Systematically Constructing a CCS Interpreter
⇒ Summary ⇐

Peter van Eijk

University of Twente
P.O. Box 217
7500 AE Enschede
Netherlands
mcvax!utrcu1!pve

ABSTRACT

Correctness is a major concern in computer science, especially for programs with complicated semantics that people have to depend on, such as compilers and interpreters. One of the new paradigms of computer science is that it is better to ensure correctness by construction and transformation than by a posteriori proof. Most examples of the application of this paradigm, however, have been small. In this paper a transformational approach to the construction of an efficient 'interpreter' for a specification language is described. One of its applications has been LOTOS, a language for specifying communication protocols. Because of space constraints however, we will discuss here the application to CCS, a smaller language, which inspired LOTOS.

Keywords: correctness preserving transformations, LOTOS implementation, CCS implementation, systematic derivation of software.

1. Introduction

The motivation of the work described here stems from our work on software tools for LOTOS. LOTOS [ISO1988] is a language for formally specifying communication protocols and services. It is based on CCS [Milner1980], Milner's Calculus of Communicating Systems. In this paper we restrict ourselves (because of space constraints) to CCS, which should be sufficient to demonstrate our method. The full LOTOS version can be found in [Eijk1988].

Specifications in LOTOS and CCS specify components of systems (also called processes or behaviours) in terms of their observable behaviour. In this model, observing a process means interacting with a process, where interaction is modeled as rendez-vous interaction. Consequently, processes are modelled as offering a number of possible interactions, where each interaction will yield a new subsequent behaviour. Processes can be composed in a parallel way, with interaction on specified gates. Such interaction of processes, although in

This work was supported in part by the Commission of the European Community as part of the ESPRIT/SEDOS project.

principle unobservable and therefore called internal interaction, can have the effect that other, observable, behaviour changes. With such languages quite intricate interaction sequences can be specified.

As LOTOS is being used in the design of realistic protocols, such as the OSI [ISO1983] protocols, it is important to develop supporting tools. The analysis in [Eijk1988] shows that one of the central tools is a simulator or interpreter. The basic functionality of a LOTOS or CCS interpreter allows to go stepwise through all the interactions and checking whether or not they conform to the specifier's intentions. More detail on the functionality of such a interpreter can be found in [Eijk1988].

As in general, the implementation of such a tool has to be based on a definition of the semantics of the language, because the functions of the tool derive their definitions from these semantics. Unfortunately these definitions are not usually directly implementable, because of the formalisms in which they are expressed (In the case of LOTOS and CCS, inference rules defining transition systems). Even if they are, the resulting time and space efficiency may not be sufficient for a particular application. The purpose of this paper is to illustrate a software development method that addresses this issue, by systematically constructing an efficient interpreter for CCS. Each step in the sequence of successive transformations employs its own definition style. Nevertheless correctness arguments can be made at each step, ensuring the correctness of the resulting definition.

The resulting operational definition of the LOTOS semantics has formed the basis for a full LOTOS simulator that is used conveniently on large specifications. Furthermore, it has formed the basis for the construction of a 'verifier' for a subset of LOTOS, which can exercise thousands of states.

2. The Method

The goal is a definition of the essential functions of a simulator that allows an efficient implementation. This definition can be used in a number of tools. The following method for systematically deriving this definition is illustrated in this paper.

Step 1. Define the functions of the tool to be implemented in terms of the semantics of the language. For a functional language for instance, the function to be implemented could be the *evaluate* function. For LOTOS the function to be implemented is *menu*, which yields a particular subset of the transitions generated by a set of inference rules.

Step 2. Analyse why this definition is unsatisfactory from a particular implementation point of view. A non exhaustive list of such insufficiencies is:

- The definition is non-constructive. E.g. defining numbers as the solution of a polynomial equation is non-constructive, although it may be computable.
- The definition is constructive, but is expressed in a language with constructs that we do not want to implement. For example, it can employ higher order functions (functions yielding functions), a common property of denotational semantics.
- The definition is inefficient to implement, as can be the case if it employs substitutions.

These problems are implementation problems and not semantical deficiencies of the definition. Consequently, the way in which these issues are resolved constitutes an implementation decision. Not all issues need necessarily be resolved. It is dependent on e.g. the particular implementation strategy chosen. There are implementation languages that can handle higher-order functions (e.g. LISPKIT[Henderson1980]) and Prolog can successfully execute certain non-constructive (that is, not directly executable by sequential machines) definitions.

