A SPIN-based Model Checker for Telecommunication Protocols*

Vivek K. Shanbhag and K. Gopinath†

January 29, 2001

Abstract

Telecommunication protocol standards have in the past and typically still use both an English description of the protocol (sometimes also followed with a behavioural SDL model) and an ASN.1[6] specification of the data-model, thus likely making the specification incomplete. ASN.1 (Abstract Syntax Notation One) is an ITU/ISO data definition language which has been developed to describe abstractly the values protocol data units can assume; this is of considerable interest for model checking as ASN.1 can be used to constrain/construct the state space of the protocol accurately. However, with current practice, any change to the English description cannot easily be checked for consistency while protocols are being developed. In this work, we have developed a SPIN-based tool called EASN (Enhanced ASN.1) where the behaviour can be formally specified through a language based upon Promela for control structures but with data models from ASN.1. An attempt is also made to use international standards (for example, X/Open std on ASN.1/C++ translation) as available so that the tool can be realised with pluggable components. We have used EASN to validate a simplified model of RLC (Radio Linc Control) in the W-CDMA (3G GSM) stack that imports datatypes from its associated ASN.1 Model. In this paper, we discuss the motivation and design of the EASN language, the architecture and implementation of the verification tool for EASN along with an example usage and some preliminary performance indicators.

Keywords: Model Checking, SPIN, ASN.1, Telecommunication protocols, State Compaction, Promela, Incremental Hashing

*Thanks are due to Nokia Research Center, Helsinki for funding this work under SID project 99033. We thank Ari Ahtiainen and Markku Turunen of NRC for their initial project formulation and some key ideas in the software engineering aspects, Dinesh Shanbhag for helping us understand the ASN.1/C++ standard, and Matti Luukkainen for many suggestions and criticisms.

†CSA Dept, Indian Institute of Science, Bangalore, 560 012 INDIA; {vivek, gopi}@csa.iisc.ernet.in

1 Introduction

Next generation protocols for mobile devices have become very complex and it is becoming increasingly difficult for standards bodies to be sure of the correctness of protocols during the standardization process. This has become an impediment in defining new standards. What one needs is a way of specifying an evolving protocol and have some confidence that, at a certain level of abstraction, the protocol is consistent inspite of modifications.

There are languages like Promela that can be used, but their data structuring capabilities do not match those that are used in telecommunication protocols. ASN.1 (Abstract Syntax Notation One) is a widely used data definition language in telecommunication protocol specification. It will help the standardization process if a model checker could be augmented with ASN.1 data modelling capabilities to check correctness of interim versions of a protocol before establishing a standard. Inspite of prototypes that are built, they often cannot exercise all aspects of a protocol, especially those that are evolving.

In addition, due to the presence of mechanisms such as subtyping, etc in ASN.1, model checking can be more effective as unreachable parts of the state space that could be introduced in simpler data models in other languages need not be considered.

Hence, we have designed a language called EASN (Enhanced ASN.1) that combines the control structures of Promela with the data definition capabilities of ASN.1. We present our verification tool for EASN and its architecture, followed by an example usage. We base our implementation on SPIN, to benefit from its many capabilities.

1.1 Why ASN.1?

ASN.1 separates data modelling into abstract and transfer syntax. The abstract syntax only specifies the universe of abstract values that can be assumed by variables in the model without any concern for how they are mapped to a particular machine, compiler, OS, etc. Hence from the point of view of model checking, an abstract syntax constrains the state space as much as possible IF there is a mechanism by which a system state vector can be encoded with exactly only the possible values of its constituent substates. The latter is a chief feature of the state compaction infrastructure that has been developed for the EASN system described here. This is equivalent to model checking with abstract data modelling that does not require examining unreachable parts of the system state space introduced due to lack of subtyping, etc. ASN.1 has a subtyping feature with a well developed notation for expressing constraints. Note that data here actually means the control data in the protocols and hence our concerns are different from those approaches that exploit symmetry, etc. While TTCN, the test language in ISO/ITU communities, uses ASN.1, our attempt is to marry ASN.1 with a well known model checker such as SPIN.

A significant number of protocols (in the ISDN domain, for example) are still specified using cross-array for grouping bits. Experience shows that the cost in terms of tests and validation of such informally described protocols is significantly high. Similarly, the transition from IPv4 to IPv6 is likely to be long drawn out due to the interoperability testing that could have been much reduced with a notation similar to ASN.1 that has support for extensibility.

For details about the ASN.1 language, please refer to the appendix.

1.2 Why SPIN?

SPIN is an effective model checking tool for asynchronous systems, especially designed for communication protocols. The design of control constructs in Promela has been based upon those in SDL, a language that has been used to specify communication protocols since '70s. Nondeterminism and guarded commands in Promela makes it convenient to express behavior of communicating protocol entities. The model checking SPIN system[1], which uses Promela, has many capabilities like deadlock detection, validating assertions, system invariants, detection of non-progress cycles and livelocks, and establishing LTL properties. Algorithms that effect substantial space and time savings, like bit-state hashing, on-the-fly model-checking and partial-order reduction have been incorporated into SPIN. Hence, modifying the SPIN system to handle ASN.1 has been a design goal.

SPIN has a **simulator** that randomly checks only a portion of the state space and also a (generated) **validator** that can attempt to exhaustively check the state space of the system or can use techniques like bit-state hashing to check a substantial portion of the state space with a fairly high level of assurance. Our EASN system also has these components.

1.3 EASN Language

ASN.1 can be used to define the datatypes and constant values in an application. Promela, however, is a complete language with a set of basic data types and typedef construct to help users compose datatypes, and a set of control constructs that can be used to define the behaviour of protocol entities.

The EASN Language ***replaces*** all the datatyping capabilities of Promela with ASN.1. Hence, none of the data types of Promela are retained in EASN, except the *chan* construct. Thus basic datatypes of Promela, namely, *bit*, *bool,* *char*, *short* and *int,* as well as related constructs, *unsigned*, *bitfields, typedef* and the *mtype* declaration do not form part of EASN. Channel definition syntax and the capability of defining arrays of channels is retained as there is no similar construct in ASN.1. Defining arrays of other types through the same syntax is, however, disallowed in EASN (as the sequence-of construct in ASN.1 provides this functionality in EASN). We need to now identify as to what subset of the complete ASN.1 Language is incorporated into EASN. We do this in the next section.

As ASN.1 has a far more richer and expressive datatypes compared to Promela, EASN needs to overload the semantics of many of the operators of Promela, so as to support a natural set of operations on data. In addition, we have also augmented the set of operators as necessary. In the first version of the language and implementation, only such operator overloadings and new operators have been included as are necessitated for functional completeness. For example, through operator overloading, convenient syntax is possible to support operations like set union, intersection, difference, negation; or concatenation of strings or of two sequence-of objects of the same type, etc. However, set operations were not included as we have provided a *foreach* construct that loops through elements of certain data types. They could be included later based on the experience of using the language.

In brief,

> EASN = Promela - {mtype, typedef, bit, byte, bool, short, int, unsigned} + ASN.1 + appropriately overloaded semantics of the existing operators + few new operators.

1.4 Related Work

One interesting work relating to language design and protocol verification using the SPIN infrastructure is that of Promela++ at Cornell University[10]. Additions to the Promela language were made to make the resulting language suitable for expressing user level network protocols for high performance computing. If Promela++ is compiled with the verification option, it can do model checking. If compiled with the code option, just like YACC, it produces protocol code using 'actions'. However, the code in the 'actions' is not subject to verification through the SPIN system as it is written in C.

SPIN does various kinds of state compaction and, in EASN, we have a comparable mechanism for most of them that perform atleast as well in space. But some are unnecessary in EASN. Geldenhuys and Villiers[9] also attempt state compression in SPIN along similar lines as ours but by adding a simple construct to the Promela Language but in an *ad hoc* fashion with restrictions. For example, different orders of process activation along different execution paths are forbidden in their approach as much of the state component placement is done statically. The ranges of variables must start at zero. We do not have such restrictions.

1.5 Outline of Paper

In the next section 2, we give a brief overview of the EASN system that has been developed. Later sections present more details of each subsystem. The section 3 presents details about the language design and constructs present in EASN. The section 5 discusses the relevant aspects of the SPIN implementation necessary to understand our modifications and then discusses the EASN implementation in some detail. The last section presents an example run in both EASN and SPIN and then presents performance indicators. Finally, we end with conclusions and future work.

