EMBEDDING PROGRAMMING LOGICS IN HOL THEOREM PROVER

A. Azurat, I.S.W.B. Prasetya, and S.D.Swierstra

Institute of Information and Computing Sciences
Utrecht University, P.O.Box 80.089
3508 TB Utrecht , the Netherlands
email : {ade,wishnu,doaitse}@cs.uu.nl

VV

ABSTRACT

HOL is a theorem prover based on a higher order logic. Its expressive logic makes it suitable for embedding programming logics. Compared to other theorem provers, HOL is attractive because of its familiar and intuitive logic and because it is highly programmable. In this paper we will compare a number of commonly used embedding approaches in HOL and outline their strength and weakness. We will also outline a new alternative called *hybrid embedding* that combines the strength of other approaches, though some price will have to be paid.

1. INTRODUCTION

Program verification is a large, tedious, and quite often complicate task. It is too error prone if done without the computer aid. In the formal verification community, implementing a logic in a computer is often called mechanizing the logic and the implementation is called the mechanization of the logic. Once a logic is mechanized, proof checking becomes automatic. Proof automation ability (the computer automatically constructs proofs) can subsequently be built on top of a mechanization.

Embedding is a special mechanization technique. An embedding of a logic L_1 in another logic L_2 is the semantic encoding of L_1 in the language of L_2, intended to allow tools for L_2 to be extended to L_1 [8]. So if L_2 is already mechanized, the mechanization also extends to L_1. Typically the inference rules of L_1 will be represented as formulas of L_2. This gives another advantage of embedding, namely that it is possible to verify the soundness of those inference rules with respect to the semantics of L_1 in terms of L_2. This is very useful when experimenting with new and complicated logics.

Theorem provers such as HOL, PVS, and COQ are essentially mechanizations. They are however special because their logics are very expressive, and therefore are excellent to be used as 'hosts' for embedding. HOL is especially attractive because it is based on a very familiar and intuitive logic and because it is easy for the users to build customized proof utilities.

They are various ways to do embedding. For example, we can distinguish them into so-called shallow and deep embedding[5]. Deep embedding embeds a logic completely, whereas shallow embedding only does this partially. Consequently, we can do less with shallow embedding, but it is also easier to construct. In some embedding of programming logics, programs variables are represented by first class values in the host logic. Depending on what we want to use a logic for, this can be an important aspect. For example, program transformations, composition, and refinements are often constrained by conditions which require first class representation of program variables to formulate. On the other hand, approaches in which variables are not represented as first class values typically produce cleaner representations. This paper will give a comparison between these approaches. As it turns out, none of these approaches can be considered as ideal for HOL. One of the problems is

HOL's limited type system. This is unfortunate because, as said, HOL has several aspects which for implementors are very attractive. Therefore a new alternative called hybrid embedding will be outlined. It can be easily built as an extension of HOL. It does not have the problems of the other approaches, although some price will have to be paid.

Section 2 describes the running example we will use to compare the embedding approaches. Section 3, 4 and 5 discuss the approaches and their problems. All embeddings target HOL as the host logic. We will however strip irrelevant HOL notational details from the code. Section 6 outlines an alternative approach. Finally Section 7 gives some conclusion.

2. RUNNING EXAMPLE: VSPL

Consider a very simple programming logic– abbreviated VSPL– described in Figure 1. VSPL programs are either a skip, an assignment, or a nested conditional. VSPL two types of values, namely Boolean and Integer. The logic is the standard Hoare logic. VSPL will be our running example.

1. Grammar:

Stmt	::	skip
		| *Variable* := *Expr*
		| if *Expr* then *Stmt* else *Stmt*
Expr	::	*Expr* = *Expr*
		| *Expr* ^ *Expr*
		| *Expr* → *Expr*
		| *Expr* + *Expr*
		| ¬ *Expr*
		| *Variable*
Spec	::	{*Expr* } *Stmt* {*Expr* }
		| [*Expr*]

2. Inference rules:

Skip Rule
$$\frac{[P \Rightarrow Q]}{\{P\}\ skip\ \{Q\}}$$

Assignment Rule
$$\frac{[P \Rightarrow Q[E/x]]}{\{P\}\ x := E\ \{Q\}}$$

If-then-else Rule
$$\frac{\{P \wedge g\}A\{Q\}, \{P \wedge \neg g\}B\{Q\}}{\{P\}\ if\ \ g\ then\ A\ else\ B\{Q\}}$$

Figure 1: VSPL

3. SHALLOW-F EMBEDDING

This section will show a shallow embedding of VSPL in HOL using an approach where VSPL variables are represented by first class HOL values. This embedding will be called Shallow-F embedding.

