 8, 16 or 32 bits integer values, where boolean and character values are coerced to 1 and 8 bits integer values, respectively;
- sized memory locations *locs* for integer values;
- arrays;
- communication channels, which are buffers of fixed size for tuples of data values, used for synchronous and asynchronous communications;
- container values, typeless and of any size, only assignable and comparable;
- memory locations *locCs* for those containers, with a type related to a trait;
- procedure declarations related to trait operators, with their associated pre and postconditions, and invariants;
- procedure definitions.

[3]It is interesting to note here that, akin to imperative languages, PROMELA is a *pragmatic* language, rather than a formal language. Its semantics are however completely described in terms of finite transition systems, on which verification methods can be applied.

The state space of LA/PRO consists of 2 parts: native PROMELA types, and ADT *containers*.

Native PROMELA types are integer values defined over several finite domains, and array structures. Integer values behave like their C counterparts. Arrays correspond to PASCAL-style arrays. This language has no notion of pointer. It basically takes the PROMELA language and expands it with a couple of structures to relate operations on sorts to control. For convenience in the formulation of some expressions, we also extend the PROMELA language with set operators, including iterators. Sets are interpreted over integer values.

The LIL's purpose is relate operations in the implementation language to formally defined operations on sorts, and describe the effects of the latters on the state model. The state model is therefore extended to abstract types, not directly supported by the language, at least at the level of abstraction we are dealing with. Abstract types are represented by containers in our model.

The LIL will also capture restrictions on the use of ADT and operators. Since LARCH's philosophy is to be general and to underspecify at the LSL level, domain restrictions (i.e. partial functions) will be captured in the LIL.

The extension of LA/PRO to a typical LIL is the *communication structure.* The communication structure serves two purposes: synchronization of operations and data transfer. We use them to express *when* the sequencing can occur, and to invoke operations remotely through message passing and data transfer. The synchronization part is where the encapsulation of state may be broken, albeit in a controlled way: the state information of different ADT's must be exchanged for the proper behaviour to be decided. This also means that verification can no longer be purely local, unlike the more usual pre and postcondition model.

A LA/PRO specification serves two purposes: the first one is to formalize the implementation of an ADT in terms of lower level operations, especially when cooperation is involved. The other is to relate the high level operations of the system to ADTs. Unless we have distributed data structures, the communication structure appears only in the this latter case. It is then straightforward to validate the communication model on its own. When we do have distributed data structures though, communications can be embedded within the operator implementation hierarchy and the interactions become hidden.

LA/PRO statically verifies the well-formedness of expressions involving sorts, that is, the existence of operators and the compatibility of the types.

3.2 Domain Integration

The integration problem in this context is the relation between the semantic model of ADT's and that of PROMELA/SPIN. It is directly related to interactions between the container and control state spaces, such as, for example, the use of a "contained" value to make a decision on the flow of control. There are different ways to exploit PROMELA/SPIN's modeling concepts depending on the nature of the interaction we want to verify.

The major problem we are confronted with is the cardinality of the domain of the ADT, and its contribution to the explosion of the state space. Domains of ADT's need not be explicitly defined. When they are, for example when we manipulate naturals, they cannot be directly integrated into a finite-state model. We use several techniques to alleviate this problem.

The first technique is a non-deterministic selection. When a branching structure is affected by the result of an ADT operation, we can simply consider the alternatives independently of the values directing them. The most simple example is a boolean operation: we have two branches depending on the value of the expression (true or false). We will simply explore both branches independently of their likelihood.

This is a simple and effective technique as long as values are not carried along the branches. If this is the case, then we must restrict ourselves to a bounded domain, or rather, a small bounded domain for these values, to contain state-space explosion. This presents several difficulties. The ADT may not have a domain explicitly defined, or it may be unbounded. This means that, for the purpose of the verification only, we must define a characterization of the domain compact enough to prevent the explosion of the state space. We must also introduce variables of a suitable type to hold the values.

Once we propagate values across communication links, we can use the verification engine to validate also preconditions and invariants of operations, to the extend where all sorts involved have a well defined domain and we have variables to use in lieu of the formal parameters. A postcondition would in turn be used to update the value of the variable.

Let us note that the full validation of preconditions and invariants would require a symbolic evaluation: the state encoding is not always feasible, or practical [4].

In a constraint-oriented perspective, we could also consider the propagation not of values but of subdomains. This could reduce the branching structure since we would not have to consider all values of the domain individually. It requires however an engine that we do not have in PROMELA/SPIN and that is seldom found in verification tools in general.

3.3 Verification

Verification in a LA/PRO model involves several steps. First, the interface language itself can be sort-checked, as in any LIL. Second, we can extract a pure PROMELA model for finite-state verification. This model may in turn be further refined in the following steps.

We need to define a mapping of some ADT's into a finite domain, and provide an evaluation for the operations that manipulate and produce values, when they cannot be abstracted away and replaced by a non-deterministic behaviour.

