MASTER

POOSL-compiler for Smalltalk and C++

de Leijer, M.L.

Award date:
1997

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
POOSL-Compiler for Smalltalk and C++

By M.L. de Leijer

Coaches : ir. M.C.W. Geilen
dr. ir. J.P.M. Voeten
dr. ing. P.H.A. van der Putten
Supervisor : prof. ir. M.P.J. Stevens
Period : September 1996 - August 1997

The Faculty of Electrical Engineering of Eindhoven University of Technology does not accept any responsibility regarding the contents of Master's Theses.
Abstract

To support development of complex information-technology systems, adequate analysis, specification and design methods are required. At the section Information and Communication Systems of the faculty Electrical Engineering at the Eindhoven University of Technology, development of these methods is subject of active research. This research resulted in the method SHE (Software/Hardware Engineering) [PV97][PVS95]. SHE is an object-oriented method for co-development of complex reactive software/hardware systems, covering analysis, specification and design. SHE incorporates the formal specification language POOSL (Parallel Object-Oriented Specification Language) [Voe95a][Voe95b][PV97].

To create a useful environment for developing complex systems with SHE, software tools that support the specification process are indispensable. Because of the formal syntax and semantics of POOSL, SHE has the potentials for formal verification, transformation, simulation and implementation. POOSL specifications have to be translated to target code in order to be able to perform simulations and implementations.

This thesis describes the design and implementation of a compiler for POOSL, which is able to translate POOSL specifications to Smalltalk [PPS92] and to C++ [Str92]. The POOSL-compiler is implemented in the Smalltalk programming environment. The following results of other projects are used as a starting-point for the implementation of the POOSL-compiler:

- The syntax and grammar of POOSL formulated in [Kup96]
- The mapping of POOSL on Smalltalk formulated by Mark Geilen
- The mapping of POOSL on C++ formulated in [Tet97].

The POOSL syntax and grammar are modified for semantic and syntactical reasons. The new syntax and grammar of POOSL are formulated in Appendix A and B. The mapping on Smalltalk and C++ are given in Appendix C and D. The Smalltalk and C++ code that has to be generated can be seen as an intermediate code, because in both cases supporting target code is required to implement the semantics of POOSL. This support is implemented by the POOSL-simulator (by Mark Geilen) and the C++ library (by [Tet97]). Because the POOSL-simulator holds the POOSL classes of a system only translation of methods is required.

The POOSL-compiler is not (yet) able to check the context conditions of POOSL, so they must be satisfied by the user.
# Contents

Abstract ........................................................................................................ iii

1 Introduction ........................................................................................................ 9
   1.1 Project Context ......................................................................................... 9
   1.2 Objectives ................................................................................................. 9
   1.3 Report Organisation ............................................................................... 11

2 The Language POOSL .................................................................................... 13
   2.1 Introduction ............................................................................................. 13
   2.2 The Basics of POOSL ............................................................................. 13
      2.2.1 Data Class ....................................................................................... 13
      2.2.2 Process Class .................................................................................. 14
      2.2.3 Cluster Class .................................................................................... 14
   2.3 The Syntax of POOSL ............................................................................. 14
      2.3.1 Changes ........................................................................................... 15
   2.4 Context Conditions ............................................................................... 16

3 The Smalltalk Environment .......................................................................... 17
   3.1 Introduction ............................................................................................. 17
   3.2 Basics of Smalltalk ............................................................................... 17
      3.2.1 Classes ............................................................................................ 17
      3.2.2 Methods .......................................................................................... 18
      3.2.3 Special Variables ............................................................................ 18
   3.3 Used Predefined Classes ......................................................................... 18

4 The Mapping of POOSL on Smalltalk ............................................................. 21
   4.1 Introduction ............................................................................................. 21
   4.2 Required Information ............................................................................. 21

5 The Mapping of POOSL on C++ .................................................................. 23
   5.1 Introduction ............................................................................................. 23
   5.2 Required Information ............................................................................. 23

6 Compiler Construction .................................................................................. 25
   6.1 Introduction ............................................................................................. 25
   6.2 Compiler Properties .............................................................................. 25
      6.2.1 Scanner ............................................................................................ 26
      6.2.2 Parser .............................................................................................. 26
   6.3 Why a Compiler Construction Tool? ..................................................... 27

7 T-gen ............................................................................................................. 29
   7.1 Introduction ............................................................................................. 29
1 Introduction

1.1 Project Context

The design of current information-technology products becomes increasingly difficult because of their growing complexity. Most of these products contain information-processing systems that consist of a combination of software and hardware components. To support development of these complex systems, adequate analysis, specification and design methods are required. At the section Information and Communication Systems of the faculty Electrical Engineering at the Eindhoven University of Technology, development of these methods is subject of active research. This research resulted in the method SHE (Software/Hardware Engineering) [PV97][PVS95].

