Nondeterminism in Formal Development of Concurrent Programs: A Constructive Approach

Hassan Haghighi[1] , Seyyed Hassan Mirian-Hosseinabadi[2]

Department of Computer Engineering
Sharif University of Technology
Tehran, Iran
[1]haghighi@ce.sharif.edu, [2]mirian@sharif.edu

Abstract. It is now widely accepted that programming concurrent software is a complex, error-prone task. Therefore it is useful to specify, develop, and verify concurrent programs using formal methods. In our continuing work, we try to develop a constructive framework for extracting concurrent programs from their formal specifications. In this framework, we use CZ specification language and rely on a translation of CZ set theory into Martin-Löf's theory of types. In this paper, we introduce the first part of our work in which we regard and track the nondeterminism involved in CZ formal specifications of concurrent programs.

Key words: concurrency, constructivism, formal program development, nondeterminism in CZ language, nondeterminism in Martin-Löf's theory of types

1 Introduction

The application of *formal methods* to the specification of software systems is expected to increase the level of confidence in the correctness of final programs. However, the gap between the two phases of *formal specification* and program development causes this confidence to be degraded yet again during the transition from the specification to the actual program. Also, for complex programs, such as *concurrent* ones, *verification* does not effectively work. Hence a number of approaches have been proposed for developing programs from their formal specifications directly. Some methodologies for formal development of concurrent (or parallel) systems are *stepwise refinement* [1], *compositional* [3, 8], *Owcki-Gries theory-based* [4, 10].

In our continuing work, we try to provide a *constructive* framework for deriving concurrent programs from correctness proofs of their formal specifications in *CZ* formal specification language. *CZ* (*Constructive Z*) was introduced in [9] as a constructive version of the *Z* notation in order to employ the facilities of *Z* in organizing and manipulating formal specifications and the ability of constructive formalisms in program development. In [9], Mirian also provided an interpretation of *CZ* constructive underlying theory, *CZ set theory*, in *Martin-Löf's theory of types* [7]. In our future framework, one can write a specification of his or her own concurrent program in *CZ*. Then, by the existing interpretation of *CZ* in Martin-Löf's theory of types, this specification can be translated into a corresponding type (or specification) in Martin-Löf's theory of types. Finally, a concurrent program, which satisfies the initial *CZ* specification, can be derived from a correctness proof of the type theoretical specification.

Although many important problems are encountered when developing concurrent systems, including (dynamic) process creation, multi-threading, communication, scheduling, mutual exclusion, deadlock, and starvation, in this paper, we only concentrate on a special

problem, namely nondeterminism. Concurrency introduces nondeterminism as a consequence of timing dependencies during process (or thread) creation, synchronisation, and communication [2, 11, 12]: In a concurrent system, two or more candidates (programs, processes, threads, or expressions) compete for a common resource (e.g., a lock, an address space, a communication network, or a shared memory). To resolve this competition, a choice should be made whose result is not necessarily deterministic.

Difficult situations may occur when nondeterminism remains unnoticed during specifying, designing, and testing (concurrent) programs. For example, if only one of the possible executions occurs during testing, the others remain fully untested. In this case, users may rely on their application's fault-free operation while the program still holds the possibility for serious bugs. Also, as we have shown in [5], when formal specifications involve *implicit* nondeterminism, during deriving a program from a correctness proof of an implicitly nondeterministic specification, only one of the possible behaviors extracts from its proof tree and appears in the final program. In this way, the competition between concurrent components, specified in formal specifications of concurrent systems, cannot be implemented.

To solve the above mentioned problems, appropriate notations and semantics, which facilitate *explicit* specification of nondeterminism, should be introduced to formal specification languages. Also, current frameworks for deriving programs from correctness proofs of their formal specifications should be extended in a way which preserves the effects of the specified nondeterminism in final programs. In [6], some basic notations have been added to *CZ* language which facilitate explicit specification of nondeterminism. Also, an approach has been introduced in order to model nondeterminism in type theoretical specifications. Finally, the existing translation of *CZ* into Martin-Löf's theory of types [9] has been extended to map nondeterminate constructs in *CZ* specifications to their counterparts in Martin-Löf's theory of types. In this paper, we use this extended translation to extract concurrent programs from their nondeterministic specifications in *CZ*.

