Logic Verification of ANSI-C code with SPIN

Gerard J. Holzmann

Bell Laboratories, Lucent Technologies,
Murray Hill, New Jersey 07974, USA.
gerard@research.bell-labs.com

Abstract. We describe a tool, called AX, that can be used in combination with the model checker SPIN to efficiently verify logical properties of distributed software systems implemented in ANSI-standard C [18]. AX, short for Automaton eXtractor, can extract verification models from C code at a user defined level of abstraction. Target applications include telephone switching software, distributed operating systems code, protocol implementations, concurrency control methods, and client-server applications.
This paper discusses how AX is currently implemented, and how we plan to extend it. The tool was used in the formal verification of two substantial software applications: a commercial checkpoint management system and the call processing code for a new telephone switch.

1. Introduction

The construction of a reliable logic verification system for software systems has long been an elusive goal. It is well-known that in general even simple properties, such as program termination or system deadlock, are formally undecidable [23]. Efforts that target the development of mechanical verification systems therefore usually concentrate on carefully defined subsets of programs. Such subsets can, for instance, be obtained by imposing syntactic restrictions on general programming languages. These restrictions, however, are usually shunned by practicing programmers. An alternative method that we will explore here, is to define a system of abstractions for mechanically extracting verifiable *models* from programs written in a general purpose programming language. The abstractions are defined in such a way that the feasible executions of the modeled programs can be analyzed, if necessary exhaustively, by a model checker.

The staples of the industrial software quality control process from today, i.e., code inspection, peer review, and lab testing, work well for stand-alone, sequential applications that display primarily deterministic behavior. These methods, however, fail when they are applied to distributed software systems. A distributed system is not stand-alone or sequential, and its behavior is typically non-deterministic. It is not too surprising that the behavior of even non-buggy distributed applications can easily defy our human reasoning skills. In many cases, though, it is possible to analyze such systems thoroughly with the help of model checking techniques.

Our aim is to develop a system where default abstractions are sufficient to make the verification problem tractable, with a capability to perform fast approximate checks mechanically. By adjusting the abstractions, the user of such a system should be able to increase the level of precision of a check, but it is important that this is not a prerequisite for the applicability of the method. The default checks should return helpful results also without user guidance.

Verification Engine. The verification engine that is central to the work described here is the Bell Labs model checker SPIN [12].[1] Until now, to user of a model checking system will typically manually define an abstract model that captures the essential aspects of an application [2],[13]. Properties can then be formulated as assertions, or more generally as formulae in propositional temporal logic [20]. SPIN can perform either an exhaustive check that proves whether or not the model satisfies the property, or it can deliver a best-effort estimate of this fact within given time or memory constraints. For reasonable constraints (e.g., given a memory arena of around 64 Megabytes, and a runtime constraint of up to ten minutes) the probability of missing a property violation in even a best-effort check is usually small [14].

The use of a mechanical model extraction directly from source code, *without* imposing restrictions to or requiring prior modifications of that source code, can for the first time make it practical to use software model checkers in routine software development. The manual effort is now moved from the construction of the model itself to the definition of a set of abstraction rules, as will be explained. The rules can be checked for soundness and completeness by the model extractor, and may in some cases be generated by default to limit user interaction.

Earlier Work. We earlier attempted to perform direct verification of program sources written in a subset of the CCITT specification language SDL [11]. This lead to the development of the verifier SDLvalid, which was used at our company between 1988 and 1993. Though successful as a verification effort, the restrictions that we had to impose upon the language were too strict for routine programming. Special purposes languages such as SDL also appear to have fallen from favor among programmers, and today most new applications are written in general purpose languages such as C, C++, or Java. Although the tool SDLvalid is itself no longer used, it inspired a number of other verification tools, several being sold commercially today.

An early prototype of a model extractor for C code was constructed in August 1998, to support the direct verification of a commercial call processing application, as documented in [15]. The call processing software is written in ANSI-C, annotated with special markers to identify the main control states. We wrote a parser for this application, and used it to mechanically extract SPIN models from the code. These models were then checked against a library of generic call processing requirements. Over a period of 18 months, the model extraction and verification process was repeated every few days, to track changes in the code and to assure its continued compliance with the requirements. The verification method proved to be thorough and efficient — capable of rapidly intercepting a class of software defects with minimal user involvement. This application of model checking techniques to source level C code is the largest such effort known to us.