SHE is an object-oriented method for co-development of complex reactive software/hardware systems, covering analysis, specification and design. SHE incorporates the formal specification language POOSL (Parallel Object-Oriented Specification Language) [Voe95a][Voe95b]. In addition, a set of behaviour-preserving transformations is supported by this method [VPS96]. These transformations modify a system with respect to for example architecture and topology, but preserve functional behaviour.

SHE starts with an informal object-oriented analysis and architecture design and subsequently produces rigorous behaviour and architecture descriptions specified in POOSL. During analysis and design POOSL specifications are gradually transformed to incorporate architectural decisions and design constrains. For a complete description of the SHE method, see [PV97].

To create a useful environment for developing complex systems with SHE, software tools that support the specification process are indispensable. Because of the formal syntax and semantics of POOSL, SHE has the potentials for formal verification, transformation, compilation, simulation and implementation.

1.2 Objectives

The objective of this master’s project is to design a compiler for POOSL, which is able to translate POOSL specifications to Smalltalk [PPS92] and to C++ [Str92] implementations. To accomplish this, a syntax for POOSL is needed, which is suitable for computer automated interpretation. This (concrete) syntax and accompanying grammar of POOSL is formulated in [Kup96]. In addition, it is necessary to have a specification of by what means POOSL can be translated to the target code, i.e. a
mapping scheme. The mapping of POOSL on Smalltalk is designed and formulated by Mark Geilen, who is implementing the POOSL-simulator. The mapping for translating POOSL to C++ is designed and formulated in [Tet97]. Translations to both Smalltalk and C++ can be seen as a translation to an intermediate code, because in both cases supporting target code is required to implement the semantics of POOSL. In Smalltalk, this support is implemented within the POOSL-simulator and for C++, it is implemented by the C++ library of [Tet97].

The results of [Kup96], Mark Geilen and [Tet97] are used as a starting-point for the implementation of the POOSL-compiler. Figure 1-1 shows how these results are related to the POOSL-compiler.

\[\text{POOSL Specification} \rightarrow \text{POOSL-Compiler} \]

\[\text{Syntax & Grammar} \rightarrow \text{Mapping to Smalltalk} \rightarrow \text{Smalltalk Code} \]

\[\text{Mapping to C++} \rightarrow \text{C++ Code} \rightarrow \text{C++ Library} \]

\[\text{Figure 1-1: Project Overview.}\]

A simulator tool is used for verification and validation purposes of a system, i.e. for checking whether the system model satisfies informal requirements. To encourage the industry to use the SHE method a simulator tool is indispensable. Because of the properties, especially for simulation, the Smalltalk programming environment is chosen for implementing the POOSL-simulator. To make the simulator able to interpret a POOSL specification, a compiler is needed for translating POOSL to Smalltalk. Because of the necessary interaction of the simulator with the compiler, it is obvious to implement the POOSL-compiler also in the Smalltalk programming environment.
A compiler that translates POOSL specifications to C++ is very useful for automated (prototype) implementations and (fast) simulations. The language C++ is chosen because it is one of the most widely used and accepted object-oriented programming languages in the industry. It is possible to translate C++ to various different target (platform) codes. The compiler for translation to C++ can easily be deduced from (and integrated within) the compiler that translates POOSL to Smalltalk.

### 1.3 Report Organisation

This thesis is organised into 11 chapters, as follows.

**Introduction.** In Chapter 1, the context and the objectives of this master’s project are described.

**The Language POOSL.** In Chapter 2, an introduction to POOSL is given. In addition, the modified POOSL syntax and accompanying grammar are formulated.

**The Smalltalk Environment.** In Chapter 3, an introduction to the Smalltalk programming environment is given. The Smalltalk environment is used for implementing the POOSL-compiler.

**The Mapping of POOSL on Smalltalk.** In Chapter 4, the mapping of POOSL on Smalltalk is given. The problems, which will be encountered to translate POOSL to Smalltalk according to the mapping, are discussed.

**The Mapping of POOSL on C++.** In Chapter 5, the mapping of POOSL on C++ is given. The problems, which will be encountered to translate POOSL to C++ according to the mapping, are discussed.