In section 2, we investigate the relationship between concurrency and nondeterminism. Relying on the results of [6], in section 3, we introduce a template for extracting concurrent programs from their formal (nondeterministic) specifications in *CZ*. The last section is devoted to the conclusion and directions for future work.

2 Concurrency and Nondeterminism

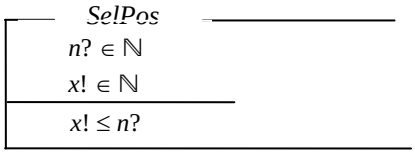
Nondeterminism is frequently encountered when reasoning about concurrent systems [2, 11, 12]:

- Threads are created so that they share the same address space which authorizes nondeterministic effects.
- In a concurrent system, many facilities or resources need to be shared by several processes (or threads). In other words, both processes and threads need to synchronise among them in order to cooperate effectively when sharing resources. When several processes (or threads) compete for the same resource, nondeterministic effects arise.
- Processes (or threads) can communicate with each other either via shared variables or via message passing. Communication by shared variables is similar to the synchronisation methods. It is based on the assumption that processes share common memory and communicate via shared variables which are stored within it. When communicating by message passing, processes are assumed to share a communication network and exchange data in messages by send and receive primitives. In this case, nondeterministic behavior may be observed in both sharing the communication network and using wild card receives. A call to a receive operation with a wild card as the source identifier allows a message from any process to be accepted. Thus the

unpredictable arrival order of messages may influence the program's behavior in a nondeterministic way.

The simple Example 2.1 shows a nondeterministic choice among some concurrent candidates which compete for a common resource.

Example 2.1 Assume that there are a number of concurrent (or parallel) components, coordinated by a common coordinator. The communication between each component and the coordinator is established by sending synchronisation messages via a shared communication network. Two components cannot send their messages along the shared network simultaneously. The following operation schema specifies an operation which nondeterministically selects a component with the identifier x from $n + 1$ concurrent components to communicate with the coordinator (This specification involves implicit nondeterminism):



The operation schema *SelPos* can be also considered as a specification of an operation which selects a message among $n + 1$ received messages at a wild card receive command.□

3 A Template for Extracting Concurrent Programs

In [6], we have introduced a framework which enables us to specify nondeterminism explicitly in *CZ* specification language and then translate nondeterministic *CZ* specifications into their counterparts in Martin-Löf's theory of types[1]. In this section, we suggest a template which utilizes this framework for developing concurrent programs.

According to the discussions of section 2, it can be easily implied that in all of the nondeterministic situations in concurrent environments, a set of rival items has been given from which one element, that satisfies a condition, must be selected. For example, at a wild card receive operation, we should select a process (from the set of available processes) which has sent a message to the receiver, or in a concurrent database, a transaction should be selected (from the set of active transactions) whose first operation does not conflict with the current lock base.

Now we consider a generalized form of the operation schema *SelPos* (in example 2.1) which specifies a competition among some concurrent components and is (implicitly) nondeterministic. In the following *CZ* specification, the operation schema *SelComp* specifies an operation which selects one winner from some concurrent candidates. $x_1, ..., x_m$ are input or before state variables, and *alt*! is an output variable which corresponds to the identifier of the selected candidate (the winner). This candidate should satisfy the predicate (condition) *P* which is the combination of the pre and the postconditions of *SelComp*.

[1] The reader is assumed to be familiar with this work.