In February 2000 we generalized the model extraction method by removing the reliance on the special annotation for control states. The new model extractor was embedded it into a publicly available parser for programs written in full ANSI-standard C, [5]. By doing so we could replace two programs (named Pry and Catch [15]) with a single one (AX). The revision took approximately one month, and produced a fully general model checking environment for programs written in C, without making any assumptions about the specifics of the code and, importantly, without

[1] Online references for SPIN can be found at http://cm.bell-labs.com/cm/cs/what/spin/Man/.

adding any restrictions to the source language.

Related Work. Several attempts have been made to build translators that convert program text literally, without abstraction, into the input language of a model checker. Since no abstraction is used in these cases, one has to resort to imposing restrictions on the input language. Such subsets were defined, for instance, for Ada [4], Java [8],[9], and, as noted, for SDL [11].

For Java, two recent projects are also based on the systematic use of abstraction techniques: the Bandera toolset, in development at Kansas State University [3], and the Pathfinder-2 project at NASA Ames Research Center [24]. The work in [3] bases the model extraction process on the use of program slicing techniques [8],[19],[22]. Both [3] and [24] also target the inclusion of mechanically verified predicate abstraction techniques, by linking the model extraction process to either a theorem prover or a decision procedure for Pressburger arithmetic.

The work we discuss in this paper does not require, but also does not exclude, the use of external tools such as slicers or theorem provers. It is based on the definition of an abstraction filter that encapsulates the abstraction rules for a given body of program code. The method is independent of the manner in which the abstraction rules are generated. Importantly: they allow for the user to define or revise the rules either manually or mechanically. The abstraction rules can be checked for soundness and completeness independently, or they could be machine generated in a predefined way. The problem of syntax conversion from the source language to the model checking language is avoided in [24] by making use of the fact that Java programs can be compiled into a generic byte-code format for a Java virtual machine. The context in which a model extractor for C code has to operate is significantly more complex. There is, for instance, no virtual machine definition for ANSI-C, the use of pointers is unrestricted and can easily defy the pointer-alias analysis algorithms used in program slicers or data dependency algorithms. The use of an abstraction table provides an effective general mechanism for dealing with these practical issues, as we will illustrate here.

A number of other techniques attempt to find a more informal compromise between conventional system testing and simulation on the one side and program verification and logic model checking techniques on the other side. Interesting examples of these are Verisoft [7], which can be used on standard C code, and Eraser [21], which can be used for analyzing Java code. These tools analyze programs by monitoring the direct execution of the code. There is no attempt to compute a representation of the reachable state space, or to check anything other than safety properties. The method that we propose in this paper can be distinguished from these approaches by not only being more thorough (providing full LTL model checking capabilities), but also by allowing us to perform model checking on also partially specified systems.

In the sections that follow we will describe how the AX model extractor works, and we discuss some examples of its use on unrestricted industrial size code.

2. Model Extraction

We can distinguish three separate phases in the model extraction process. Optionally, the three phases described here can be preceded by the use of program slicing and by an independent verification step to prove the soundness of the abstraction rules used. We will not discuss program slicing or soundness checks here, but instead concentrate on the model extraction process itself. The three phases are then as follows. In the

first phase the model extractor creates a parse tree of the program source text, taking care to preserve all pertinent information about the original source. The *second phase* adds a semantic interpretation based on either a predefined or a user-defined set of abstraction rules, and the *third phase* optimizes the structure thus obtained and generates the verification model. We discuss each of these phases in more detail below.

2.1. First Phase: Parsing

A typical parser for a programming languages attempts to discard as much information as possible about the original program source, since this usually has no further significance to code-generation. In our case this type of information is important, and must be preserved, for instance, to allow for accurate cross-referencing between the executions of the source program and those of the extracted model.

The full parse tree that is built by the model extractor defines the control flow structure for each procedure in the original program, and it contains information about the type and scope of each data object used. The output of the parsing phase could be represented as an uninterpreted model, interpreting only control flow structure of the program, but treating everything else as an annotation. All basic actions (i.e., declarations, assignments, function calls) and branch conditions are collected, and can be tabulated, but they are as yet uninterpreted.

2.2. Second Phase: Interpretation

The *second phase* of the model extraction process applies an interpretation to each basic action and condition. All basic actions and conditions that appear in the source of the application can be inserted into a table, where it can be paired with an interpretation. If no abstraction table is provided the model extractor can generate one and fill it with a safe default interpretation that we will describe below. In many cases the default interpretations suffice, but the user can also revise the table. When model extraction is repeated (it takes just a fraction of a second) the model extractor will then use the revised abstraction table instead of the defaults and generate a model that conforms to the new choices. We refer to the basic actions and conditions as the "entries" in the abstraction table.