**Compiler Construction.** In Chapter 6, the general properties of a compiler are explained. In addition, the reason why a compiler construction tool is used to implement the POOSL-compiler is explained.

**T-gen.** In Chapter 7, the functionality of the compiler construction tool T-gen (Translator-Generator) is explained. T-gen is used to implement the POOSL-compiler.

**Parse Tree Construction.** In Chapter 8, the construction of the parse tree by the POOSL-compiler is explained. The used parse tree nodes and the tree building strategy are discussed.

**Parse Tree Decoration.** In Chapter 9, the decoration of the parse tree is explained. The traversal of the tree is explained. In addition, the way of rearranging information within the tree, which is needed for code generation, is discussed.
**Code Generation.** In Chapter 10, the code generation for Smalltalk and C++ are discussed. Because of tree decoration, all necessary information is now available for code generation.

**Conclusions, Results and Future Work.** In Chapter 11, the conclusions and the results of this master's project will be presented. In addition, some recommendations for future work will be discussed.
2 The Language POOSL

2.1 Introduction

The language POOSL is used to formalise the informal models produced by the SHE method. POOSL is based on the object-oriented paradigm to support flexible and reusable design. The language is composed of a process and a data part, which consists of cluster and process classes and respectively data classes. These classes will be briefly explained in the next sections. For a complete description of the POOSL language, see [PV97].

2.2 The Basics of POOSL

A specification in POOSL consists of a fixed number of process objects and clusters of process objects. Process objects are concurrent entities, but their internal behaviour is sequential. Processes and clusters are statically interconnected in a topology of communication channels, through which they can communicate by exchanging messages. Message exchange is based upon the pair-wise message passing (rendezvous) mechanism (of CCS [Mil80]). Processes and clusters are used for modelling the behaviour and architecture of a system.

Data objects are sequential entities that are used for dynamical creation and flowing of information through a system. Data objects have similar properties as traditional objects defined in sequential object-oriented languages such as Smalltalk and C++.

2.2.1 Data Class

Data objects are instantiated from data classes. The behaviour of the objects of a data class is specified in a data class definition. Data classes can have instance variables and methods. Instance variables implement the attributes of information a system may need. They contain (references to) other data objects. Methods describe the behaviour of how objects respond to the reception of messages.

There are four (five) primitive data classes of generally used data types, namely Boolean, Integer, Real, and Char (and String). Primitive data objects are the basic building blocks for other data objects. The methods of primitive data classes define the usual operations such as relational and mathematical operations. The primitive data classes are extended with the primitive data objects bunk, iunk, runk and cunk to represent the 'unknown'
values in the respective classes Boolean, Integer, Real, and Char. They are used to model data abstraction. Another special primitive object is nil, which represents the object to which a variable refers when it has not been assigned a value yet.

2.2.2 Process Class

Process objects are instantiated from process classes. The behaviour of the objects of a process class is specified in a process class definition. Process classes can have channels, instance variables and methods. Processes communicate by sending messages to each other through channels. When data objects are passed by communication between processes, deep copies of these data objects are made. Deep copies are complete copies of the structure of objects including all objects that are indirectly referenced. Processes do not share any data objects. Processes can contain internal data in the form of data objects stored in instance variables. Methods describe behaviour of the process and are comparable with procedures of imperative programming languages such as C or Pascal.

2.2.3 Cluster Class

Next to processes, POOSL supports clusters. A cluster is a boundary around a group of collaborating processes and other clusters. Clusters are instantiated from cluster classes. Clusters are built from other clusters or processes by using parallel composition, channel hiding and channel renaming. Clusters are specified in a cluster class definition.

Clusters are useful for creating hierarchy in a system specification. By using clusters, it is also possible to indicate some other boundaries with respect to for example implementation, concurrency and distribution.

2.3 The Syntax of POOSL

For creating a POOSL compiler, a syntax for specifying POOSL is required. A concrete (parseable) syntax is necessary, because the abstract syntax defined in [Voe95b] does not allow unambiguous parsing (see Section 6.2.2). The concrete syntax of POOSL and the accompanying context-free grammar in Backus-Naur Form (BNF) is initially formulated in [Kup96].

An attempt was made to modify the concrete syntax of [Kup96], to let it correspond more to the abstract syntax concerning the use of data statements and expression within a process method. However, to accomplish this, the grammar would become much more complex due to avoidance of ambiguity. Therefore this attempt was dropped.
In appendix A, the current concrete syntax of POOSL is illustrated by syntax diagrams (also called railroad diagrams). The syntax diagrams reflect the properties of the context-free grammar, i.e. they indicate what the POOSL-compiler accepts\(^1\). In Appendix B, the context-free grammar used to implement the POOSL-compiler is given in Backus-Naur Form.