SelComp

$x_1 \in T_1$

$\vdots$

$x_m \in T_m$

$alt! \in \mathbb{N}$

$P(x_1, \ldots, x_m, alt!)$

The previous specification involved implicit nondeterminism. To make it explicit, we use the notions of multi-schema and nondeterministic variable [6] as follows (The variable *alt*! becomes nondeterministic):

MSelComp

$x_1 \in T_1$

$\vdots$

$x_m \in T_m$

$alt! \in \&\mathbb{N}$

$P(x_1, \ldots, x_m, alt!)$

To extract a program from the above specification, we apply the extended function ς [6] to the schema *MSelComp* step by step. The result of these applications is the following type in Martin-Löf's theory of types:

$$[\![MSelComp]\!]_\varsigma = \prod \alpha_1 \in (\varsigma(T_1))^-, \ldots, \alpha_m \in (\varsigma(T_m))^- . \sum \beta \in \&\mathbb{N} . [\![P(\alpha_1, \ldots, \alpha_m, \beta)]\!]_\varsigma$$

Since a nondeterministic choice among concurrent candidates is bounded, the above specification can be interpreted as follows:

$$[\![MSelComp]\!]_\varsigma = \prod \alpha_1 \in (\varsigma(T_1))^-, \ldots, \alpha_m \in (\varsigma(T_m))^- . \sum ndv \in List(\mathbb{N}).$$
$$\prod \beta \in \mathbb{N} . InList(\beta, ndv) \Leftrightarrow [\![P(\alpha_1, \ldots, \alpha_m, \beta)]\!]_\varsigma$$

Finally, we can extract a program from a correctness proof of the above type theoretical specification.

We can use the above results as a general template to extract concurrent programs from their formal nondeterministic specifications in *CZ*. In example 3.1, we apply this template to the simple example 2.1.

Example 3.1 To extract a program from the operation schema *SelPos*, we replace it by a multi-schema *MSelPos* whose nondeterminism is explicit:

MSelPos

$n? \in \mathbb{N}$

$x! \in \&\mathbb{N}$

$x! \leq n?$

By applying ς to *MSelPos*, the following type theoretical specification is obtained:

$$[\![MSelPos]\!]_\varsigma = \prod \alpha \in \mathbb{N} . \sum \gamma \in List(\mathbb{N}) . \prod \beta \in \mathbb{N} . InList(\beta, \gamma) \Leftrightarrow \beta \leq^+ \alpha$$

We can finally extract a program from a correctness proof of the above specification. Part of this proof is shown in Appendix A. The resulting program is

$$prog = \lambda\alpha. R_{x,y}(\alpha, (\langle 0 \rangle, p), (\langle s(x) \rangle \frown fst(y), q))$$

p and *q* are the correctness proofs of $z=\langle 0\rangle$ and $v=\langle s(x)\rangle \wedge fst(y)$, respectively. The extracted program is a recursive function which results in the sequence $\langle 0\rangle$ for $\alpha=0$. Also, in the recursive step (computing the output for $\alpha=s(x)$), the result is equal to $\langle s(x)\rangle \wedge fst(y)$. $\wedge$ is the symbol of concatenation of two sequences and $fst(y)$ is the output for $\alpha=x$. For each $\alpha \in \mathbb{N}$, *prog* produces a sequence consisting of all possible outputs and thus preserves the competition among concurrent components. □

4 Conclusions and Future Work

In this paper, we have introduced part of our continuing work in which we try to develop a constructive framework for extracting concurrent programs from their formal specifications in *CZ*. Although many important problems are encountered when developing concurrent systems, we have only concentrated on their inherent nondeterminism. This research proceeds to examine other problems within our constructive framework.