2 EASN, the Verification Tool

2.1 Encoding State Efficiently

SPIN represents state quite efficiently but, for reasons of alignment, etc, allows padding and other extraneous matter in the state vector. Since our system uses ASN.1 data models, we can require that all variables be as constrained as possible in the space of values that they can take through the use of subtyping. For example, if an integer variable takes values from 8..15 only, we can represent the state of that variable in 3 bits. Further, if there are only two variables that

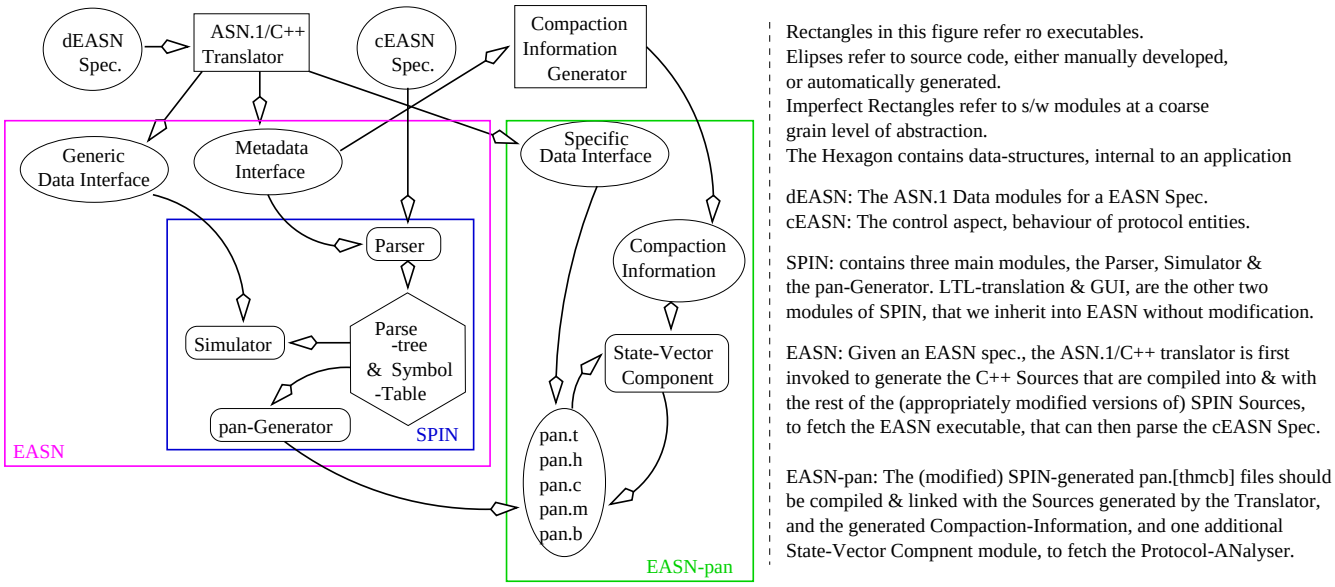

Figure 1: The EASN system

are constrained to be between, say, 5..7 and 3..7, there are only 15 possibilities and both can be represented in only 4 bits instead of either 2+3 (5 bits) or worse 3+3 (6 bits)[1]. Similarly, if an ASN.1 datatype assumes only integer values 5, 7, 11 and 13, only 2 bits of space is needed in the linearised state-vector for objects of this type.

Our design for EASN, therefore, has a critical facility called the state compaction infrastructure that views the state space of the system as a multi-dimensional array (with one dimension for every component of the state of EASN), and consequently, every state of the system, as a point in this multi-dimensional space. We use a column-major linearisation. Note that a row-major linearisation is not useful as the number of components that comprise the system keeps going up and down as the system evolves. In addition, this design enables incremental computation of state hash values (see section5.4).

2.2 Brief Outline of the EASN Implementation

SPIN is open source. We intend EASN to be open source too. Parsing ASN.1 is not easy and we wanted to avoid doing it ourselves, if we could. Nokia Research Center (NRC) has an ASN.1 parser (BEX) for the draft version of the ASN.1/C++ standard that we could use. We needed to be able to use the NRC parser, while not compromising the open source goal. We have been able to use the X/Open ASN.1/C++ translator std[7] to architect the tool so as to enable other users besides us and NRC to realise it by plugging in any X/Open compliant ASN.1/C++ translator into the system. A block diagram is given in figure 1.

BEX is not completely conformant to the NMF standard. Hence there is need for an interface to BEX-generated C++ through a small module that encapsulates the requirement of the parser and the simulator modules. This requirement is encapsulated into a thin layer of software that enables the requirement from these modules to be met when the standard-conformant ASN.1/C++ translatoris replaced with BEX.

An EASN system specification (to be simulated/verified) consists of two *compilation units*. One contains all the ASN.1 modules (the dEASN spec.) that is parsed by the ASN.1/C++ translator to generate the C++ sources. The other compilation unit contains the *behavioural* specification of the protocol entities (the cEASN spec.) that is parsed by the EASN parser (a modified Promela parser, derived from SPIN). It is the variable declarations in the cEASN spec that ties it to the dEASN spec as their types are defined in the ASN.1 modules. The EASN parser *imports* all the relevant information regarding a type, from the generated C++ source, by querying its meta-data interface. This information is then stored in the internal symbol-tables of EASN.

[1]Experienced ASN.1 users may note that such an encoding is even better than the often very compact PER encoding.

2.2.1 The EASN Simulator

The EASN simulator (section 1.2) (a modified Promela simulator, derived from SPIN), besides requiring the information generated by the parser, requires to access data values and modify them through permitted operations. However, since the simulator engine has no knowledge of the specific ASN.1 types that might be used in different EASN specifications, these data operations must be carried out using the generic data interface that supports operations on objects of types *a priori* unknown. The ASN.1/C++ translator also exports such a complete functional interface to access the values held in objects of ASN.1 types.

2.2.2 The EASN Validator

SPIN, and EASN too, generates a set of files `pan.[chmtb]` that are compiled together into a model checking executable (section 1.2). Some of these files, for example .h file, define structures corresponding to the various *proctypes* and *queues* that comprise the system, and the *state* structure. Components of these structures, together, form the state of the system being analysed. We shall refer to the state of the system as the state of SPIN, to differentiate it from what we shall later call the state of EASN.

The state of SPIN is kept in one place in memory, but two sections of code in the generated `pan` files view it differently. The code in `pan.[mb]` corresponds to the transitions that take the system from one state to another, in the forward and backward directions respectively. This code views groups of components in a structured manner, either as some *process* or *queue* structure, or some *global-variable*. However, the code in `pan.c` that has to do with constructing, modifying, maintaining, storing state (into the hash table / or on stack) and comparing for equality views the very same state of SPIN as a block of memory without any further structure. This makes for a highly optimised implementation of the SPIN validator but, in the context of EASN, where many of the components of the state of EASN need to be C++ objects, this two-views-of-the-same-memory is problematic.

The state of EASN, therefore, is organised differently. We *encapsulate* every *actual* component of the state inside an object of type MSVComponent (Minimal-State-Vector-Component, a C++ template type). The state of EASN, then, is simply an array of such encapsulated objects. This representation of the state of EASN is useful for code in `pan.[mb]`. Since the state manipulating code in `pan.c` needs to view the state as a contiguous chunk of memory, we also maintain a ***consistent,*** *linearised* representation of the state of EASN. The consistency is guaranteed by the functionality of the *encapsulating* class.

In order to play its role, the encapsulating class needs to know some information regarding the type of the object that it encapsulates. This information, for every type, is made available through a function call interface in the state compaction information module that is automatically generated by a compaction information generator (figure 1).

2.3 EASN System from a User Perspective

The user first uses an available ASN.1/C++ translator tool that confirms to the NMF standards to generate C++ source corresponding to all the ASN.1 modules that together form the dEASN spec. The generated C++ source, that contains both the generic/specific data interfaces and the metadata interface, is then complied using a C++ compiler (g++, for instance) to create (say) asn1 link modules. The generated C++ source would contain some header (include) files and their implementation along with some tool-provided run-time support. The header files are included in the source code for the EASN implementation, and then the EASN implementation is compiled to generate its set of link modules. These link modules are then linked with the asn1 link modules to generate the EASN (executable) tool.

The ASN.1/C++ translator generated C++ source is *compiled into* the executable that processes the cEASN spec corresponding to the dEASN spec used to generate the C++ source. The EASN system then parses the cEASN spec and uses services offered by the metadata interface to validate the types of variables instantiated, their usage in expressions, their compatibility with various operators, and such, to ultimately generate the parse tree and symbol table data structures. This completes the role of the parser. The user can then use the GUI to choose to either simulate the system or to generate the validator.