In these syntax diagrams, a rounded box reflects a terminal, i.e. a literal string in the POOSL code. Some rounded boxes contain several characters enclosed in square brackets, which define an alternation of possible characters (see also Section 7.3). A square box refers to a non-terminal, i.e. a reference to another syntax diagram.

2.3.1 Changes

To correspond more to the abstract syntax and to remove some impractical syntax constructs, the concrete syntax of [Kup96] is subjected to a number of small modifications.

**ProcessClass:**
- An initial method call can no longer have any return parameters. Previously the syntax allowed an initial method call to have return parameters, which was not in accordance with the abstract syntax.

**ProcessStatements:**
- The type (data class) of the return parameters of a method call is omitted, because the type is already defined in the method definition. For example, `methodO(a, b : Bit)` is now `methodO(a, b)`.
- Select statements now have to have at least two alternatives. Previously the syntax allowed `sel S les;` which is not in accordance with the corresponding choice statement (or) in the abstract syntax.
- To avoid successive semicolons between statements, a composition of statements may no longer have ‘empty’ statements.
- When using a composition of statements as part of an abort or interrupt statement, round brackets must be placed around the composition. The round brackets may be omitted if only a single statement is used. For example, `(S1; S2) abort (S3; S4)` (previously this was `(S1; S2 abort S3; S4)`)
- Communication messages now may have no parameters but may still have round brackets. Previously the syntax only allowed that when a communication message had no parameters the round brackets had to be omitted. For example, `ch!ack` may now be `ch!ackO` as well as `ch!ack()`.

**DataClass:**
- The list of instance variable declarations in a data class definition now may be empty, which was previously not allowed.

---

\(^1\) Without taking context conditions into account.
DataStatements:

- A return statement may occur everywhere in a data method (early return). Context checks must determine if a method always terminates with a return statement. This is done to make it is possible to write, if E then S1; return E else S2; return E fi.
- A variable or self can be sent successive messages. Previously the syntax only allowed one message per expression. For example, x := self m1(); x m2(); can now be stated in one expression self m1() m2().
- The (data) select statement has been removed. A select statement (or choice statement) within a data method is not part of the abstract syntax.

Expressions:

- The notation of unary Boolean not-operator has been changed from ! to not. This is done because it is a more object-oriented notation that respects the sending of messages to objects. For example, !bool is now denoted bool not.

Literals:

- Identifiers (<ident>) now may have successive underscores.
- Data classes Char and String (<character>, <string>) are extended with more characters.
- Data class Integer (<int>) is extended with hexadecimal (<hexNumber>) and binary (<binNumber>) notation.
- Comments (<comment>) may be nested.

2.4 Context Conditions

Context conditions are the (syntactic) requirements of a language, which have to be satisfied and which can not be described in Backus-Naur Form. The context conditions of POOSL are described in [PV97]. Due to the introduction of an early return statement (see above), an additional context condition is introduced. This context condition implies that a method always must terminate with a return statement.

Currently the POOSL-compiler does not check any context conditions. An additional traversal of the decorated parse tree (see Chapter 9) can be performed to determine whether all context conditions are met. Some information will already be available in the decorated parse tree but the decoration procedure will have to be extended to let the context checker be able to check all context conditions.
3 The Smalltalk Environment

3.1 Introduction

Because of the properties, especially for simulation, the Smalltalk programming environment [PPS92] is chosen for implementing the POOSL-simulator. To make the simulator able to interpret a POOSL specification, a compiler is needed for translating POOSL to Smalltalk. Because of the interaction of the simulator with the compiler, it is obvious to implement the POOSL-compiler also in the Smalltalk programming environment. This chapter will describe the basics of Smalltalk.

3.2 Basics of Smalltalk

Smalltalk is an object-oriented programming environment. A Smalltalk application is implemented and operates within the Smalltalk environment. All entities within the Smalltalk environment are objects. A Smalltalk application consists of a collection of data objects that interact with one another via built-in routines called methods. An object typically is made up of one or more private variables, the data, combined with a set of methods for manipulating that data.

3.2.1 Classes

A class defines the behaviour (instance methods) and a data template (instance variables) of similar objects. When an object is instantiated by a class it gets its own instance variables and it can make use of the instance methods of the class. A class itself is also an object, it can have its own variables (class variables) and messages (class methods) can be sent to it. The Smalltalk environment consists of a large number of predefined classes.