Step 3. The core of our approach is to remove the deficiencies one at a time by rewriting the definition of the functions to be implemented in such a way that an implementation decision that is made is reflected. In general this implementation decision is a decision on the representation of a certain central data structure that will enable a particular part of the functions to be realised more efficiently. Each step of refinement constitutes a transformation of the definition that preserves its correctness. If the correctness of each step is proved, the resulting implementation is guaranteed to be correct.

Step 4. After steps 2 and 3 have been repeated for a sufficient number of times, the definition of the functions will be detailed enough to admit a direct translation into a programming language. As mentioned earlier the class of programming languages is not necessarily restricted to high level languages like PASCAL.

3. Comparison with other approaches

Our approach is similar to Bjørner [Bjørner1982] where the evaluation function for a simple applicative language (named SAL) is given in a denotational style. This definition is then gradually transformed into a definition that can directly be used as the definition of an interpreter for SAL. Subsequently the definition is transformed in such a way that it lends itself to direct code generation, thus effectively describing a compiler. In this derivation the environment of variables is accessed through the common ‘display’ technique, although by making different implementation choices one can also arrive at the also common ‘static link’ technique.

A related line of research aimed at methods for arriving at executable specifications is to define formalisms that can be used to concisely specifying the semantics of a language while at the same time being executable. Examples of these include: attribute grammars [Kastens1982, Ganzinger1982, Teitelbaum1984], SIS (based on denotational semantics) [Mosses1979], PSP [Paulson1982]. A characteristic of this approach is that they force the language definition to use a predefined specification formalism. The only ways in which the efficiency of the resulting interpreter or compiler can be ameliorated is by ‘knowing’ which constructs are more efficiently implemented or by improving the generator itself. The advantage of this approach is that a prototype implementation is immediately available. The disadvantages are that the specification is confined to a formalism that may or may not be suitable, and that efficiency improvement options are limited.

Consequently the advantages of Bjørners method are that at each step in the transformation process the most convenient specification style can be chosen and that all implementation decisions are possible and can be reflected in a clear way. Since it is basically an informal method the disadvantage is that it cannot be supported easily by tools.

In a more general setting our approach conforms partly to the notions of ‘stepwise refinement’ [Wirth1974] and ‘implementing specification freedoms’ [London1982]. However, in contrast with the usual stepwise refinement approach we do have a complete specification at the first step, it is just not implementation

oriented enough. With respect to the second reference, although our method shares the basic idea of adding implementation detail at each step, we will not restrict ourselves in the same way to one a priori defined formal notation.

4. The Language CCS and its Semantics

The definition of CCS can be found in [Milner1980], chapter 5. Since this is a summary, only a small part can be given here.

The semantics of CCS is defined by an inference system which employs the following relation: 'behaviour B can produce or move to behaviour B´ under event *event'*, which is denoted $B -^{event}\rightarrow B'$. Examples of event denotations are $g?(v)$, which means: accept a value v on gate g, and $g!(v)$, which means: send a value v on gate g. The basic ground case for the inference system is formed by the so-called inference axioms, for example:

$$g?x:t;B -^{g?(v)}\rightarrow B\{v/x\}$$

Explanation: The process $g?x:t;B$ can do any receive event with a value v of type t, and then in the result v is substituted for x. The notation $B\{v/x\}$ denotes substitution of v for x in B.

Inference rules then describe the behaviour of expressions in terms of their components. One of the rules of parallel composition is:

$$\frac{B_1 -^{g?(v)}\rightarrow B_1',\ B_2 -^{g!(v)}\rightarrow B_2'}{B_1 | B_2 -^{\tau}\rightarrow B_1' | B_2'}$$

This reads: if B_2 can send a value v on gate g and B_1 can receive that value, then they can communicate, and their parallel composition $B_1 | B_2$ will after an internal step (denoted by τ) transform to the parallel composition of their successors.

5. Overview of Interpreter Functions

The basic model of the interpreter is that of a machine that traverses the tree of all possible behaviour. This tree is termed the communication tree by Milner. The nodes of this tree represent possible states, and the branches represent events. Note that internal events (τ-steps) are explicit elements in these trees.

The functions that are fundamental to the interpreter are:

- The *menu* function that yields the possible events in a certain state.
- The *next* function that allows to make a certain state change.

Other functions can be expressed in them, such as undoing the last step.

At each state there are a number of events possible. The list of these possible events is called the menu. In terms of the communication tree, the menu is the multiset of branches of the current node.