The simulator module makes calls to the generic interface and some components of the specific interface (only those that export the data access services corresponding to the basic data types of ASN.1). If the user chooses to generate the validator instead, the validator-generator takes control and generates C++ source in the `pan.[hcmbt]` files, similar to SPIN. An additional intermediate step requires that another program called the Compaction Information Generator (*cigen*, for short), using the metadata interface generated by the ASN.1/C++ translator, generates the Compaction Information that has to be linked with the `pan` files. Finally, all these generated source has to be compiled and linked

with the asn1 link-modules, and with the new svcomp (State Vector Component) module, to generate the *pan* (the protocol-analyser). Control can then be handed over to *pan*, the output of which can be studied.

3 The EASN Language Definition

As discussed earlier, an EASN specification consists of two parts: dEASN and cEASN. The ASN.1 modules that comprise the dEASN are expressed in a subset of the ASN.1 syntax and semantics defined in the X.680 series[6] set of ITU-T Recommendations.

The dEASN spec only defines a certain set of types and constant values. The objects of these types are defined in the cEASN spec. The behavior of the communicating entities that comprise the system is expressed using the control constructs of EASN. The state of this system comprises of the various global objects and the process-local states. These objects (either created as variables of some type defined in the dEASN spec or imported through the use of constants defined there), at the most detailed component levels, would contain objects of the ASN.1 basic data-types. In EASN, the allowable set of ASN.1 basic data-types are the ones listed in section A.1. Similarly, the manner in which they are composed to form the complex higher-level objects are listed in section A.2.

Below, we only describe the constructs that can be used in the cEASN spec. The description assumes knowledge of the Promela language syntax and semantics and describes the control constructs of EASN in comparison. Certain features of Promela are disallowed; certain operators and control constructs are added to Promela; and lastly, certain constructs and operators in Promela are overloaded. The justification, motivation and design rationale for each of these modifications are given below. The linkage between the types and constants used in the cEASN spec and the definitions for those in the dEASN spec is through references to them in the cEASN spec, as described below.

3.1 Variables

All variables in the cEASN spec need to correspond to some type in the associated dEASN spec. The syntax for variable declarations is:

'typeReference variableName [, variableName]'*

The *typeReference* above is specified as *Module-name::Type-name*. This fully resolved type name completely identifies the type definition. We would like to emphasize here, that, except for constant-values, no ASN.1 syntax is accepted in the cEASN spec. The *typeReference* above must resolve against an exported type in the the dEASN spec. Variables can be created either in the global scope, or in the scope of a proctype. Visibility semantics for these objects is similar to those in the Promela Language.

As an instance, consider the need to declare an array of 5 integers, with the values always between 8 and 21. In the Promela Language, one could write the following declaration, using the value range:

```
byte array [5];
```

In EASN, there will be some portion of this information expressed in the ASN.1 module (say) MyModule, and the rest in the cEASN spec. The ASN.1 module could contain:

```
MyModule DEFINITIONS ::= BEGIN
  ...
  Required-type ::= SEQUENCE (SIZE 5) OF INTEGER (8..21);
  ...
END -- End of the MyModule
```

The cEASN spec would then contain:

```
MyModule::Required_type array;
```

3.2 Operators in EASN

In Promela, objects of various types are *compatible* with most operators in the language. In EASN, however, we constrain the types of objects that can be operated upon by the various operators, so as to allow only semantically correct expressions.

3.2.1 Arithmetic Operators: +, -, *, /, % and unary -

Arithmetic operators, typically, operate on integers. In EASN, an object/expression of types either Integer, Boolean, Null, or Enumerated, can be used on either side of an arithmetic operator (an Object or expression of any other type would be flagged as invalid usage). The operands will then be used to evaluate the (sub)expression; an arithmetic operator will thus use its two (or one) operand(s) to generate an integer result. The two operands would be used as follows:

- An Integer object will evaluate to itself.
- A Boolean object will be upgraded to an integer, either 0 or 1, based upon its truth value.
- Null objects evaluate to 0.
- An Object of some Enumerated type will evaluate to the integer value associated with the enumerator value assigned to it.

3.2.2 Comparison Operators: <, >, <=, >=, == and !=

All objects in EASN are managed through the use of the various functionality exported by the (respective) C++ Classes corresponding to them. The ASN.1/C++ Translation Standard mandates the support for all the above comparison operations between arbitrary objects in the system. The (sub)expression corresponding to this set of operators generates a boolean result. Use of these operators by the EASN specification is evaluated through appropriate calls to this supporting infrastructure.

3.2.3 Logical Operators: ||, && and !

These operators, typically, operate on Boolean (TRUE / FALSE) values. Boolean objects or expressions, or integers as described in section 3.2.1 are valid operands for these operators. Non-zero integer values are interpreted as TRUE, whereas zero or NullType objects are interpreted as FALSE.

3.2.4 Bitwise Operators: &, ^, | and ~

These operators typically operate on bits and collections of them. An integer object can be interpreted as a sequence of (64, 32, 16 or 8, as the case may be) bits. In EASN, therefore, operands of types either BitString, BMPString, OctetString, or Integer (as implied in section 3.2.1) can be used along with these operators. In case the two operands are of different types, then one of them is promoted to the type of the other and the result of the operation is also the same type. The latter one in the ordered list above is converted to the type of the other. Note, that when defining the semantics of the promotion, one needs to specify as to how the 'endian-ness' issue is handled. In EASN, we have aligned ourselves with the ASN.1 view, that the 'bit zero' of the BitString is the least-significant-bit (and similarly, that the 'byte zero' of an OctetString is the least-significant-byte, and so on). Operands of types other than those listed here can not be used with these operators.

3.2.5 Shift Operators: << and >>

These operators are used to shift a collection of bits either to the left or right. In that sense they could have been classified as bitwise operators. However, in EASN, the semantics of these shift operators has been overloaded for certain combinations of operand types.

If both the operands are either BitStrings, OctetStrings, BMPStrings or IA5Strings, then the resultant value is their concatenation, in the above order of genericity. Therefore, if one of the operands is a IA5String value and the other OctetString, the result is of type OctetString.

If, however, the second operand is an integer, then these operators take on their usual semantics of effecting a bit-pattern shift. In this case the first operand must be of either of the string types, or Integer. The return type is the same as that of the first operand.

3.2.6 Indexing Operator: []

This subscript operator can be used to index into an array of channels or into a Sequence-Of objects, or into objects of some string Type. When used to index into objects of type BitString, OctetString, IA5String or BMPString, the resultant expression is an LValue expression of the respective String Type of length 1.

3.2.7 Assignment Operator: =

Use of this operator is one of the mechanisms of assigning values to variables (objects). The other way to achieve this is through the use of the *recv* operator that reads values from a channel and assigns them to the appropriate (receiver) objects specified in the mesg-receive list for the operation. Attempts to assign values to objects using either of these mechanisms has to pass the type-compatibility check: either the types for the *lhs* and the *rhs* are the same, or they should be compatible in the sense described in section 3.2.1.

3.2.8 Channel Operators: ?, !, ??, !!, ? <>, ?? <> and len

In Promela, all the above operators need their lhs operand to be a reference to a channel, except the unary operator 'len' that acts upon channel-references. In EASN, except for the semantics of the len operator, all the above operators mean and behave the same as in Promela. The unary operator 'len' in EASN has additional overloaded semantics as follows: when used to operate on any of the string types (BitString, BMPString, OctetString or IA5String) it evaluates to the length of the operand; when used with a Sequence-Of type of object, the result is the number of elements in it; for a Set-Of object, its cardinality is returned.

3.3 Additional Operators

An interesting aspect of the ASN.1 Language (with no comparable feature in C, C++, or Promela) is that component members of sets and sequences (sectionA.2: these constructs are comparable to the 'struct' facility in C) can be defined as being optional. This means that objects of such types may have certain components missing. This feature of ASN.1 has led us to add a few operands that can be used to ensure that a value to be accessed does indeed exist. These new operators can be used to branch control on the basis of the presence or absence of some component of interest. These operators and others are described below. Additionally, there are the membership query operators to ask if an object is contained in a collection of objects such as a set or a sequence.

present

This unary operator is used to operate on a component of an object. The syntax is *'present varref'*. The *varref* must evaluate to a l-value and must refer to an optional component of some Set or Sequence type object, or to some component of a 'Choice' data-type. The executability of this expression is true iff the referenced component exists and can be read. If the *varref* refers to an non-optional component, such usage is in error, and the parser shall report it as such.