The rule table allows us to implement a remarkably broad range of systematic abstraction techniques. We mention some of the more important types below.

1. Local slicing rules. These rules are used to identify data objects, and their associated operations, that are irrelevant to the properties to be proven. Each declaration, function call, or statement within the set to be hidden from the verification model is interpreted as a skip (a null operation). Each condition that refers to a hidden object is made non-deterministic. A data dependency analysis can be used to identify the associated operations for each data object that is to be sliced away in this manner.
2. Abstraction relations. More general types of abstraction relations can also be captured as conversion rules in the table. This includes support for stating predicate abstractions [24]. A predicate abstraction is used when the implementation value domain of specific data objects carries more information than necessary to carry out a verification. The minimally required information can be captured in boolean predicates. For each such predicate we then introduce a boolean variable (replacing the original declaration of the data objects in the table), and only the boolean value of the predicate is tracked in the model, where possible. In

some cases, the loss of access to the full value of the original data object can make it impossible to evaluate the predicate. In those cases non-determinism is introduced to indicate that either boolean value would be possible. The predicate abstraction relations can be derived or checked with separate external tools, such as a theorem prover or a decision procedure, as in [24].

3. Syntax conversion. An important class of rules in the table will deal with mere syntax conversions, e.g., to accurately represent the transmission or reception of messages exchanged between processes, or the creation of new asynchronous process threads. We will give some examples of this below.

4. Restriction of verification scope. Finally, the same formalism of the abstraction table can be used to define restrictions to the scope of the verification attempt itself, e.g., by limiting buffer sizes, message vocabulary, the number of channels or processes, etc.

The use of semantic interpretation of a program through the tabled abstraction methods is at the core of the method introduced here. We discuss more details in a separate section below.

2.3. Third Phase: Optimization

A notable portion of the source text of an application is often of no direct relevance to the verification of general functional properties of the code. With the corresponding actions *hidden*, the control structure can of course be simplified. For instance, if we find a fragment like (in SPIN's specification language):

```
if
:: true -> /* comment1 */
:: true -> /* comment2 */
fi
```

specifying a nondeterministic choice between two commented out portions of code, clear we can simplify the model without changing its semantics by removing it completely. Similarly, a structure such as

```
if
:: true -> stmnt1
fi;
stmnt2
```

can be reduced to:

```
stmnt1; stmnt2
```

and in a structure like

```
false -> stmnt1; stmnt2; ...
```

everything after the unsatisfiable condition can be omitted. The model extractor uses a small set of rewrite rules to optimize the verification models for these cases.

3. Tabled Abstraction

The larger part of the abstraction table consists of a left-hand side entry, which is a canonicalized representation of a basic statement, a condition or its negation, from the source text of the application, and a right-hand side that specifies its semantic interpretation in the verification model. In many cases, a safe pre-defined interpretation

can be applied, assigned automatically by the model extractor. Four predefined types of interpretations for actions are listed in Table 1.

Table 1. Predefined Interpretations.

Type	Meaning
print	Embed source statement into a print action in the model
comment	Include in the model as a comment only
hide	Do not represent in the model
keep	Preserve in the model, subject to global Substitute rules

The predefined types from Table 1 can be used to either suppress (slice away) or preserve specific *types* of program statements in the model. The use of data types that are not representable in the input language of the model checker, for instance, is by default suppressed, but can be replaced with a more targeted, and independently verified, type of abstraction by the user. The capability to apply 'local' slicing, at the statement level, is valuable in dealing with the sometimes low-level details of routine applications written in C. It is, for instance, possible to slice away all function calls that can be shown to be redundant to the verification effort with a single entry in the abstraction table.

An example of an abstraction table with three mapping rules and two global Substitute rules, is as follows.

```
Substitute   FALSE                    false
Substitute   BOOL                     bit

D: int pData=GetDataPointer();        hide
D: BOOL m_bConnected                  keep
A: *((int *)pData)=(int)nStatus       print
A: m_bConnected=FALSE                 keep
```

Declarations are prefixed (by the model extractor) with a designation "D:" and assignments are prefixed with "A:". Assume that it can be determined that the use of variable pData is irrelevant to the property to be proven. The variable declaration is suppressed from the model with the mapping hide, but can nonetheless preserve visibility of access to the variable by mapping assignments to print. The print designation means that whenever this statement is encountered the verification model will not execute but print the source text of the statement. The latter capability is important to allow for the proper interpretation of error-trails produced by the model checker.