3.1. Semantic

Shallow embedding concentrates on how the semantics of the guest logic can be represented in a theorem prover. It is less concerned with various syntactic structures and constraints of the guest logic.

A state of a program at a given moment describes the values of the program's variables at that moment. We can represent a state by a function from variables to values. We can use string to represent variables (actually, variables' names). This representation is quite simple and is used by many others, e.g. [2, 7, 10, 13]. Alternatively, one can also use lists to represent states.

Since VSPL only has two kind of values, booleans and integers, we can represent VSPL values in HOL with the following HOL data type:

Code 3.1 :
```
datatype Value_IB =
fromInt int | fromBool bool
```

However, if we use this specific datatype the resulting embedding of VSPL inference rules will only work on that specific representation of data values: they cannot be reused if one decides to extend VSPL with new types of data values. To allow reuse, we can represent VSPL data values with a HOL type variable ι which can later be instantiated by a concrete HOL type. Alternatively, we can represent VSPL data values by a specific HOL type Value whose property is now left unspecified. Whenever we introduce a new VSPL type X we also add a function that injects X into Value. Both approaches are almost equivalent, however the second approach is slightly more flexible, so we will focus on it. Below we define the semantic domains of VSPL:

Definition 3.2 : Semantic domains of VSPL
```
type State = string → Value
type Expr = State → Value
type Pred = State → bool
type Stmt = State → State
```

We can now define the HOL semantics of VSPL statements:

Definition 3.3 : Semantics of Stmt

SKIP	= (\s. s)
IF g THEN A ELSE B	= (\s. if (g s) then A s else B s)
x ASG E	= (\s. (\v. if v=x then E s else s v))

where s is of type State.

1. Skip-rule
|- (P ==> Q) ==> HOA SKIP P Q

2. Cond-rule.
|- HOA A (P AND g) Q ∧ HOA B (P AND NOT g) Q
==>
HOA (IF g THEN A ELSE B) P Q

3. Assign-rule
|- VALID (p IMP (q o (x ASG E)))
==>
HOA (x ASG E) p q

Figure 2: VSPL's inference rules in HOL

3.2. Inferences Rules

VSPL's inference rules can be represented by HOL formulas. Figure 2 shows the resulting HOL theorems, representing VSPL's inference rules. Proving them is quite easy.

Passing theorems to HOL's Modus Ponens inference rule will have the same effect as invoking the inference rules they represent. Since (VSPL) inference rules are represented as theorems, it means they have to be proven first. So, no inference rule can be embedded if it cannot be proven to be derivable from its semantics in HOL. So, embedding is very safe.

3.3. Representing and Verifying Program

Consider the following VSPL expression:

Example 3.4 :
$b \wedge \neg(x + 1 = y)$

One may expect to represent the expression in HOL with:

Code 3.5 :
```
(\s. s "b") AND NOT (\s. s "x" + 1 = s "y")
```

However this is not type correct. It requires states (the s) to be functions returning boolean values (as in s "b") as well as integers (as in s "x" + 1). Moreover, we have decided states are functions from `State` to `Value`. Fortunately we have left the property of `Value` unspecified in the semantics. We can now say that it is large enough to contain booleans and integers. We then add the needed constructors to construct a `Value` from a boolean or integer and the corresponding and destructors:

Definition 3.6 : Constructors and Destructors of Value
```
fromInt : int → Value
fromBool : bool → Value
toInt : Value → int
toBool : Value → bool
```

In addition, we have to impose that each `from-` function forms an injection and that its counterpart (the corresponding `to-` function) is its inverse. The expression in Example 3.4 can now be represented correctly by:

Code 3.7 :
```
(\s. (toBool o s) "b")
  AND
NOT(\s. (toInt o s) "x" + 1 = (toInt o s) "y")
```