One mode of usage for this expression is to guard all accesses to optional components of objects. An instance of possible usage in an *if ... fi* construct is as follows.

```
if
:: present choiceTypeObject.component1; ...
:: present choiceTypeObject.component2; ...
:: present sequenceTypeObject.optionalComponent; ...
fi
```

absent

This unary operator is the dual of the *present* operator above. The syntax is *absent varref*. It is executable iff the referenced component is not present. This operator is used to transfer control, flagged by the absence of an optional component. When the *varref* refers to a component that is non-optional, such usage is in error, and the parser will point out the error.

annul

Valid use of this unary operator is when it is asked to operate upon optional components of objects. When so used, this operator behaves as a dual of the assignment operator. The assignment operator is used to assign vales to (components of) objects. The annul operator is used to deassign (optional components of) objects. For instance, if we were to assign value 5 to an optional integer component of an object, and later (along the path of control) we were required to affect this optional component to not exist, then *annul varref* is the operation that would be used to achieve this. If *varref* were to refer to non-optional components of objects or to gross objects, then the parser would flag this as an error.

annul makes executable any *'absent varref'* expressions that might be awaiting deassignment of the referenced optional component, just as an assignment (to the referenced component) would make executable any *'present varref'* expressions that might be blocking read accesses to it.

substring

This operator requires the function call syntax since it requires three operators. The *'substring (reference-to-string-value, from-position, number-of-units)'* expression returns an object of the same type as its first argument. The second and third arguments to this call should be integers (or objects promotable to integers, as in section 3.2.1). The first operand can be any of the String types.

3.4 A Note on Subtyping

ASN.1 provides constructs for expressing value constraints (section A.4). Making this information available to the verification tool enables it to function more effectively, using less memory. It is therefore mandated, in EASN, that all objects in the cEASN system must be associated with types defined in the dEASN module that are value and/or size constrained. It is additionally recommended, that they be as tightly constrained as possible, based on all the information that is available with the system-designer.

4 Software Engineering of an EASN Implementation

Given the EASN language, one can now specify all the aspects of a protocol **formally**: the PDU's, their structure (the manner in which the various communicating entities interpret every field), and the behavior of these (protocol) entities. This formality does not help us, however, unless we process such specification to identify various kinds of incorrect, or unintended or unwanted elements of behavior that could be displayed by the system as a whole, or by some subset of its component entities. What we need for EASN are capabilities similar to that of SPIN, given a system specification expressed in the Promela Language.

Given that the EASN Language has been specified as a set of minimal and well-defined modifications with respect to Promela, would it be reasonable to expect that its implementation could be effected through some non-substantial (though not trivial) set of appropriate changes to the SPIN system? Are there features of either ASN.1 or Promela or SPIN that make such an approach fundamentally questionable? If not infeasible, would it also be possible to retain all (or even most) of the many good features and aspects of the SPIN system? What about modifiability, maintainability, upgradability, or just even maturing over a period of time, given that ASN.1, Promela and SPIN are all live, evolving pieces of work, and have matured over a large period of time (upwards of 15 years or so).

We see three main issues: how to insulate oneself from the evolving ASN.1 and SPIN infrastructures, how to avoid the cost of translation or parsing of a very large language like ASN.1 and current non-availability of complete parser/translators for such languages. We discuss them one by one below.

4.1 How to Insulate Oneself from the Evolving ASN.1 and SPIN Infrastructure

ASN.1 is a huge language with many features. If we had to parse ASN.1 as a part of this effort, it would be a major undertaking. Similarly, SPIN has many nice features that, if we had to recreate all over again, would be an even larger task. Fortunately for us, Dr. Holzman, the author of SPIN, chose to make the source freely available (under his copyright). As for the ASN.1/C++ implementation, NRC (Nokia Research Centre, Helsinki, Finland) has an implementation of the draft version of the ASN.1/C++ spec, called BEX. Our EASN implementation architecture uses all this enabling infrastructure to minimise the amount of work required to implement EASN, and at the same time ensure that we are safeguarded from the implications of an evolving infrastructure.

- All our modifications to the SPIN sources are encapsulated in pre-processor conditional compilation flags. This enables us to incorporate any incremental change / bug-fix to SPIN quite easily and fast into EASN.
- In our design, we assume the availability of a standard-conformant ASN.1/C++ translator, rather than the draft version, so that other users, who may not have access to an implementation of the draft could still use our work. In order to use BEX, however, we develop a small module over BEX[11], that exports the interfaces that EASN requires. (The EASN requirements from a ASN.1/C++ translation can be easily met given a standard implementation).

4.2 ASN.1 to C++ Translation

NMF (Network Management Forum 1998) has an "ASN.1/C++ Application Programming Interface"[7] set of standards that propose a translation mechanism into C++, given ASN.1 modules. This standard enables us to avoid having to parse ASN.1 syntax and other related issues. The standard is briefly discussed here. It is formed of two parts, namely, Part 1: Base Classes and Specific Interface and Part 2: Generic Interface.

The C++ Data Model for ASN.1

Corresponding to every ASN.1 module definition, a C++ namespace with the same name is instantiated with a C++ class definition generated for every user defined ASN.1 type in that module. The public interface for the predefined C++ Classes corresponding to all the basic types of ASN.1 is defined to be in the 'ASN1' namespace. All predefined and generated C++ classes are derived from a fundamental Abstract Class called **ASN1::AbstractData**. Also, for each ASN.1 constant, a C++ object of that type is instantiated. The syntactic name-mangling rules to generate the appropriate C++ names for the corresponding ASN.1 names are simple. In addition, for each ASN.1 type, a C++ class is also created to export (though its public interface) meta-data regarding the properties of that type.

There would, therefore, be as many instances of a particular C++ class corresponding to some ASN.1 type as there are ASN.1 constants defined of that type in the ASN.1 module. In addition to this, a C++ class is also created corresponding to the ASN.1 type definition, that exports (though its public interface) meta-information regarding the properties of that type. Such class definitions corresponding to the basic types already exist in the ASN1 namespace. These classes and all classes that are defined corresponding to the various user-defined ASN.1 types are public subclasses of an abstract class called **ASN1::AbstractType.**

Part 1: Base Classes and Specific Interface

This document defines the C++ class mapping for all the basic ASN.1 data types including the ones described in Section A.1. Also the rules followed to generate C++ code for user-defined types in the ASN.1 module(s) using constructs described in Section A.2 and the subtyping mechanism are defined here. In the generated C++ code, public access methods for every component of the SEQUENCE, CHOICE, and SET type are included. The names of these methods are chosen by name mangling those of the corresponding components in the ASN.1 module specification. This interface, called the specific data interface, is exported by the generated classes for the corresponding SEQUENCE, SET and CHOICE type definitions given in the ASN.1 module.

Part 2: Generic Interface

This document defines the metadata interface as well as the generic data interface. For instance, a C++ interface for querying the metadata of ASN.1 types, methods to get the (names of the) enumerators of an ENUMERATED type or those to get the (names of) components of a SEQUENCE type. Querying through this interface could retrieve all the necessary information to enable parsing EASN code that declares variables of ASN.1 types and operates upon them.

The definition of the Generic Data Interface involves exporting a functionally complete interface that can be used to access the values held in objects of arbitrary type confirming to the data model. This interface, for instance, defines C++ classes like CHOICE and StructuredData which provide generic methods to access the components of a CHOICE, SEQUENCE or SET object. These methods take the component name as the (string) parameter to provide access to the corresponding component. This generic data interface can be used, along with the metadata interface described above, to discover the composition of data structures and operate on-the-fly. In our EASN implementation architecture, we rely on this interface to minimise our work.

5 EASN System: Implementation

We shall now move over to discussing the implementation of the EASN System. We shall begin by discussing the implementation of the SPIN System as a set of sub-systems: here we use the word sub-system to identify either a set of C-modules, or sometimes a single C-module, or even just a set of functions / functionality within a C-module. We shall then briefly describe the usage of the EASN tool, from the user's view. We then talk about the details of our implementation, with respect to the SPIN description below.

5.1 SPIN and its Subsystems[8]

SPIN is implemented, mostly, in ANSI C. It, therefore, comprises of various include files and link modules. Apart from C, there is (only) the GUI module that is entirely implemented in the Tcl/Tk scripting language. One of our primary tasks, before attempting any modifications to SPIN, was to come up with an abstract view of the SPIN sources that would make it possible to break down our implementation effort into a set of well-defined subtasks, that we could then attempt systematically. Recall from section 4.1, that our implementation of EASN from SPIN was to retain all the good features of SPIN, and also be able to upgrade to future releases of SPIN, with minimal effort[2]. Below, we list the main components of the SPIN system, from the perspective of a designer wishing to modify SPIN.