Data Dependency. To verify that abstraction rules are applied consistently, the model extractor performs a data dependency analysis and checks it against the rules. If, for instance, an assignment to a data object, or its declaration, is hidden (mapped), it is checked that all other read and write accesses to the same data object, and all access to other data objects that are dependent on it, are also hidden (mapped).

If a particular statement does not appear in the abstraction table a default rule is applied. For assignments, the default rule could be print, so in that case the above entry need not be included explicitly in the abstraction table at all. The user can override builtin defaults by entering a new definition for the default rule in an (optional) Preferences file. There can be an entry in this preferences file for each basic type of

statement (e.g., declarations, assignments, function calls, conditions), with a corresponding default that will then replace the predefined default.

Point Abstractions. All branch conditions, e.g. those used in iteration and selection statements to effect control flow, are entered twice into the abstraction table by the model extractor: once in positive form as found in the source text, and once in negated form. The reason for this apparent redundancy is that in the verification model we have the option of mapping *both* versions to true, and thus introduce non-determinism into the model. This type of mapping is sometimes called a 'point-abstraction,' and readily shown to preserve soundness. Consider, for instance, the following case:

C: (device_busy(x->line))	true
C: !(device_busy(x->line))	true

The precise determination if a given device is idle or busy is considered to be beyond the scope of the verification here. For verification purposes it suffices to state that both cases can occur, and the results of the verification should hold no matter what the outcome of the call is.

Restriction. In a similar vain, though, we can use a mapping to false as a constraint, to restrict the verification attempt to just one case:

F: (device_busy(x->line))	true
F: !(device_busy(x->line))	false

Here the verification would check correct operation of the system when the device polled is always busy.

Syntax Conversion. In some cases, the predefined interpretations from Table 1 are not adequate to cover the specifics of a verification. For the applications that we have considered so far, this applied to fewer than 20% of the entries in the machine generated abstraction tables. The following example illustrates typical use.

F: m_pMon->SendEvent(dest_Id,etype)	destq!etype
F: _beginthread(assign,0,0)	run assign(0,0)

The first entry defines a syntax conversion for the transmission of a message. The second entry defines the startup of an asynchronous process, executing a given function. Note that within the source text of the application these types of statements can take any form, since there is no generally accepted standard library for such operations. The abstraction table here serves to standardize the format that is used, without impeding the freedom of the programmer to chose an arbitrary representation. Automatic abstraction or checking techniques can generally not verify the correctness of these semantic interpretations. For a given application environment, though, we can manually provide a library of standard interpretations and include them into the abstraction tables.

Domain Restriction. The next table illustrates the definition of a domain reduction, or predicate abstraction, in the use of an integer variable.

D: int timer	bool timer
A: (timer > 0)	keep
A: !(timer > 0)	keep

```
A: timer = 60      timer = true
A: timer = 10      timer = true
A: timer = 0       timer = false
E: timer--         if :: timer = true :: timer = false fi
```

We assume here that the precise value of the timer is not relevant to the verification being undertaken, but the fact that the timer is running or not is relevant. We introduce a new boolean variable for the predicate "(timer > 0)", which is true when we know the timer has a non-zero value, and false when the timer-value is zero. We can use a decision procedure, as in [24] to derive the corresponding abstractions, and prove them sound. In this case we can trivially also do this by hand. When the timer variable is decremented, as illustrated in the last entry from the sample table above, the semantic interpretation reflects that the resulting value could be either positive or zero, using non-deterministic choice.

Short-Hands. The abstraction table is generally much smaller than the program text from which it is derived. The user can shorten it still further by exploiting some features of the model extractor. First, any entry that maintains its default mapping can be omitted from a user-maintained table: the model extractor can fill in these missing entries as needed. Second, the user can use patterns to assign the same mapping to larger groups of entries that match the pattern. For instance, suppose that all calls of the C library-functions `memcpy` and `strcpy` are to be hidden. We can avoid having to list all different calls by using ellipses, as follows:

```
F: memcpy(...      hide
F: strcpy(...      hide
```

This method could be expanded into a more general pattern matching method based on regular expressions. The above prefix match, however, seems to suffice in practice.

The second method for introducing abbreviations uses the Substitute rule that was shown earlier. Substitute rules take effect only on mappings of type keep, and they are applied in the order in which they are defined in the abstraction table.