Unfortunately, the use of constructors and destructors clutters the representation to the point that it becomes unreadable, even for a simple program. Consider the following simple VSPL specification:

Example 3.8 :
$\{b \wedge \neg(x = y)\}$
if $x = y$ then $x = x + 1$ else $y = y + 1$
$\{b \wedge \neg(x + 1 = y)\}$

Here is its representation in HOL:

Code 3.9 :
```
HOA
(IF (\s. (toInt o s) "x" = (toInt o s) "y")
   THEN ("x" ASG (fromInt o (\s. (toInt o s) "x" + 1)))
   ELSE ("y" ASG (fromInt o (\s. (toInt o s) "y" + 1))))
((\s. (toBool o s) "b") AND
   NOT(\s. (toInt o s) "x" = (toInt o s) "y"))
((\s. (toInt o s) "b") AND
   NOT(\s. (toInt o s) "x" + 1 = (toInt o s) "y"))
```

Here is another inconvenience. Consider again the VSPL expression in Example 3.5. If in VSPL `b` is intended to be an integer variable, then the expression is not correctly typed in VSPL. However, its representation in HOL (Code 3.7) is a correctly typed HOL expression, regardless the intended VSPL type of `b`, `x`, and `y`. So, we cannot rely on HOL's own type system to type check VSPL sentences. A dedicated type checker for VSPL will have to be built, either externally, or embedded in HOL.

shallow-F embedding also has a problem in representing nested types, like array or list. Suppose we now extend the type of VSPL such that it allows (nested) lists. Unless we want to represent each level of nesting with a concrete HOL type, the obvious way to represent VSPL type list in HOL is with the HOL type `t list` where t is a type variable representing an arbitrary type. However, we still need the `from-` and `to-` functions. In particular, the `from-` function has to inject the type `t list` into `Value`. Since `t` can be any type, it is not possible in HOL to construct such an injection (else we will be introducing the Russel paradox in HOL) [6].

Recall that in the shallow-F embedding program variables have first class HOL representation (in our example, they are represented by HOL strings). This is an advantage when the embedded logic is intended to support program transformations (note that programs composition and refinement are special cases of transformations) since they are often constrained by conditions on the used program variables.

For example, consider the following transformation. It states that we can safely weaken the guard g ^ h of an `if-then-else` statement by dropping the h provided the `else` branch can realize the post-condition if h does not hold:

Example 3.10 :

$$\frac{\{P\}\text{ if } g \text{ then else } A \text{ else } B\,\{Q\} \qquad \{P \wedge \neg h\}\, B\,\{Q\}}{\{P\}\text{ if } g \wedge h \text{ then } A \text{ else } B\,\{Q\}}$$

However we can also assert something stronger. The following rule states that it is sufficient to show that the `else` branch does not modify any (free) variables of the post-condition Q when h does not hold:

Example 3.11 :

$$\frac{\begin{array}{l}\{P\}\ \text{if } g \text{ then } A \text{ else } B\ \{Q\} \\ Q \textit{ is confined by } V \\ B \textit{ preserves } V \textit{ when } \neg h \\ \quad [P \quad \wedge \neg h \Rightarrow q]\end{array}}{\{P\}\ \textit{if } g \wedge h \textit{ then } A \textit{ else } B\{Q\}}$$

Given a predicate Q and a set of variables V , Q is confined by V means that V is a set of the free variables in Q. A preserves V when ¬h means that the action A will not change any variable in V when ¬h holds. It is quite obvious, that to define the semantics of confined and reserves we will need first class representation of variables –see [4]) for their formal definition.

4. SHALLOW-R EMBEDDING

This section will briefly show another shallow embedding approach where (concrete) states are represented by records (alternatively, although less sophisticated, one can use tuples). For example, consider a program P with two variables, namely b and x. A state of P in which the value of b is true and the value of x is 0 can be represented by the following record in **HOL: <| b=T; x=0 |>**. Notice that b and x are the field names of the record. They are not first class HOL value. So we cannot, for example, prove in HOL that <| **b=T; x=0** |> actually contains a field called x (although we can still access the value stored in any field).

The approach produces cleaner representations of programs and specifications and is used by many, for example as in [12, 1, 9].