GUI: This module is implemented in Tcl/Tk entirely, and requires only cosmetic changes when having to function in the context of EASN.

Parser: This module comprises of a handcrafted lexer and a YACC-generated Promela parser, along with the necessary supporting functionality in associated link modules. Like for all languages, the parser parses the input specification, checks for syntactic correctness, and creates internal data structures that capture all the relevant information from the specification to enable either simulation of the system specified, or generation of a validator for it. This was the starting point for our modifications, since we had a clear set of requirements in the form of an EASN language spec, drawn out as a set of modifications to Promela.

Parse tree and symbol table: This module includes the various data structures defined in the `spin.h` file, and the supporting functionality provided in various other link modules. The definitions for some of the data structures involved were slightly modified to reflect the change in the data model of EASN with respect to that of Promela.

Simulator: This module includes all the code that operates upon the fully constructed parse tree and the completely populated symbol table. Its role is to effect a random walk on the reachable subset of the entire state space of the specified system, starting at the initial state. One of the key functions is the recursive `eval` function. Given a node of the parse tree, it returns an integer value on evaluating the expression tree rooted at it, since any basic type in Promela is representable in a 32-bit integer, a key design decision in the SPIN implementation. (Similarly, all objects of higher-order types are also represented, passed as parameters to processes (at their initialisation), or passed into and out of message channels, as a collection of the required number of integer objects). In EASN, our major modification to this module stems from our need to change the function call interface of `eval`.

Validator-Generator: The starting point for this subsystem, similar to the simulator described above, is the information in the parse-tree and the associated symbol-table. It comprises of all the files that begin with the `pangen` prefix. This module generates the *pan* (protocol analyser) files in ANSI C. The generated files (described below) are then compiled together to form the executable protocol-analyser that actually analyses the Promela spec.

Protocol-Analyser: This program conducts a search over the reachable state space of the system described in the Promela spec, using a *user-configured* combination of algorithmic components. The basic state space search algorithmic framework remains broadly the same; however, intermediate steps, like whether to use partial-order reduction methods, whether to use complete state storage or use bit-state hashing, whether to incorporate fairness, safety properties, acceptance cycles, etc. are user provided inputs that are used to select the code fragments that compose the complete search loop. The entire program is one single C compilation-unit, with the rest of the files mentioned below, being `#included` into the `pan.c` file.

[2] A good indicator of how well we have been able to meet this design goal is the fact that when we first began making our modifications to SPIN, we were working with the version 2.5.4 of the sources, and by the time we could get our EASN system tested and stable, we had gone through 4 'upgrading's and at the time of making this release, we are in sync with the version 3.4.1 of the SPIN sources.

`pan.h` This file declares forward prototypes of certain functions that are defined in the `pan.c` files, but may be used before they are defined. More importantly, the structures corresponding to the state of the system (including those of various proctypes and channels defined in the Promela spec) are also defined. For instance, when the Promela spec has a (system-) global or a (proctype-)local variable declaration, the generated `pan.h` file has a member variable (with the same name) of the corresponding C type in the state structure, or in the proctype structure. In EASN, the structure corresponding to these various proctypes, channels, and the state of the system are defined differently. Also, the prototypes of certain functionality had to be changed.

`pan.m` This file contains, as a single list of case options, C statements corresponding to every state transition (**m**ove) in the input Promela spec. Transitions in this file take the system *forward* in its state space search. All of this file is `#included` as the body of a C switch statement. For instance, an original Promela statement that, say, increments a certain (proctype-)local variable is translated into a `case` entry in this file that is a C expression incrementing the value of the corresponding `proctype` member variable. However, just before incrementing, the value of the variable is copied into a *stack trace*. (The next item illustrates the use of the stack-trace). In EASN, the contents of this generated file continue to remain the same. However, since we *import* the support for operating with ASN.1/C++ Objects from the C++ bindings, where any access to some component of a structured object requires us to make a function call, it looks slightly different.

`pan.b` Similar to the `pan.m` file, this file is also a single list of case options with one `case` listed here for every `case` in the file above. Each case item body expects, at the top of the stack trace, the information that it needs to *restore* the state of the system to the point earlier, *before* the corresponding `case` took it forward. This file, therefore, contains all the code that takes the system *backward* from where it is to where it was, before it chose any particular forward move from the `pan.m` file. The stack trace, then, is the information that is generated by the `case`s in the `pan.m` file, and consumed by the code in the `pan.b` file. In EASN, just like for the previous file, the corresponding `pan.b` file needs to work with C++ objects imported from the ASN.1/C++ translation and is therefore different to that extent.

`pan.t` The code in this file contains the functionality that generates the transition matrix that encodes the behavioral semantics of the system. These transition objects refer to the case labels that identify the cases in the `pan.[mb]` files above. The various members of the transition structure are used by the policy code (below). EASN does not modify either the definition of the transition or the way transition objects are created and populated by this `pan.t` code.

State-of-SPIN: We refer to the `now` object of type `State` (whose structure is defined in the `pan.h` file), with all the process and queue objects overlaid onto it, as the State-of-SPIN. Some members of the State structure are book keeping information that is used by the policy code in `pan.c` to make decisions when conducting the state space search. As and when processes and queues are created in the system, space is allocated for them so as to overlay them on the unused trailing portion of *now*. Also, as and when queues become inaccessible or processes reach the end of their computation and therefore can be flushed out of the system, the space that they occupy is recovered but in a strict last-in-first-out fashion. The various functions defined in the `pan.c` file that compress (or compact) *now*, put it into the hash-table (or stack), check for it in the hash-table (or stack), compute various kinds of hash values for it, compare two representations (whether compressed or hash-compacted, or hashed, ...) of *now* for equality, etc. are termed as *mechanisms,* below. The code that makes calls to these mechanism functions and uses the information returned to make decisions, such as whether to move up the search tree or down, is referred to as the *policy* code.

`pan.c`–**Policy:** The main loop of the generated protocol-analyser (or validator) is the policy-code. This main loop requires a hash-store, where reached states are stored. It also maintains a stack, that contains some representation of the various states that have been visited along the path from the initial state of the system up until the current state. This stack is used to detect acceptance cycles, or non-progress cycles. The information in the hash-store is used to avoid re-visiting states that have been explored in the past. This policy code encapsulates the core of the SPIN-algorithms, including detection of assertion violation, cycle detection, partial-order reduction, and LTL-property conformance. To inherit all these facilities offered by SPIN into EASN, therefore, requires care so that we do not tamper with this policy code.

`pan.c`–**Mechanisms:** This code includes the functionality that actually stores the reached-state into the hash-store and/or the stack and checks to see if a particular reached-state has been visited in the past. Based upon various

options that can be selected when generating the validator, and some additional options that can be selected when compiling and executing it, SPIN offers many choices for compaction / compression algorithms including bit-state hashing. All these functions are tightly tied to the particular linearised representation of the State-of-SPIN. In EASN, where we have a different representation for the State-of-EASN, we have to reimplement this functionality. The correctness of our implementation, therefore, hinges upon the correctness of our management of the various representations of the state of the system.

5.2 EASN and its Subsystems

EASN has all the modules of SPIN listed above and a few more (see figure 1). Those SPIN modules that are modified in EASN are described only to the extent to which they differ. The sub-systems of SPIN that are not modified in EASN are not included below, for example, the GUI part, and the `pan.t` file.

Except for our choice to change the representation of the state of the generated validator, the major change in EASN from SPIN is the type system. Hence, in the implementation of EASN, this should translate into modifications to SPIN wherever it deals with variables (and expressions involving variables). Therefore, entire pieces of code that have to do with the handling the semantics of the control constructs of Promela need not be modified at all.

Parser: The lexer has been modified to modify the set of keywords in the language, to recognise the ASN.1 value-notation for constant values and to identify ASN.1 type-references and value-references. On encountering an ASN.1 typename, the parser queries the ASN.1 metadata interface for the complete set of attributes associated with that type and updates its symbol table accordingly. Notice, therefore, that only those types from the ASN.1 module that are actually referred in the cEASN spec, are actually imported into the symbol tree.

Parse tree and symbol table: The definitions of the structures of some of the types that build the symbol table and the expression tree in the parse tree have been modified. For instance, the optionality and the default value attributes of the component of structured types need to be represented in the `Symbol` structure. Similarly, in order to enable the compatibility check for operators and operands, the `Lextok` structure has been enhanced.