We must also identify an encapsulation of ADT's into processes. Each process will have a channel dedicated to operation requests, and another for answers.

We need a general "main" programme to drive the verification, that is, which invokes or interacts (through channels) with the high level operations.

Where there are interactions at the implementation level for distributed ADT's, we have two alternatives: either to recursively expand the operations invoked to expose the levels where the interaction occur, or to isolate the operations involved and verify them with a separate, dedicated, "main". This latter method allows keeping the verification

[4]mainly because predicates can include any operators on defined sorts.

model small, at the cost of some extra work. In the first case, the operations may be verified in a more precise context.

Further properties to verify in the PROMELA model, such as temporal properties, must be explicitly coded in that language.

At the LSL level, verification is conducted as usual, first syntactically checking the definition of the traits, and then using the larch proof assistant to check the consistency of the axioms.

3.4 Integration tool

We have shown that the verification process is not straightforward, although some of its aspects can be automated with adequate tools. So far, we have performed our translations in an *ad-hoc* basis. We can however sketch some of the features we would hope to find in such a tool.

Syntax and type checking are the typical tasks that any LIL-checker must perform. Types must be related to sorts and all operations must be compatible with the trait. To perform these checks, we would require a LSL abstraction for PROMELA control structures. However, PROMELA's type definitions can easily be derived from the types of other existing LILs.

The mapping of ADT's to a finite domain is done manually, as well as the implementation of their operations over the finite domain. Of course, this does not need to be done for all the ADT's, but only those interacting with the communication structure. They can be identified mechanically.

The generation of the main programme is also a manual process, as we need to define how the main level operators interact. The refinement process, where required, is also manual.

4 Experimentation

These concepts have been developed on the basis of an informal integration of LARCH and PROMELA studied as part of Pino's thesis [Pino93]. In that work, *variable declaration tables* defined an equivalence between the sorts of the relevant traits and PROMELA builtin types, i.e. the domain mapping. Operators were transformed into messages types exchanged by processes, or simply flagged as comments. Assertions were inserted in the code where specific checks need to be made, to validate the requirements. Verifications were performed completely separately in the LARCH and the PROMELA/SPIN universe.

This experimental work has been done in a narrower focus with regard to integration than has been described. Its purpose was the capture and validation of customer requirements. Nevertheless, it has given us quite valuable insight on the problem of formal integration of the two languages, and the issues of the simplification (i.e. size reduction) of the PROMELA model, to make it amenable to automatic verification using SPIN.

5 Conclusions

We have described the integration of two different formal methods in a unified framework. We have also sketched how it can be used in a practical setting. We have found the integrating two well established formal methods gave us a more comfortable environment then using similar dual modeling languages such as SDL or LOTOS.

We have not yet developed the tools necessary to support the integration, creating interfaces in an *ad-hoc* basis. Nevertheless, we are studying the verifications and transformation that such a tool should make, with a critical eye on the abstractions which can simplify the PROMELA model. There is also a need for a methodology to use the tools and conduct the integration. This is a general problem for formal methods.

We have found so far the LARCH–PROMELA combination to be quite adequate for our purposes. We have used these tools to validate the requirements of a more sophisticated telco security problem [Pino93]. We are pursuing our investigation of their use in the formalization of delegation activities.

Acknowledgments
Parts of the problem of integrating LARCH and PROMELA/SPIN have been studied by Pino as part of his master's thesis research. The notion of integrating promela as a larch interface language was suggested by J. Wing.

References

[Chen89] Chen (Jolly). – The Larch/Generic interface language. – 1989. S. B. Thesis, Department of Electrical Engineering and Computer Science, MIT.

[Cooke92] Cooke (J.). – Editorial - Formal Methods: What?, Why? and When? *The Computer Journal*, vol. 35 (5), 1992, pp. 417–418.

[Ehrig et al.85] Ehrig (H.) et Mahr (B.). – *Fundamentals of Algebraic Specifications I – Equations and Initial Semantics.* – Springer-Verlag, 1985.

[Grégoire93] Grégoire (J-Ch.). – Management with Delegation. *In: IFIP'93, AIPs Techniques for LAN and MAN Management, Paris, France*, pp. II/13–II/21.

[Guttag et al.93] Guttag (John V.) et Horning (James J.), editors. – *Larch: Languages and Tools for Formal Specification.* – Springer-Verlag, 1993, *Texts and Monographs in Computer Science.* With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[Hoare85] Hoare (C.A.R.). – *Communicating Sequencial Processes.* – Prentice Hall, 1985.

[Holzmann91] Holzmann (G.). – *Design and validation of computer protocols.* – Prentice Hall Software Series, 1991.

[Pino93] Pino (L.). – *A Formal Method for Modeling and Analysis of Requirements for Software.* – Master's thesis, INRS-Telecommunications, April 1993.