The shallow-R embedding requires a slightly different semantic domains. As in the shallow-F embedding, the type `Pred` represents predicates, and `Stmt` represents statements. However, they are now parameterized by a type variable 's which represents the type of states of an arbitrary VSPL programs:

Definition 4.1 : Semantic domains

```
type 's Pred = 's  → bool
type 's Stmt = 's  →  's
```

With the exception of assignment, VSPL statements, VSPL boolean operators and VSPL inference rules can be defined in the same way as the shallow-F embedding, though they should now be defined in terms of the new semantic domains.

As for the assignment, the target variable cannot be concretely represented in the shallow-R embedding. For example, the VSPL assignment x = x+1 is represented by the state transition function: **(\s. s with x := s.x + 1)**. Given a record s (representing a state), this function updates the field x with its old value (s.x) plus 1. We now redefine ASG as follows:

Definition 4.2 : Semantics of Assignment

```
ASG f = f
```

The inference rule for assignment now looks like this in HOL:

Definition 4.3 : Assign Rule

```
|- VALID (p IMP (q o f)) ==> HOA (ASG f) p q
```

We can now represent the VSPL specification in Example 3.8 as follows:

Code 4.4 :

```
HOA
(IF (\s. s.x = s.y)
    THEN (ASG (\s. s with x := (s.x + 1)))
    ELSE (ASG (\s. s with y := (s.y + 1))))
((\s. s.b) AND NOT (\s. s.x = s.y))
((\s. s.b) AND NOT (\s. s.x + 1 = s.y))
```

where s is assumed to have a record type, which have at least b, x, and y as fields. Compared to the shallow-F embedding representation, this one is apparently cleaner.

shallow-R embedding can also reuse much of HOL's type checking to do its own type checking. Consider again the VSPL expression in Example 3.4. If b is intended to be an integer variable, the expression is not type correct in VSPL. Recall that in the shallow-F embedding, HOL will not be able to see this, unless a VSPL type checker is explicitly included in the embedding. In the shallow-R embedding, the expression will be represented by:

```
((\s. s.b) AND NOT (\s. s.x + 1 = s.y))
```

Since b is intended to be an integer variable in VSPL, we should choose a representation where the states are represented by records where the field b has the type integer. So, s.b above returns an integer. However, AND expects a boolean, so the expression is also not type correct in HOL, and hence rejected.

shallow-R embedding also has no problem in representing nested types (e.g. list and array). The problem occurs in the shallow-F embedding because it uses an intermediate type `Value` to represent all possible forms of VSPL values. shallow- R embedding does not need such an intermediate representation. However, the shallow-R embedding also has its drawback:

1. The fact that variable names is not represented by first class HOL values means that we will not be

able to embed program transformation rules where variables names have to be treated as first class values, such as the transformation rule in Example 3.11.

2. Two programs P and Q with different set of variables must be represented using states with di_erent record types. To compose P and Q, we must construct a new record type which can accommodate the variables of both program and subsequently introduce functions to inject the state space of the original P and Q into the new state space. If we do a lot of composition, this will clutter the representation (Unfortunately HOL does not support extended record, which will eliminate this problem. See for example [9]).

5. DEEP EMBEDDING

Shallow embedding does not represent every aspect of the embedded logic. For example, as in Sections 3 and 4, statements are typically represented in HOL by functions. It is possible then, to prove in HOL whether two programs, which are just compositions of functions, are equivalent. It is not however possible to prove in HOL, for example, that a program contains at least N assignments.

One can make a deeper embedding by making more aspects of the embedded logic explicit in the guest logic. Obviously a deeper embedding is more powerful, but, as we will discuss later, it also has its own problems. It is di_cult to say where the border between shallow and deep embedding exactly lies. But one can safely say that a representation of a logic L_1 in L_2 such that every aspect of L_1 can be given semantics, and therefore analyzed, in L_2 is deep embedding.