The *next* function allows one to move to a subsequent state, or to put it in another way, to exercise a certain transition, by making an event happen. Note that this may be an internal event. In case of an interaction with the environment that needs a value (like $g?x:integer$), this value is a component of the application of *next.*

The interpreter functions can be described directly in terms of the transition relation. Let the transition relation be denoted as above by

$$B \text{–}^{event}\!\!\rightarrow B'$$

then the *menu* function is given as

$$menu(B) = \{ (event, B') \mid B \text{–}^{event}\!\!\rightarrow B' \}$$

The *next* function is then a function from menu elements and values to behaviours, which fills in a value if necessary. Formally

$$next(B, event) = B' \quad \text{when} \quad B \text{–}^{event}\!\!\rightarrow B'$$

6. Definition of the Transformations

The need for transformations is motivated by undesirable properties of the definitions of the functions. Certainly, only computable functions are worth considering. Therefore, as a first step, we review the definitions of the next and menu functions for their computability. The menu function yields a set of events (actually a multiset or bag). For this set to be finite it is sufficient that we have a suitable representation of possible events and that the behaviour expressions are ‘guardedly well-defined’. To see the need for the first condition (finite representation) consider the behaviour

B = g?x:integer; B´

This behaviour allows an infinite number of events, one for each integer value. Nevertheless we can denote this infinite set in a finite way by allowing elements of the structure ‘g?x:integer’ in the menu. Guardedly well-definedness ensures that a behaviour cannot call itself recursively without participating in an event first. Without this condition, the menu could still have an infinite number of elements. Since this is a condition on behaviour expressions, it does not influence the interpreter function definitions. It is now decidable whether or not

$$B \text{–}^{event}\!\!\rightarrow B'$$

holds for particular B, *event*, and B'. This is a property of the particular inference rules. The next function is equally computable.

The definition of the interpreter functions now has the drawback of not being very oriented towards imperative programming languages. It is possible to derive an interpreter written in Prolog for this definition, but every computation involves a large amount of backtracking. Therefore an important transformation is one that yields a constructive, algorithmic definition of the menu.

The last step described in this paper involves eliminating substitutions of values for variables. Substitutions are cumbersome and inefficient to implement. They involve making copies of expressions substituted in,

and care must be exercised to avoid scope conflicts. It should be noted that semantically there is nothing wrong with the definition that employs substitutions.

In each following subsection a definition of the interpreter functions is given together with a motivation of the transformation that led to it.

6.1. Finite representation of the menu

For a finite representation of the menu we must transform the inference rules in such a way that they no longer generate infinite sets of transitions. This means that implicit enumerations over sets of values have to be turned into explicit parameterisations. Such implicit enumerations take the form of e.g. '... where E is an arbitrary value of type t'. This occurs in the rule for the CCS construct $g?x:t\,;B$. The transformation involves passing up the responsibility of providing such a value, and makes both the event and the resulting behaviour functions over such values. Instead of the menu containing events like *g?(1), g?(2), g?(3), etc.* we describe it as *g?(x:integer),* which introduces variables in the events and resulting behaviours. Where the original inference system generates only closed events, i.e. without variables, the new inference system also generates open events, containing variables.

Besides the change in representation of events, the effect is that the substitution of values for variables is moved from the axiom for action prefixing to the rules for communication of parallel processes. One of these rules for parallel communication then becomes after this transformation:

$$\frac{B_1 \text{-}^{g?(x:t)}\!\!\rightarrow B_1',\ B_2 \text{-}^{g!(v)}\!\!\rightarrow B_2'}{B_1 \mid B_2 \text{-}^{\tau}\!\!\rightarrow B_1'\{v/x\} \mid B_2'}$$

(other details omitted in this summary).

Let $\rightarrow$ and $\underline{\rightarrow}$ denote the first and second inference systems respectively, then the correctness claim is stated as: the transition system generated by $B \text{-}^{event}\!\!\rightarrow B'$ is equal to the transition system generated by $\forall\, v\ B \text{-}^{event(v)}\!\!\underline{\rightarrow} B'(v)$. This can be proven by induction over the inference rules. The first *event* is from the first inference system, and therefore closed, The second $event(v)$ is an instantiated open event from the second inference system.

The *next* function now changes, and has additional parameters to instantiate the open events.

6.2. Constructive definition of the menu

The definition of the interpreter functions using inference rules has the disadvantage of not being a constructive or algorithmic definition. This means that it cannot be mapped directly to a regular programming language. It is possible to some extent to translate it into a language such as Prolog as shown in [Briand1986, Eijk1986]. Basically, that approach involves translating the inference rules into a Prolog program that *generates* all possible transitions and then use the Prolog execution mechanism to compute only the transitions $B \text{-}^{event}\!\!\rightarrow B'$ for one particular B. E.g. the Prolog predicate $infer(B_1, Event, B_2)$ is defined by case analysis over the structure of B_1. Nevertheless, such an execution involves backtracking because, for example, in an expression like $B_1 \mid B_2$ the interpreter has to try out the possible interactions of the behaviours for all gates.