References

1. R. J. R. Back, "Refinement of Parallel and Reactive Programs", Lecture Notes for the Summer School on Program Design Calculi, Germany, Springer-Verlag, pp. 73-92, 1993.
2. M. Broy, "A Theory for Nondeterminism, Parallelism, Communication, and Concurrency", Theoretical Computer Science, vol. 45, no. 1, pp. 1-61, 1986.
3. K. M. Chandy and M. Charpentier, "An Experiment in Program Composition and Proof", Formal Methods in System Design, vol. 20, pp. 7-21, 2002.
4. D. Goldson and B. Dongol, "Concurrent Program Design in the Extended Theory of Owicki and Gries", in Proc. CATS 2005, Australia, 2005.
5. H. Haghighi and S. H. Mirian, "Nondeterminism in Type Theoretical Specifications", in Proc. of the 10th Annual CSI Computer Conference, Iran, 2004.
6. H. Haghighi and S. H. Mirian, "An Approach to Nondeterminism in Translation of CZ Set Theory into Martin-Löf's Theory of Types", in Proc. Foundations of Software Eng. Conf. (Fsen 2005), LNTCS 159, 2006.
7. P. Martin-Löf, "An Intuitionistic Theory of Types: Predicative Part", H.E. Rose and J.C. Sheperdson (Editors), in Proc. Logic Colloquium 73, North Holland, pp. 73-118, 1975.
8. D. Meier and B. Sanders. "Composing Leads-to Properties" Theoretical Computer Science, vol. 243, no. 1-2, pp. 339-361, 2000.
9. S. H. Mirian, "Constructive Z", Ph.D. dissertation, Dept. Comp. Sci., Essex Univ., 1997.
10. S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach", Commun. ACM, vol. 19, no. 5, pp. 279-285, 1976.
11. E. Spiliopoulou, "Concurrent and Distributed Functional Systems", Ph.D. dissertation, Dept. Computer Science, Bristol Univ., 1999.
12. C. S. Pitcher, "Functional Programming and Erratic Nondeterminism", Ph.D. dissertation, Trinity College, Oxford Univ., 2001.

Appendix A

Extracting a program from the multi-schema *MSelPos*

$A \equiv \sum\gamma\in List(\mathbb{N}) \bullet \prod\beta\in\mathbb{N} \bullet InList(\beta, \gamma) \Leftrightarrow \beta \leq+ \alpha$
$B \equiv \prod\beta\in\mathbb{N} \bullet InList(\beta, \gamma) \Leftrightarrow \beta \leq+ \alpha$
$\Gamma \equiv \alpha\in\mathbb{N}, x\in\mathbb{N}, y\in A[x/\alpha]$

$prog = \lambda\alpha \bullet t$ $\quad t = R_{x,y}(\alpha, u, f)$
$u = (z, p)$ $\quad z = \langle 0\rangle$
$f = (v, q)$ $\quad v = \langle s(x)\rangle \frown fst(y)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha\in\mathbb{N} \vdash z = \langle 0\rangle}\,Hyp \quad \dfrac{}{\alpha\in\mathbb{N} \vdash head(v) = s(x)}\,Hyp \quad \dfrac{}{\alpha\in\mathbb{N} \vdash tail(v) = fst(y)}\,Hyp}{\alpha\in\mathbb{N} \vdash z\in List(\mathbb{N}) \quad \alpha\in\mathbb{N} \vdash p\in B[0/\alpha][z/\gamma]}\,\sum i}{\alpha\in\mathbb{N} \vdash u\in A[0/\alpha]} \quad \dfrac{}{\Gamma \vdash v\in List(\mathbb{N}) \quad \Gamma \vdash q\in B[s(x)/\alpha][v/\gamma]}\,Hyp}{\alpha\in\mathbb{N}, x\in\mathbb{N}, y\in A[x/\alpha] \vdash f\in A[s(x)/\alpha]}\,\sum i \quad \dfrac{\dfrac{}{\mathbb{N}\ type}\,\mathbb{N}f}{\alpha\in\mathbb{N} \vdash \alpha\in\mathbb{N}}\,ass \quad \alpha\in\mathbb{N} \vdash u\in A[0/\alpha]}{\alpha\in\mathbb{N} \vdash t\in A}\,\mathbb{N}e$$

$$prog \in (\prod\alpha\in\mathbb{N} \bullet \sum\gamma\in List(\mathbb{N}) \bullet \prod\beta\in\mathbb{N} \bullet InList(\beta, \gamma) \Leftrightarrow \beta \leq+ \alpha)$$