So-called data type is usually used to deeply embed a language in HOL. Data type is a data representation method common in functional languages. Using a data type D one can conveniently represents the context free grammar of a language L. Each of value of D will represent the complete syntactical of the corresponding sentence of L. Consider again the VSPL example. Here is a fraction of the HOL data types representing its grammar:

```
datatype A_stmt =
     Skip
   | Asg string A_expr
   | IfThenElse A_expr A_stmt A_stmt
```

HOL values of type A_stmt represents VSPL statements. However, HOL does not know yet what the relation between A_stmt-values and programs. We have to define so-called semantic functions that define the intended meaning those values. For example, like this:

Definition 5.1 :

```
MSt Skip                = SKIP
MSt (Asg x e)           = x ASG (MEx e)
MSt (IfThenElse g A B) =
IF (toBool o MEx g) THEN MSt A ELSE MSt
B
```

where SKIP, ASG, and IF-THEN-ELSE are HOL constants as defined in the shallow-F embedding (Section 3) and MEx is the semantic function for the corresponding deep embedding representation of VSPL expressions. Notice that the semantic function MSt actually translates our deep embedding representation of statements to their shallow embedding representation. In general, we can view an embedding as to consists of layers where each layer is more powerful than the layers beneath it, and is defined semantically in terms of the latter.

Note also that the semantics of assignment cannot be expressed in terms of shallow-R embedding. Consider the following attempt to do so:

```
 MSt (Asg x e) = (\s. s with x := MEx
e s)
```

Note that the x in Asg x e is a HOL value. More specifically, it is a HOL string representing the name of some VSPL variable. On the other hand, the x inside the lambda expression refers to a field name of a record. Although the field has the same name as the other x, HOL cannot compare an ordinary HOL value against a record's field name. Field names are simply not first class HOL values.

Notice that given a statement, which is now a value of HOL type A_stmt, it is now possible in HOL to, for example, count the number of assignments in it and prove properties about it.

As said, HOL data types can represent the complete context free grammar of the embedded logic. This gives another advantage, namely that it becomes easier to write the parsers and pretty printers to interface the embedding with the concrete syntax of the embedded logic L [11]. This will allow the users to interface with the embedding in the language of L itself, rather than in terms of embedded representations which are quite unreadable for human. In principle, it is also possible to embed the parsers and other syntax driven tools (such as a type checker) for L in HOL. The advantage is that their correctness can also be verified (though it does not mean that it can be easily done).

Despite being more powerful than shallow embedding, deep embedding has its own drawback. Programming languages and their logics tend to evolve after sometime. Some old features may be extended for some improvement. If we add a new language construct to a deeply embedded logic L, its representing HOL data types have to be extended too, along with their semantic deeply embedded logic L, its representing HOL data types have to be extended too, along with their semantic functions. Unfortunately, this will cause the old inference rules of L, which are embedded in HOL as theorems, to be no longer valid. So we will have to prove them all over again. Worse yet, because of the extension, some of the old proofs may now fail, so we have to 'debug' them, which can be quite time consuming.

Like in the shallow-F embedding, we also cannot use HOL's own type checker to check if, for example, a `A_stmt` value actually represent a well typed VSPL statement. A dedicated VSPL type checker will have to be built.

6. AN ALTERNATIVE: HYBRID EMBEDDING

Recall that the embedding of a logic L_1 in another logic L_2 is actually the representation of L_1 in L_2. In contrast to embedding, the direct mechanization of a logic L on some programming logic M is the implementation of L directly on M (so, not through an embedding). In embedding, inference rules are embedded by proving them first. In direct mechanization the rules are simply coded in M. compared to embedding, direct mechanization o_ers maximum power, since we can simply code down any inference rule that we want. It offers minimum maintenance, since we do not have to debug any proof, which can be quite costly, in case we change L. On the other hand, embedding, in particular deep embedding, offers maximum security.

Suppose the logic L_2 is mechanized in a programming language M. Hybrid embedding is an alternative mechanization approach. It embeds a part of L_1 in L_2 and directly mechanize the rest in M. Hybrid embedding combines the power of direct mechanization and the safety of embedding. Since only part of L_1 is embedded, changing L_1 will also incur less maintenance work. Because of this combination hybrid embedding is more suitable for mechanizing an industrial scale programming logic.