For these reasons, it is worthwhile to analyse if this backtracking can be avoided and an algorithmic definition can be found. Fortunately, this is the case as will be shown in the following.

The definition of *menu* as the solution of a set of inference rules is transformed into an algorithmic definition, that expresses the menu of a construct in the menu of its components. A typical example of this transformation is the following (B_1+B_2 stands for: choose from B_1 and B_2):

$$menu\,(B_1{+}B_2) = \{(event\,,B') \mid\ B_1{+}B_2 \text{–}^{event}\!\!\rightarrow B'\}\ \text{(by definition)}$$

by applying the inference rules for + we get:

$$menu\,(B_1{+}B_2) = \{(event\,,B') \mid\ B_1 \text{–}^{event}\!\!\rightarrow B'\}+$$
$$\{(event\,,B') \mid\ B_2 \text{–}^{event}\!\!\rightarrow B'\}$$

which is obviously equivalent to:

$$menu\,(B_1{+}B_2) = menu\,(B_1){+}menu\,(B_2)$$

Other cases are left out of this summary, but for the sake of illustration we give part of the parallel case:

$$menu\,(B_1 \mid B_2) =$$
$$\{(\tau, B_1'\{v/x\} \mid B_2') \mid \quad (g?(x:t), B_1') \in menu\,(B_1)\ \&\ (g\,!(v), B_2') \in menu\,(B_2)\}$$
$$+\ \cdots$$

The menu of $B_1 \mid B_2$ consists of, among others, an internal (τ) step when B_1 and B_2 communicate.

This transformation does not have an effect on the definition of *next*.

6.3. Definition of the menu with substitutions eliminated

Elimination of substitutions has a large impact on the efficiency of the implementation of *menu*. The basic idea in this transformation is that a substitution $\{v/x\}$ is replaced by the environment $[x \leftarrow v]$, and each expression evaluation takes the environment into account. In a sequential programming language, each state has one environment list. In the presence of parallel expressions, however, different scopes coexist concurrently. Example:

g?x:int; h!(x+x); B1 | g!3; h?y:int; B2

Both sides evolve and introduce new variable bindings independently. This implies that we need a way to attach different environments to different parts of the behaviour expressions that appear in the menu. This introduces a change in the structure of the behaviour expressions that appear as arguments to the menu function: e.g. a parallel composition now combines two behaviours and two value environments. Example: $(VE_1, B_1) \mid (VE_2, B_2)$.

The menu is now a function of an environment and a behaviour (full definition left out). The demonstration case from the previous section becomes:

$$
\begin{aligned}
&\mathit{menu}\,((VE_1, B_1) \mid (VE_2, B_2)) = \\
&\quad \{(\tau, (VE_1' \hat{}\, [x \leftarrow v\,], B_1') \mid (VE_2', B_2')) \mid \\
&\qquad (g?\,(x:t), (VE_1', B_1')) \in \mathit{menu}\,((VE_1, B_1)) \;\&\; \\
&\qquad (g\,!(v), (VE_2', B_2')) \in \mathit{menu}\,((VE_2, B_2))\} \\
&\quad + \cdots
\end{aligned}
$$

The notation $VE \hat{}\, [x \leftarrow v\,]$ stands for the concatenation of environment VE with an environment containing a value assignment of v to x, with VE having lower priority.

The function *next* now updates the environments to reflect instantiation by a value.

7. More possible transformations

The transformations described in this paper are motivated by undesirable properties of certain definitions. Although the last definition is a suitable one for the implementation of a CCS interpreter, it is not by necessity the endpoint in the process of applying transformations. In the following a few other transformations are described.

In the current definition the resulting menu is computed bottom-up. This can lead to the generation of large intermediate results of which only a fraction appears in the menu of the current expression. It is possible to change the evaluation to a multi-pass strategy, where in the first pass only the gates and sorts of the event offers are computed. The second pass would then only compute those events and resulting behaviours that actually appear in the menu. This would not change the exponential complexity of the computation, but would reduce the cost of the computation by a sizable factor. Such a version of the menu computation can be very conveniently described using attribute grammars. Each component of an element of the menu (gatename, sort-list, resulting behaviour, etc.) would then be represented by a separate attribute.

The menu and next functions effectively describe a CCS interpreter. In order to move towards compilation of CCS expressions, a step that can be done is making the menu function non-recursive by suitably representing the execution stack of the menu function. This stack (or whatever structure results) and the data structures already manifest in the menu function, would then be part of the run-time data structure of the execution of a CCS expression.