We will give some statistics on the size and contents of typical abstraction tables in the section on **Application**.

Priorities of Abstraction Rules. Abstraction table, preferences file, and the builtin defaults of the model extractor may all give different instructions for the conversion of a specific fragment of program code. The various rules therefore have different priorities, as follows:

1 If a source fragment is matched in the abstraction table, the mapping from the table always takes precedence.

2 In the absence of an explicit mapping, the model extractor first builds a list of all data objects that are accessed in the given fragment of source code. For each such object, a check is made in the preferences file (if defined) to see if a restriction on that data object was defined. The restriction, if it appears, will be one of the entries of Table 1. If multiple data objects are accessed in a given source fragment, each could define a different restriction. In this case the restriction that appears first in Table 1 is applied. (E.g., print takes the highest priority, and keep the lowest.) If, however, the source fragment is classified as a condition, a mapping to hide or comment is replaced with a mapping to true.

3 If no explicit mapping applies and no data restrictions apply, then the default

type rules will be applied. First note that each source fragment at this level can be classified as one of four distinct types: assignment (A), condition (C), declaration (D), or function call (F). For each of these types the model extractor has a builtin mapping to one of the entries from Table 1. The builtin mapping can be overruled by the user with an entry in the preferences file. The redefined rule, however, also in this case remains restricted to the mappings specified in Table 1.

The model extractor can generate the templates for the abstraction table and the preference file automatically, so that any later adjustment to the focus of the verification consists only of making small revisions in one or two pre-populated files.

Synopsis. Note that the use of an abstraction table provides support for a fairly broad range of standard abstraction techniques, including local slicing (data hiding) and point-abstractions (introducing non-determinism), but also model restrictions and, e.g., reductions to the cardinality of a data domain. In addition, the table format allows us to define systematic rules for syntax conversion and model annotation, critical model extraction capabilities that no other method previously described appears to provide.

4. Verification

The output of phases one, two, and three are all executable and can be expressed in a format that is acceptable to the verification engine (which in our case is SPIN). For a meaningful verification, however, two additional items have to be formulated, in as little or as much detail as preferred. First, the user can formulate the functional properties that the application is expected to satisfy. By default the verifier can check for absence of deadlocks, completeness, absence of race conditions, and the like. But, generally, for more precise checks can be made through the use of properties expressed in temporal logic. The same is true in conventional testing: a targeted test of required functionality of the code is far more effective than some default random executions of the system.

The second item that needs to be added to properly direct the verification is an environment model: a stylized set of assumptions about the behavior of the environment in which the application is to be used. Testers often refer to this activity as *test scaffolding* or the construction of a *test harness*. The test harness serves to separate the application from its environment, so that it can be studied in detail under reproducible conditions.

In a verification environment the capability to build environment models and test drivers that can be connected in any way desired to the verification target is particularly rich. It remains to be seen how this power can be made available to those not familiar with the model checking paradigm. One method may be to allow the user to specify the test drivers in C, just like regular test scaffolding that would be constructed to support conventional testing, and to extract the test driver automata from that code, just like the main verification model. In the applications we have pursued we found it simpler to specify the test harness directly in the specification language of the model checker, but this is likely to be a personal preference, not a prescript for general use of the method.

5. Application

A predecessor of AX [15] was used in the verification of the call processing software for the PathStar access server [6]. This version of the tool, like AX, used a abstraction table, but in this case the table was entirely used-defined, and initially had no facility for abbreviations and default designations such as keep, hide, and print. In the PathStar application, all call processing code is grouped into a single module of the system, written in ANSI-C with some small annotations, as discussed earlier in this paper. The details of the verification effort have been described elsewhere. Here we concentrate on the model extraction framework itself.

Approximately 25% of the verification model for this application was handwritten, the remaining 75% mechanically extracted from the program source. The 25% handwritten code consists primarily of a detailed test harness, capturing our assumptions of the possible behaviors of local and remote subscribers and of neighboring switches in the network. The test harness contained six different types of test drivers for this purpose.

In the abstraction table for the call processing code about 60% of the code is mapped to keep, 30% is mapped to either hide or print, and the remaining 10% has an explicit alternate mapping. In most cases, the explicit alternate mapping support a notion of abstraction. Timers, for instance, are mapped from an integer domain into a Boolean domain, to exploit the fact that only the on/off status of the timers was required to verify system properties for this application.