Simulator: One of our modifications to this subsystem is to the `eval` function that evaluates an expression in the given state of the system. In EASN, this function returns a (pointer to a) dynamically allocated object of type `AbstractData`. This makes it convenient to implement the operand type upgrade associated with some of the overloaded operators. Another modification has to do with passing parameters to process invocations, or passing messages through channels using `send` and `recv` operators. In EASN, since the C++ types that are referred through their ASN.1 names are already compiled into the (simulator) application, entire C++ objects can be handled without, as in SPIN, having to compute the number of integer objects required to represent types, whose objects need to be passed as parameters (while invoking process instances through the run statements), or through channels.

Validator-Generator: EASN also generates all the files that SPIN does, and at the level of abstraction in the previous section, the contents of the EASN generated pan files are similar too. We discuss the details below.

Protocol-Analyser: The SPIN generated pan files are complete in so far as they do not need to link with anything more than the standard C Library. EASN generated files, however, need to link with run time support, ASN.1 support, state vector component support, GNU multi-precision arithmetic, and the compaction information. All of these linked together to produce *pan*, the protocol-analyser.

State-of-EASN: The *now* object in SPIN generated validators encapsulates the State-of-SPIN. As presented in the section on SPIN, the code in the `pan.[mb]` files deals with the structured objects that overlay onto `now` (and the other members of the `State` structure that correspond to global variables in the Promela spec). Whereas the *policy* code from `pan.c` uses the book-keeping information to store the state of the search, the *mechanisms* code views *now* merely as a sequence of contiguous bytes. This 'multiple-views-of-a-single-chunk-of-memory' tends to become unmanageable in the context of EASN, where the objects that comprise the state of the system are C++ objects that can have virtual-table pointers, RTTI, and other inaccessible private components. The State-of-EASN, therefore has two representations: one which lends itself to be used by structured users such as the code in the `pan.[mb]` files and the *policy* code, and the other, the linearised version, which is intended for use by the *mechanisms* code. By ensuring that these two representations are mutually consistent but only at points where the control changes from the structured-view code to the linearised-view code, we benefit by

implementing *incrementally* both the consistency updates as well as the computation of the hash-values for the linearised versions. Consequently, the *mechanisms* code in the EASN generated analysers is much simpler than that for SPIN.

svcomp-Module: In EASN, the consistency between the two representations discussed above is achieved by *encapsulating* every component of the State-of-EASN into (a publicly derived subclass of) the MSVComponent (Minimal-State-Vector-Component) template class whose functionality *incrementally ensures* consistency. We call this facility the *compaction infrastructure*. In order for the MSVComponent class to execute its responsibility, it requires to know two pieces of information for every type that it encapsulates: the cardinality c of the value-set represented by the type, and a mapping from this value-set onto integers in the range: $0..c$. This information is required to be available through a function-call interface. If the value of c for every type that is imported from ASN.1 is representable using 32-bit integers, then a more efficient implementation of this module called *lsvcomp* (light-svcomp) can be used instead. The call interface is as follows:

- `MP_INT * EASN_GetCardinality (Type *)` returns, for a given type, the cardinality of its value-set.
- `MP_INT * EASN_GetIndex (Type * object)` returns, given an object of some type, the index of that value in the value-set of that type.

This module has been implemented through the use of the **G**NU **M**ulti-**P**recision (GMP) package. The `MP_INT` type in the prototypes above is a type exported by this GMP library. In case of the lsvcomp module, the return types for the above functions has to be `mp_limb_t` that is just wide enough.

cigen This sub-system generates the implementation for the two functions `EASN_GetCardinality` and `EASN_GetIndex` for every type in the ASN.1 module. In our current implementation, it has been implemented as a stand-alone tool that uses the metadata interface to generate these functions for all the types in the ASN.1 module definition, but it could also be implemented as an additional link module of the EASN system, thereby generating the *compaction information* only for those types from the ASN.1 module that are imported into the cEASN Spec.

Compaction-information: This is the set of functions that are generated by the cigen utility above and needs to link with the generated `pan` files to generate the required validator.

panrts-module: This module is required because of the automatic type upgrading semantics of some of the operands of EASN. Also, there is a gap between the set of operations that are supported by the C++ classes corresponding to the basic datatypes of ASN.1 and the operators that we support on them in our EASN Language definition. This module fills the gap.

`pan.h`**:** The structures defined in this file are similar to those of SPIN, except that members of the structures that actually correspond to variables in the cEASN spec are encapsulated into the MSVComponent class (or one of its subclasses). For instance, if the finite state automaton corresponding to a proctype has 19 local states, in the SPIN generated `pan.h` file, the corresponding structure has a member, which identifies the state a particular process, of type `char` to store this information. EASN generates, for the same purpose, a member with type `iSVComponent<20>`. Similarly, an cEASN variable of type `asn::Integer` causes the generated `pan.h` file to have the type of the corresponding member variable to be `SVComponent<asn::Integer>`.

`pan.c`**-mechanisms:** The entire set of these mechanisms that work with the linearised view of the state of the system have been re-implemented for EASN-generated validators. For instance, the default compression algorithm used in SPIN `pan` is unnecessary for linearised versions of the State-of-EASN. As another example, consider the implementation of the hash compact version of the compression routine in SPIN `pan`. It computes the compressed version of every process and queue state in the system separately from the global state of the system and individually hashes these to generate much smaller id's which are then used to compose a new (far shorter) representation of the reached-state of the system, which is then stored into the hash table. This two-level hashing is implemented incrementally by recording the necessary book-keeping information in the *compaction infrastructure*, at the point of installing and un-installing components of the State-of-EASN. Contiguous subsets of these system components are demarcated as belonging to either the global accessible state, or of a particular process or queue and separate linearisations for these sets of components are also maintained, incrementally and consistently. These representations are then used to directly compose the next level representation of the reached state that is then put into the hash-table.

5.3 Ensuring Consistency - Incrementally

The *compaction infrastructure*, equipped with the necessary *compaction information*, views the state space of the system as a multi-dimensional array (with one dimension for every component of the system), and consequently, every state as a point in this multi-dimensional space. Another representation used is a column-major linearisation of this multi-dimensional array.

Since, as also is the case with the State-of-SPIN, the number of components that compose the system-state can increase (if new processes or queues are added) and decrease (if the last process reaches the end of its computation), the compaction infrastructure has to also recompute its mapping from the multi-dimensional array to a linearised representation. Only the column-major linearisation is useful as the row-major linearisation cannot handle varying numbers of processes. The compaction algorithm, therefore, associates a weight along with every component of the state-of-EASN at the point of installing it and uses this weight to appropriately increase or decrease the impact of the change in the value of this component on the linearised representation. We assume here that the system comprises of components numbered from 0 through n, and we use the prime notation to denote the *new* value of any entity.

c_j**:** The *cardinality* of the value-set associated with the type of component j of the system.

i_j**:** The *index* of current value of component j of the system into the value-set associated with its type.

$w_0 = 1$**:** The *weight* of the first component of the system is 1.

$w_j = \prod_{k=0}^{j-1} c_k$**:** The *weight* of component j of the system is the product of the *cardinalities* of the previous $j-1$ components of the system.

$L = \sum_{j=0}^{n} w_j i_j$**:** The column-major linearisation of the system is the sum of the products of the *weight* of the component and its *index*, over *all* the n components of the system.

$\delta L = w_j(\acute{i}_j - i_j)$ **:** The incremental impact of the change in value of some component (say, j) is the product of its *weight* and the difference in the *indices* of its new-value & old-value.

Whenever the system moves from one state to another, only those few state components that are responsible for the change determine the update on the linearised representation. The compaction infrastructure updates the representation incrementally which is then stored into the hash-store when an exhaustive search is conducted.

5.4 Incremental Hashing

There are functions in SPIN-generated `pan.c`, called `d_hash`, `r_hash` and `s_hash`, that generate (1 or 2) 32-bit hash values by using the complete octet-string (representing a part or the full state) that has been passed to them. SPIN generated validators use these functions to compute hash values corresponding to reached states that are represented by *now*. The user can specify any of a set of 32 hash constants to be used by these algorithms. Typically, only a few of the components of the system are responsible for its change in state at any given point in its evolution. Hence, an incremental computation of the hash value can improve performance which is done through our *compaction infrastructure*.

SPIN uses polynomial arithmetic to compute a 32-bit intermediate quantity (that is further used to generate the hash-values), by performing a fast division operation on the *now* state vector using the chosen (32-bit) hash constant as the divisor. In EASN, we use integer division on the linearised representation of the reached state, again using the 32-bit hash constant. The 32-bit remainder, thus generated, is used to generate the hash values, just as they are in SPIN.