For example, the hybrid embedding of VSPL in HOL, which is mechanized in ML, may consist of a shallow-F embedding as described in Section 3 and a set of data types representing VSPL's syntax and a set of semantic functions (such as the data type `A_stmt` representing VSPL statements and the function `Mst` that maps `A_stmt` values to their shallow-F embedding semantics) in the style of deep embedding as shown in Section 5. However, unlike in the deep embedding, these data types and semantic functions are no longer HOL values. They are now implemented directly in ML. For example, it becomes possible to circumvent the problem of representing nested types encountered in the shallow-F embedding –the solution is rather too technical to explain, see [4]. On the other hand, we will not be able to, for example, prove that the semantic function `Mst` is monotonic.

Although directly coded (unembedded) inference rules are in principle unsafe, it is still possible to extend the code of such a rule so that its invocation generates a host logic proof to validate the result of the invocation. In this way we can make such a rule safer. For some rules, it may even be possible to not only validate their results, but completely verify them.

7. CONCLUDING REMARKS

We have discussed a number of embedding methods. The following table summarizes their strength and weaknesses

Table 1: Comparison of embedding methods. '-' and '+' indicate weak/strong point. '--' and '++' indicate weaker/stronger point.

Criteria	Shallow-F	Shallow-R	Deep	Hybrid
Works Effort	+	+	– –	–
Readability	– –	–	+	++
Safety	+	+	++	–
Extensibility & Maintenance	–	–	– –	++
Nested type	–	–	–	+
Type checking	–	++	+	+
Program Transformation	+	–	++	++

Hybrid embedding, despite being less secure than embedding, is still safer than mechanization without embedding. It is powerful and requires low maintenance. It is suitable for mechanizing industrial scale programming logic.

We are currently experimenting this approach in a tool called xMECH [3]. The tool is essentially a hybrid embedding of a logic for distributed systems. The logic features a PROMELA like programming language, a higher order expression language, and a UNITY style specification language. The host logic is HOL. So far, we are satisfied with the approach. The work is still on-going, as we are experimenting with ways to improve its security.

REFERENCES

[1] Sten Agerholm. *Mechanizing Program Verification in HOL*, Master's thesis, Computer Science Department Aarhus University, 1992.

[2] Flemming Andersen, *A Theorem Prover for UNITY in Higher Order Logic*, PhD thesis, Technical

Report on Xmech.,*Technical Report* UU-CS-2002-008, Institute of Information and Computing Sciences Utrecht University, P.O.Box 80.089 3508 TB Utrecht The Netherlands, January 2002.

[4] A. Azurat and I.S.W.B. Prasetya, A Survey on Embedding Programming Logic in A Theorem Prover, *Technical Report* UU-CS-2002-007, Institute of Information and Computing Sciences Utrecht University, P.O.Box 80.089 3508 TB Utrecht The Netherlands, January 2002.

[5] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel, Experience With Embedding Hardware Description Languages in HOL, In *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156, Nijmegen, 1992. North-Holland.

[6] Twan Laan, *The Evolution of Type Theory in Logic And Mathematics*, PhD thesis, Technische Universiteit Eindhoven, 1997.

[7] M.J.C. Gordon, Mechanizing Programming Logics in Higher-order Logic, In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin.

[8] C'esar Mu noz and John Rushby, Structural Embeddings: Mechanization With Method, In Jeannette Wing and Jim Woodcock, editors, FM99: *The World Congress in Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 452–471, Toulouse, France, sep 1999. Springer-Verlag.

[9] Lawrence C. Paulson, Mechanizing UNITY in Isabelle, *ACM Transactions on Computational Logic*, 1(1):3–32, July 2000.

[10] I.S.W.B. Prasetya, *Mechanically Supported Design of Self-stabilizing Algorithms*, PhD thesis, Utrecht University, 1995.

[11] S. D. Swierstra and P. Azero, Attribute Grammars in The Functional Style, In *Proceedings of the SI2000*, Chapman-Hall, 1998.

[12] J. von Wright and K. Sere, Program Transformations and Refinements in HOL, In Myla Archer, Jennifer J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the International Workshop on the HOL Theorem Proving System and its Applications*, pages 231–241, Los Alamitos, CA, USA, August 1992. IEEE Computer Society Press.

[13] Tanja Vos, *Unity in Diversity: A Stratified Approach to the Verification of Distributed Algorithm*, PhD thesis, Utrecht University 2000.