A more recent application of AX was to the code of a checkpoint management system. The code for this application is written in C++, which first had to be converted into ANSI-standard C code to support model extraction. For this application the test harness formed was 31% of the verification model, the remaining 69% mechanically extracted from the source code. A total of ten procedures of interest were identified and mechanically extracted from the code as part of the verification model.

In the abstraction table for the checkpoint manager about 39% of the code is mapped to keep, 40% to either hide, print, or comment, and the remaining 21% was given an explicit mapping, mostly to express simple syntactical translations for message send and receive events.

Table 2. Statistics on Tabled Abstraction.

	PathStar Code	Checkpoint Manager	FeaVer System	Alternating Bit Test	Quicksort Test
Nr. Entries in Map	478	198	104	24	16
Nr. Elipse Rules	34	19	20	1	0
Nr. Substitute Rules	0	5	0	0	0
Default Rules	90%	79%	34%	50%	25%
Explicit Rules	10%	21%	66%	50%	75%
% Model Handwritten	25%	31%	50%	4%	73%
% Model Extracted	75%	69%	50%	96%	27%

Table 2 summarizes these statistics, together with the same metrics for the application of the model extractor to implementations in C of a client-server application that is used to implement the distributed software for the FeaVer verification system from [16].

For comparison the same statistics for two small test cases applications are also

included. The first test case is a 54 line implementation in C of the alternating bit protocol [1]. The second test is a deliberately chosen non-concurrent and data-intensive application. For this test we extracted a SPIN model from a 59 line implementation in C of the quicksort algorithm, based on [17]. The model extraction method does not target these types of applications, but it can still be useful to consider how badly the system might perform if it is applied outside its normal domain. For this application the larger part of the model was a handwritten routine for supplying input and for verifying that the output returned was properly sorted.

For the larger applications, the amount of code that can be treated with a default mapping is also the largest, limiting required user actions considerably compared with the effort that would be required to construct the verification models manually. The client-server code for the FeaVer system has more data dependencies, which show up as a larger fraction of the abstraction tables being mapped explicitly. In many cases, these explicit mappings consist of relatively simple syntactical conversions from C code to SPIN code. Once the abstraction table is defined for a given application, it can track changes in the source code with relative ease, and the user need only review occasional additions or revisions of entries in the table to see if the default choices from the model extractor need adjustment.

6. Extensions

The abstraction table used in the prototype version of our model extractor serves two purposes: (1) to define default and user-defined abstractions and (2) to apply syntactic conversions from C to SPIN's input language PROMELA. The current model extractor uses a default rule to print or hide constructs that, because of language differences, would otherwise require user-intervention to be converted into PROMELA code. This applies, for instance, to statements that access data objects via pointers and to statements that contain function calls. These defaults are the 'last resort' of the model extractor, when it encounters a statement that has no obvious representation in PROMELA. The defaults allows the model extractor to generate syntactically correct verification models, that can be used immediately to perform simulations and initial verifications, but clearly the quality of the verification can be improved substantially if the user adjusts the default mappings in these cases by defining abstractions in PROMELA that match the semantics from the statements in C.

The need for syntactic conversions and manual intervention can be reduced if we allow for included C code fragments in a SPIN verification model. Such an extension was, for instance, described in [10] for an early predecessor of SPIN, serving a different purpose. The C code fragments are not parsed by SPIN but included *as is* into the model checking code that is generated from a verification model (which is C source text). Since the data objects that are accessed through the included C code fragments can contribute to the system *state*, they must now be included in the model checker's *state vector* [12]. The model extractor can be adjusted to automatically generate the right information for the most commonly occurring cases. As a syntax for the included code fragments we can consider, for instance,

```
c_decl { ... }
c_code { ... }
```

respectively for included declarations and included code, both in ANSI C syntax.

The included C code is likely to use memory more liberally than PROMELA would. It

may use dynamically allocated memory and might access data in arbitrary ways via pointers. Both issues can in principle be resolved within the model checker. Dynamically allocated memory, for instance, can be tracked in the model checker by monitoring calls to memory allocation routines like sbrk() and malloc(). This gives the model checker an accurate view of all memory that can contain state information. The memory can be compressed and added to the state-vectors. The option would increase the runtime and memory requirements of a search, but would preserve its feasibility.

With bit-state hashing, the memory overhead for the state-space is avoided, though not necessarily the memory overhead for the depth-first search stack. In a bit-state search some state vectors are maintained in uncompressed form on the depth-first search stack (i.e., for actions that have no simple inverse that SPIN can precompute). For exceptionally large state-vectors, this could become prohibitively expensive when elaborate use is made of included C. In this case we can consider using SPIN's disk-cycling option, which maintains only a small portion of the stack in memory with only negligible runtime overhead.