We introduce one more attribute that is maintained for every component of the state of the system:
$r_j = w_j \bmod H$. The *remainder* after dividing the *weight* by the chosen 32-bit Hash-Constant, H.

Below, we discuss the incremental computation to generate this hash-value $\acute{R}$ for a new state that resulted with a change in the system component j, given that the hash-value for the old system state was R.

$\acute{R}$
$= \acute{L} \bmod H$
$= (L + \delta L) \bmod H$
$= ((L \bmod H) + (\delta L \bmod H)) \bmod H$
$= (R + w_j(\acute{i}_j - i_j) \bmod H)) \bmod H$
$= (R + r_j \delta i_j) \bmod H$

```
1  /* mtype = { msg0, msg1, ack0, ack1 }; */
2
3  chan sender   = [1] of { asn::MtypeAbp };
4  chan receiver = [1] of { asn::MtypeAbp };
5
6  inline recv(cur_msg, cur_ack, lst_msg, lst_ack) {
7     do
8     :: receiver?cur_msg -> sender!cur_ack; break
9     :: receiver?lst_msg -> sender!lst_ack
10     od;
11 }
12
13 inline phase(msg, good_ack, bad_ack) {
14    do
15    :: sender?good_ack -> break
16    :: sender?bad_ack
17    :: timeout ->
18       if
19       :: receiver!msg;
20       :: skip /* lose message */
21       fi;
22    od
23 }
24
25 active proctype Sender() {
26    do
27    :: phase(msg1, ack1, ack0);
28       phase(msg0, ack0, ack1)
29    od
30 }
31 active proctype Receiver() {
32    do
33    :: recv(msg1, ack1, msg0, ack0);
34       recv(msg0, ack0, msg1, ack1)
35    od                                   (1)
36 }
```

```
1  Easn DEFINITIONS ::=
2  BEGIN
3
4  ...
5  MtypeAbp ::= ENUMERATED {
6     msg0, msg1, ack0, ack1
7  }
8                                   (2)
9  END
```

```
...
State-vector 24 byte, depth reached 9, errors: 0
     12 states, stored
      3 states, matched
     15 transitions (= stored+matched)
      0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)                (3)
...
```

```
...
State-vector 12 byte, depth reached 9, errors: 0
     12 states, stored
      3 states, matched
     15 transitions (= stored+matched)
      0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)                (4)
...
```

1: The cEASN Spec. 2: The dEASN Spec. The original Promela Spec. can be recovered from the cEASN Spec, by uncommenting line # 1.
3: The SPIN-pan output. 4: The EASN-pan output.

Figure 2: ABP in SPIN and EASN

.

Notice that in the context of the *lsvcomp* implementation, all the four quantities on the RHS of the equation above, namely, $R, r_j, \delta i_j$, and H, are all 32-bit operands. This allows for an efficient implementation of this incremental hash-value computation.

More interestingly, under certain combination of SPIN options (-DBITSTATE & -DSAFETY) while compiling the validator, it turns out that the linearised version of the state is neither stored on the stack nor in the hash-table, which reduces the burden of generating, computing and maintaining it, since we can generate the hash values corresponding to the reached state incrementally, directly available from the *compaction infrastructure*.

However, an associated fact about this incremental hash-value computation scheme is that the value H above needs to be identified at compile time itself, since it is used by the constructors of the object wrappers in the *compaction infrastructure*, some of which are called before the main function gets control. This is in contrast with SPIN generated validators, which can be compiled once and then executed many times for various values of H.

5.5 Correctness of Implementation *vis-a-vis* SPIN

In deriving an EASN implementation from SPIN sources (given the above indicated modifications), we identified the following invariant that could be a necessary and sufficient condition to convince oneself that neither the simulator engine of EASN, nor the state-space exploration engine of the generated validator gives different results than SPIN:

> Given a Promela spec. s and a cEASN spec. e, derived from s by changing all its variable types to equivalent ASN.1 types (defined in an associated ASN.1 module, appropriately imported into EASN): A. Simulation runs of SPIN over s and of EASN over e should show identical selection sequence of state-transitions, for the same seed value; B. The sequence in which the reachable states of the system are visited by the generated validators (by SPIN for s and by EASN for e) must be identical (for exhaustive state-space searches), with/without partial-order reduction or never-claims.

EASN preserves this invariant for all the test cases we have tried. EASN generated validators, therefore, will not likely to have any more bugs than were inherited from the SPIN sources that were modified to craft EASN.

M	Options	Validator	Time	Memory	States	Transitions	State, size	Depth
S	SAFETY	SPIN	0.32	2.825	13380	18550	112	2380
N		EASN	0.97	2.518	13380	18550	64	2380
O	(plus)	SPIN	2.75	9.134	80486	271814	112	9999
O	NOREDUCE	EASN	11.06	7.495	80486	271814	56	9999
P	(plus)	SPIN	1.61	0.929	79536	268903	112	9999
Y	BITSTATE	EASN	6.57	0.929	80486	271675	0	9999
L	SAFETY	SPIN	0.04	1.493	97	97	196	108
E		EASN	0.06	1.493	97	97	116	108
A	(plus)	SPIN	0.87	4.014	15779	58181	196	108
D	NOREDUCE	EASN	2.24	3.399	15779	58181	92	108
E	(plus)	SPIN	0.40	0.929	15779	58181	196	108
R	BITSTATE	EASN	1.71	0.929	15779	58181	0	108

Table 1: EASN vs SPIN: time and memory performance

6 Results and Conclusions

6.1 RLC/ABP Examples

We have used EASN to validate a simplified RLC protocol in the W-CDMA GSM stack. It has a smaller state than SPIN due to the use of the subtyping information by the state compaction infrastructure for EASN. Further details of the performance of EASN will be submitted to the FMICS workshop. Due to its length, we present a much simpler ABP protocol in figure 2. Note that the state vector for EASN is half the size of SPIN's.

6.2 Performance of EASN vs SPIN

We present some preliminary performance comparisons of EASN-generated validators for specs derived from some Promela specs (snoopy and leader election)) in the SPIN/Test database, with those generated by SPIN (table 1). We compare the runs of the validators generated by both SPIN and EASN, under compilations with three sets of options, SAFETY, SAFETY and NOREDUCE, SAFETY, NOREDUCE and BITSTATE. Time is denoted in seconds, memory in megabytes, and the state(-vector) size in bytes.

Since we have made only the minimal modifications to a Promela spec to derive the corresponding cEASN spec, we expect to see (except for the bitstate runs) the number of states detected and the number of transitions explored must be the same for both the SPIN generated validator and the EASN-generated ones. Note that this is indeed true.

The number of bytes required to store the linearised State representation is smaller for EASN, than that for SPIN, also as expected, thereby reducing the memory requirement at run-time, again except fpr the bitstate runs, where both the validators use the same number of bits to store a single state. (The reduction in memory is much lower than expected, since, we believe we still have some unplugged memory leaks in our generated validators.)

Also as expected, the run-time is higher than that for SPIN-generated validators. However, the increase in run-time is many-fold, being much higher than what we anticipated. Profiling has revealed the following major contributing factors for this deterioration in the run-time.

C++: In SPIN, an integer object defined in the Promela spec translates to an integer member of a C structure, and access to it is not protected, unlike in EASN, where that compares to an `asn::Integer` type object, that (typically) securely encapsulates the actual integer (that holds the value of interest), as a private/protected member. Therefore, any Promela expression that involves use of this integer object, translates into a C++ member function call (that may or may not be inlined).

C++ Constructors and Destructors: We observe that a substantial portion of the run-time is spent initialising and destroying temporary C++ objects on the stack. Since we wanted to prevent having to incur time penalties related to allocating and deallocating objects on the heap, our generated code uses compiler temporaries, but even then one cannot avoid the constructor/destructor cost that is implied by an implementation of the ASN.1/C++ run-time support system.

Integer Arithmetic, instead of Polynomial Arithmetic: Time is also consumed in routines that have to do multi-precision arithmetic operations, like addition, multiplication and modulus. SPIN scores a big plus on this

Model	Options	Comparison Factor	N=5	N=6	N=7	N=8
Sort	SAFETY	Mem (SPIN / EASN)	1	0.88	0.825	0.821
	NOREDUCE	Time(SPIN / EASN)	3.4	3.4	3.2	2.52

Table 2: EASN vs SPIN: Scaling

aspect since it uses cheaper polynomial division operations in its hash functions. Although our choice of the integer arithmetic enables us to implement the hashing algorithm incrementally, it would be much faster if we could find another lower cost mapping from our full-blown C++ version of the State-of-EASN to some linearised representation that also enables us to compute the latter from the former incrementally, like we can now.