Since CCS is intended to be a language for the specification of concurrent systems, it would be very worthwhile to investigate possible parallelism in the evaluation of the functions. Both for the interpretation and for the compilation of CCS expressions it would be interesting to see how much of the computation involved in the next and menu functions can be distributed over multiple processors. One approach to this, which has a close relationship with the aforementioned multi-pass attribute evaluation, is to evaluate various invocations of the menu function in parallel. This is also very related to the way in which data flow machines operate. Another approach is to start from the CCS processes. One can try to distribute the processes over processors and investigate possible implementations of distributed synchronisation. It would be interesting to see if this yields a similar solution. From a preliminary analysis it seems not; e.g. in the first approach to parallelism one would expect the offers of both parts of a ‘+’-construct to be evaluated simultaneously, whereas in the second the emphasis would be on the ‘|’-construct.

8. Conclusion

This paper shows how an interpreter for CCS and its internal data structures can be derived in a systematic and correct way from a natural definition of the dynamic semantics of CCS. The same method has also been applied to LOTOS [ISO1988]. Although the impact of the LOTOS constructs on the details of the definition is considerable, the same method is still applicable. It is worth noting that in the resulting implementation, no errors in the dynamic semantics were found. The 'errors' reported where due to misunderstanding of the dynamic semantics by the users. I would like to thank Rudie Alderden, Jean-Pierre Briand, Jan Tretmans and other ESPRIT/SEDOS-C3 members for contributing to the implementation of the LOTOS simulator.

References

ISO1988.

ISO, ''Information processing - Open Systems Interconnection - The definition of the specification language LOTOS,'' International Standard ISO/IS 8807, 1988.

Milner1980.

R. Milner, ''A Calculus of Communicating Systems,'' *Lecture Notes in Computer Science*, vol. 92, Springer-Verlag, 1980.

Eijk1988.

Peter van Eijk, ''Software Tools for the Specification Language LOTOS,'' Ph.D. Thesis, Twente University of Technology, Enschede Netherlands, 1988.

ISO1983.

ISO, ''Information processing - Open Systems Interconnection - Basic Reference Model,'' International Standard ISO/IS 7498, 1983.

Henderson1980.

Peter Henderson, *Functional Programming, Application and Implementation*, Prentice-Hall International, London, 1980.

Bjørner1982.

D. Bjørner, ''Rigorous Development of Interpreters & Compilers,'' in *Formal Specification & Software Development*, ed. D. Bjørner & C. B. Jones, pp. 271-320, Prentice-Hall, Englewood Cliffs, 1982.

Kastens1982.

U. Kastens, B. Hutt, and E. Zimmermann, *GAG: A Practical Compiler Generator*, LNCS, 141, Springer-Verlag, Berlin, 1982.

Ganzinger1982.

H. Ganzinger, R. Giegerich, U. Moncke, and R. Wilhelm, ''A Truly Generative Semantics-directed Compiler Generator,'' *SIGPLAN Notices*, vol. 17, no. 6, pp. 172-184, June 1982.

Teitelbaum1984.

T. Teitelbaum and T. Reps, ‘‘The Synthesizer Generator,’’ *SIGPLAN*, vol. 19, no. 5, pp. 42-48, May 1984.

Mosses1979.

P. Mosses, ‘‘SIS - Semantics implementation system. Reference Manual and user guide,’’ Report DAIMI MD-30, Univ. Aarhus, 1979.

Paulson1982.

L. Paulson, ‘‘A semantics-directed compiler generator,’’ *POPL*, vol. 9, pp. 224-233, 1982.

Wirth1974.

N. Wirth, ‘‘On the composition of well structured programs,’’ *ACM Computing Surveys*, vol. 6, 1974.

London1982.

Philip E. London and Martin S. Feather, ‘‘Implementing Specification Freedoms,’’ *Science of Computer Programming*, vol. 2, pp. 91-131, 1982.

Briand1986.

J. P. Briand, M. C. Fehri, L. Logrippo, and A. Obaid, ‘‘Executing LOTOS specifications,’’ in *Proceedings of the sixth international workshop on protocol specification, testing and verification*, ed. Behcet Sarikaya, North-Holland, Amsterdam, 1986.

Eijk1986.

Peter van Eijk, ‘‘A comparison of behavioural language simulators,’’ in *Proceedings of the sixth international workshop on protocol specification, testing and verification*, ed. Behcet Sarikaya, North-Holland, Amsterdam, 1986.