When C code is included into a PROMELA verification model, we can no longer rely on SPIN's builtin interpreter to perform random or guided simulations. To solve this, we plan to add an option to SPIN for generating the simulation code as compilable C code, which can be linked with the related code for model checking that is already generated. Through the extended version of pan.c an identical set of simulation options can then be offered for models with included C code as SPIN does currently for pure PROMELA models.

Despite the hurdles, the option to allow included C code inside SPIN verification models appears to be feasible, and with that the possibility to perform at least approximate verifications of applications written in C. Because of the nature of the problem, it is not possible to provide any guarantees of verifiability. The thoroughness of a check can optionally be improved by a user by replacing included C code with abstracted PROMELA code, or by improving already existing abstractions, using the model extraction method we have described. If the code is entirely abstracted in PROMELA, we obtain a model with properties that are in principle fully decidable. But, also in this case effective verifiability will remain subject to the physical constraints of available memory and time.

The tabled abstraction method allows us to either set the desired level of abstraction, to obtain an increase in thoroughness and precision, or to ignore the issue if the default choices are adequate. The application of model checking techniques to program implementations can add an exceptionally thorough debugging capability to a programmer's toolset, for a class of software faults that is not adequately covered by existing techniques.

On Soundness. The emphasis in this paper has been on the work that is needed to construct a framework for model extraction for a implementation language that is widely used by practicing programmers, yet curiously often shunned by many of us who seek broader use of verification techniques in practice. The abstraction table method proposed here gives the verifier complete control over the abstraction and conversion process, perhaps more control than would at first sight appear to be desirable. Other techniques, beyond the scope of this paper, can be used to check or generate some of the abstractions, e.g., with the help of theorem proving techniques. Our plan is to add such techniques into our framework in a second phase of this work. It should be clear at this point, though, that these techniques by themselves are not

sufficient to handle the full generality of applications that can be written in a language such as C. In many cases it will be impossible to formally prove the soundness of an abstraction for a given fragment of C code, or to mechanically generate an adequate abstraction. It is nevertheless desirable that we are able to perform exploratory verifications that are based on such abstractions.

7. Conclusion

Our long-term goal is to develop a reliable testing framework for distributed software applications that can be used without knowledge of model checking techniques. The work on AX is a step in that direction.

The main benefits of the model extraction method we have described can be summarized as follows.

- It allows the application of verification techniques to both high-level design models and low-level implementation code. This makes it possible to track an application throughout a design trajectory, providing an early bug detection capability that currently is not provided by other methodologies.
- After the initial project startup, the application of the fault tracking method based on model extraction requires only modest user intervention.
- The method targets a class of concurrency related software faults that is not addressed by traditional testing techniques. The faults of this type can cause dramatic system failure.
- The use of model checking allows us to introduce reproducibility, controllability, and observability into a test environment for distributed systems: three required elements of a reliable testing strategy.

The applications most suitable for verification with AX and SPIN are control-oriented, rather than data-oriented. Our model extraction method exploits the fact that control information can always be extracted automatically from program sources, and data use can be abstracted where relevant. Applications such as the call processing code used in telephone switches, or more generally data communications protocols, client-server applications, and distributed operating systems code, contain large fragments of control-oriented code, and are therefore good targets for this methodology.

Once the capability for mechanically extracting verification models from program source code exists, the motivation to construct verification models by hand is seriously undermined. The mechanically extracted models are almost always more accurate, easier to manipulate, and can track changes in the source code more easily.

Having a reliable model extractor has allowed us to undertake verification efforts that we would not have attempted before, specifically of larger and more complex applications. An advantage of the method we have described is that the source of an application need not be complete before verification is attempted. Only the part that needs checking has to be provided, the rest can be abstracted in test drivers that become part of the verification model. For large software applications (e.g., a telephone switch) not all the software is generally available, and even if it were, it could not be compiled and executed on just any machine, for the purposes of formal verification. The method based on model extraction can be applied to any fragment of the code that is available, to confirm its properties.

To combine the benefits of program slicing with the tabled abstraction method that is

described in this paper, we are currently developing support for property-based slicing of PROMELA models, i.e., as a generic *model* slicing capability. The difficult pointer aliasing problem from programming languages such as C reappears in this context, but in a much simplified form, as a *channel* aliasing problem. The manner in which channels can be aliased in PROMELA, however, are very limited, which greatly simplifies data dependency analysis and slicing.