Huge Executables: The generated validators usually make very few system calls, but due to the very large (compared to the size of the SPIN-generated validators) size of our validators, we see much higher system activity in our runs, as compared to that in the case of SPIN validators.

We have also attempted to see how EASN compares with SPIN with a change in the problem size (6.2). Note that as the size of the system being validated is increased, the memory benefit of EASN's more compact state-vector begins to show, and also, as the SPIN state-vector size grows larger, the run-time cost of handling the same begins to reduce the gap between the two validators' performance. The sort program from the Test database has N instances of the `middle` proctype in the system, each reading from and writing into its left and right channels respectively. The system also has N channels. Increasing N, therefore increases the size of the system being analysed.

6.3 Future Work

We believe, there are mainly two lines of activity that can be pursued, to get better results, than are possible by further cleaning up our implementation of any wasteful handling of either memory and/or time, when executing the validator.

- Since the generated validator does not require the generic interface capability that is required by EASN for simulations, generating the validator in the C Language might result in better performance. This does not detract from anything we have discussed in this paper as there is also a standard translation from ASN.1 to C. The generated data types, therefore, can be accessed just like they can be in the case of Promela/SPIN, reducing some of our costs but still giving us the benefit of having the type information available from ASN.1. The subtype information can continue to be used in our linearisation algorithms and the incremental hash computation. The EASN language definition could still be the starting point for this tool which still works with the ASN.1/C++ bindings, but for the purposes of the validator, works with the ASN.1/C binding also.
- We can also study ways of reducing the cost of multi-precision arithmetic (addition, multiplication, and division).

References

[1] Holzmann, Gerald J., Doron Peled, "The state of SPIN", CAV '96.

[2] Rob Gerth, Eindhoven University, "Concise Promela Reference", August 1997, Soft-copy available with SPIN.

[3] G. Gerth, D. Peled, M. Y. Vardi, P. Wolper, "Simple On-the-fly Automatic Verification of Linear Temporal Logic", PSTV94.

[4] Holzmann, G.J., Design and Validation of Computer Protocols, Prentice Hall, 1992.

[5] Patrice Godefroid, "Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem", PhD Thesis, University of Liege, Computer Science Department, Nov. '94.

[6] Information Technology - Abstract Syntax Notation One (ASN.1): Specification of Basic Notation, (Technical Corr. 1, Amd. 1:Rules of extensibility), ITU-T Rec. X.680 (1994), Corr.1 (1995), Amd. 1 (1995); Information Object Specification (Amd. 1: Rules of Extensibility), ITU-T Rec. X.681 (1994), Amd. 1 (1995); Constraint Specification, ITU-T Rec. X.682 (1994); Parametrization of ASN.1 specifications, ITU-T Rec. X.683 (1994)

[7] ASN.1/C++ Application Programming Interface, Part 1: Base Classes & Specific Interface, NMF 040 - 1, & Part 2: Generic Interface, NMF 040 - 2, Issue 1.0, February 1998

[8] Holzmann, G.J., SPIN Sources, Version 3.4.1, 15th August 2000; "Basic Spin Manual", available with SPIN.

[9] J.Geldenhuys, PJA de Villiers, 'Runtime Efficient State Compaction in SPIN,' The 5th Intl SPIN Workshop on Theoretical Aspects of Model Checking, ed. D. Dams, M. Massnik.

[10] Anindya Basu, 'A Language-based Approach to Protocol Construction', PhD Dissertation, Cornell Univ., Aug. '97

[11] ASN.1/C++ Application Programming Interface, Issue 1.0 Draft 10a - Submission to X/Open August 21, 1996

A Introduction to ASN.1

In our work, we use the 1994 definition of ASN.1 (ITU-T Recommendation X.680, 07/94)[6]. The first version of ASN.1 was defined in 1986 and has since gone through 3 rounds of revisions, with the 1994 version being the most recent.

ASN.1 is used mainly for specification & implementation of telecommunication protocols. Its use enables communicating entities to have a common abstract view of the structure of all packets that are exchanged amongst themselves, at various levels of the protocol stack. ASN.1 specifications are organised as a collection of modules. All module-names are in a single name-space; each module, furthermore establishes a name-space that contains all the types & constants defined in it. Below, we elaborate upon the subset of the facilities provided in ASN.1 that has been selected to be replacing the Promela datatypes in EASN, version 1.

A.1 Basic Data Types

In this section we describe those basic data types of ASN.1 that have been selected to be part of EASN. In another section on ASN.1/C++, we discuss the functionality exported by the C++ translation of ASN.1 modules, as specified in the NMF Standard. This set of basic data types is richer than that of Promela, which comprises of bool, bit, char, short, int, (atmost one) mtype declaration, and unsigned.

Boolean: Values of this type can take on values 'TRUE' & 'FALSE'.

BitString: A BitString is used to store a string of bits. Names can be associated with bit-positions in this string of bits. The first bit in the string is referred to as 'bit zero' and the last bit is the 'trailing bit'.

IA5String: This data type is used for strings of ASCII characters.

BMPString: This data type is used to store strings of 2-byte characters.

OctetString: This data type is used for strings of octets.

NullType: Objects of this type can only take on the Null value. This Null value cannot be assigned to objects of any other type.

Integer: Integers in ASN.1 are of unlimited precision, similar to the mathematical concept of an integer but unlike in most programming languages. Names can be assigned to specific integer-values that can be assigned to objects of these types.

Object-Identifier: An object Identifier is a list of numbers (or named numbers). It is used to identify objects of interest unambiguously.

Enumerated-Types: This basic data type construct can be used to create enumerated data types very similar to the way it is done in C++, using the enum construct.

GeneralizedTime: Objects of this type are used to store time values, for instance, in implementation of algorithms that require timestamping. (This is actually a 'Useful Type', as classified in the ASN.1 Specification).

A.2 Constructs to Compose User-defined Data Types

Just as there are the 'struct', array ([]), & 'union' constructs in the C language to compose higher-order data-types form component-types, there are the 'Sequence', 'Sequence of' & 'Choice' constructs in ASN.1, that can be used for the same purpose, respectively. In Promela, the only comparable constructs are the 'typedef', and the array ([]) constructs.

Sequence: This construct of ASN.1 is similar to the 'struct' construct in C, whereby one can create an ordered heterogeneous collection of objects. There are additional facilities here, however, whereby, *default* values can be associated with certain components, or to specify that certain components are *optional*.

Sequence-Of: This construct is comparable to the array construct in C, whereby one can create homogeneous ordered collections of objects.

Choice: This construct in ASN.1 is comparable to the 'union' construct in C. There is no comparable construct in Promela.

The above mentioned constructs are part of the subset of ASN.1 that has been included into the current version of the EASN Language. Besides these, there are two other constructs in ASN.1 called the 'Set', & 'Set of', that are described here, merely, for completeness.

Set: This construct is similar to the Sequence construct, except that there is no ordering implied in the case of a 'Set' of components. For most practical purposes, this construct is same the as 'Sequence', above.

Set-Of: A Set of objects is similar to a Sequence Of objects, except that in the set, no ordering of elements is implied.

A.3 Constant Values

Constants can be named and assigned a value, as part of the ASN.1 module. Such constants could be either of the basic types or of any user-defined types. The syntax that is used to specify the value of such constants is referred to as the ASN.1-syntax or the value-notation, in this paper.

A.4 Constraint Specification or Subtyping

Types in ASN.1 can be viewed as mechanisms to describe sets of values. From this perspective, one can talk of sub-typing by constraining the set of values. For instance, the basic-type, Integer, can take on any arbitrary integral value; one can therefore create a type Int8, that takes on values in the range: -127 .. 128.

As another example, consider a sequence type (named Point) that contains three integer components, corresponding to X, Y & Z coordinates for points in a 3-dimensional space. Now, an unconstrained set of such points would enable collecting arbitrary points in space, whereas for a set of points where the X coordinate is constrained to value 8.21 (say) would be able to collect only points on the two-dimensional plane X=8.21.

Another form of subtyping is through size-constraining, wherein, for example, the size of a set of objects has a cardinality constraint, or the length of a bitstring is strictly set to (say) 77. Apart from those discussed here, there are other means of constraining the value-set corresponding to user-defined types, in ASN.1. In the first version of EASN language, however, only single-value, value-range, size and union constraint constructs are supported.

[2] We do not need to concern ourselves with the various aspects of tagging or the algorithms involved therein, as it does not impact us.