References

1. K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson, A note on reliable full-duplex transmission over half-duplex lines, *Comm. of the ACM*, Vol. 12, No. 5, pp. 260-261, May 1969.
2. Chan, W., Anderson, R.J., Beame, P., et al., Model checking large software specifications. *IEEE Trans. on Software Engineering*, Vol. 24, No. 7, pp. 498-519, July 1998.
3. J. Corbett, M. Dwyer, et. al. Bandera: Extracting Finite-state Models from Java Source Code. *Proc. ICSE 2000*, Limerick, Ireland, to appear.
4. M.B. Dwyer, and C.S. Pasareanu. Filter-based Model Checking of Partial Systems *Proc. ACM SIGSOFT Sixth Int. Symp. on the Foundation of Software Engineering*, November 1998.
5. S. Flisakowski, C-Tree distribution, available from http://www.cs.wisc.edu/~flisakow/.
6. J.M. Fossaceca, J.D. Sandoz, and P. Winterbottom, The PathStar™ access server: facilitating carrier-scale packet telephony. *Bell Labs Technical Journal*, Vol. 3, No. 4, pp. 86-102, Oct-Dec. 1998.
7. P. Godefroid, R.S. Hammer, L. Jagadeesan, Systematic Software Testing using Verisoft, *Bell Labs Technical Journal*, Volume 3, Number 2, April-June 1998.
8. J. Hatcliff, M.B. Dwyer, and H. Zheng, Slicing software for model construction, *Journal of Higher-Order and Symbolic Computation*, to appear 2000.
9. K. Havelund, T. Pressburger Model Checking Java Programs Using Java PathFinder *Int. Journal on Software Tools for Technology Transfer*, to appear 2000.
10. G.J. Holzmann, and R.A. Beukers, The Pandora protocol development system, *Proc. 3rd IFIP Symposium on Protocol Specification, Testing, and Verification*, PSTV83, Zurich, Sw., June 1983, North-Holland Publ., Amsterdam, pp. 357-369.
11. G.J. Holzmann, and J. Patti, Validating SDL Specifications: An Experiment, *Proc. Conf on Protocol Specification, Testing, and Verification*, Twente University, Enschede, The Netherlands, June, 1989, pp. 317-326.
12. G.J. Holzmann, The model checker SPIN. *IEEE Trans. on Software Engineering*, Vol 23, No. 5, pp. 279-295, May 1997.
13. G.J. Holzmann, Designing executable abstractions, *Proc. Formal Methods in Software Practice*, March 1998, Clearwater Beach, Florida, USA, ACM Press.
14. G.J. Holzmann, An analysis of bitstate hashing, *Formal Methods in System Design*, Kluwer, Vol. 13,, No. 3, Nov. 1998, pp. 287-305.
15. G.J. Holzmann, and M.H. Smith, A practical method for the verification of event driven systems. *Proc. Int. Conf. on Software Engineering*, ICSE99, Los Angeles, pp. 597-608, May 1999.
16. G.J. Holzmann, and M.H. Smith, Automating software feature verification, *Bell Labs Technical Journal*, April-June 2000, Special Issue on Software Complexity, to appear.
17. B.W. Kernighan, and R. Pike, *The Practice of Programming*, Addison-Wesley, Cambridge, Mass., 1999, p.33.

18. B.W. Kernighan, and D.M. Ritchie, *The C Programming Language, 2nd Edition*, Prentice Hall, Englewood Cliffs, N.J., 1988.

19. L. Millett, and T. Teitelbaum, Slicing PROMELA and its applications to protocol understanding and analysis, Proc. 4th Spin Workshop, November 1998, Paris, France.

20. A. Pnueli, The temporal logic of programs. *Proc. 18th IEEE Symposium on Foundations of Computer Science*, 1977, Providence, R.I., pp. 46-57.

21. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391-411, November 1997.

22. F. Tip, A survey of program slicing techniques. *Journal of Programming Languages*, Vol. 3, No. 3, Sept. 1995, pp. 121-189.

23. A.M. Turing, On computable numbers, with an application to the Entscheidungs problem. *Proc. London Mathematical Soc.*, Ser. 2-42, pp. 230-265 (see p. 247), 1936.

24. W. Visser, S. Park, and J. Penix, Applying predicate abstraction to model checking object-oriented programs. *Proc. 3rd ACM SOGSOFT Workshop on Formal Methods in Software Practice*, August 2000.