Smalltalk incorporates a class hierarchy with inheritance. Inheritance is a powerful feature in object-oriented programming. A subclass inherits the methods and variables of its superclass, i.e. a class higher in the hierarchy. Smalltalk provides a method-lookup mechanism that starts its search for a given method in the class of the object to which the message was sent. If no such method exists there, the mechanism climbs up in the hierarchy, stopping at each level to look for the method.

Sometimes a class has no instance variables and is not intended to have instances, but its behaviour is inherited and used by its subclasses and their instances. These special
classes are usually called abstract classes. Abstract classes are frequently useful as a repository for methods that are useful for two or more classes, none of which is a logical subclass of the other.

### 3.2.2 Methods

Smalltalk objects communicate by sending messages to each other. The messages they send consist of a selector, the name of the message, and zero or more arguments. A method is the definition of the behaviour of an object (receiver) that is sent a message. Methods can access the instance variables of the receiver and may have their own local variables. In the predefined classes of Smalltalk, the most common methods that are often used are built-in.

### 3.2.3 Special Variables

There are three special variables, for which the value changes according to the execution context: `self`, `super` and `thisContext`. They cannot be changed by an assignment.

`Self` holds a reference to the object that is executing the current message. `Self` does not necessarily point to an instance of the class whose method is being executed, because a method can be implemented in a superclass of the object to which `self` is pointing.

`Super` is very similar to `self`, except `super` tells the method lookup mechanism to begin its search one level above the executing method in the class hierarchy. This is useful when a subclass wants to add operations to its parent's method, without duplicating code.

The third special variable, `thisContext`, is a reference to the stack context of the current process, which is rarely used by a common programmer.

### 3.3 Used Predefined Classes

In this section, some predefined Smalltalk classes, which are used by the POOSL-compiler, will be explained briefly. A more extensive explanation can be found in [PPS92].

**Boolean**

The class `Boolean` consists of two subclasses, namely `True` and `False`. These classes contain the constant instances `true` and `false`. The instance methods implement logical and conditional operations.
Integer
Integers implement the representation natural numbers. The Integer class is the abstract superclass of SmallInteger, LargePositiveInteger and LargeNegativeInteger, of which SmallInteger is bounded. The instance methods implement arithmetical and iterative operations.

String
The class String represents an array of characters. An instance of String is created by sending the message new to the class String or by writing a sequence of characters between single quotes (‘abc’). If in the latter case a single quote is to be included, it must be preceded by a single quote. The instance methods implement common String operations for concatenation, copying and comparing.

Symbol
The class Symbol is a subclass of String. Instances of Symbol are Strings that are represented uniquely, i.e. there is only one instance of each symbol in the Smalltalk environment. A Symbol is preceded by a number sign and optionally enclosed in single quotes (#‘abc’). Because Symbol is a subclass of String, it inherits most methods of String.

OrderedCollection
The class OrderedCollection represents a collection of objects explicitly ordered by the sequence in which objects are added and removed. The objects are accessible by external keys that are indices (Integer). With several instance methods object can be added at different places in the collection. An OrderedCollection, as most Smalltalk collections, responds to several messages that perform collection iteration, i.e. code that is performed for each object in the OrderedCollection.

Set
The class Set represents an unordered collection of elements that are not duplicated. An object is added to the Set if and only if it not already contained an object with the same value (determined by the method =). A Set responds to collection iteration, detection messages and some arithmetic set operations.

Dictionary
A Dictionary is a set of associations, which contain a key and an associated value. A Dictionary is useful for look up objects that are associated with other objects. The keys of a Dictionary form a Set, so there can be only one key with the same value. A Dictionary responds to collection iteration and detection messages.
4 The Mapping of POOSL on Smalltalk

4.1 Introduction

The POOSL-compiler does not have to be able to translate a complete POOSL specification to Smalltalk. Only process methods, initial method calls, data methods and expressions must be translated, because all POOSL classes are stored in the POOSL-simulator. However, the POOSL-compiler must be able to parse and decorate a complete specification, so the POOSL-simulator is able to store the classes of the specification by sending messages to nodes within the parse tree.

In Appendix C, a translation table for translating POOSL to Smalltalk is given. The Smalltalk code that has to be generated by the POOSL-compiler can be seen as an intermediate code, because most of the semantic properties of a POOSL specification are implemented by the POOSL-simulator (developed by Mark Geilen).

4.2 Required Information

From Appendix C it arises that each statement and expression must be able to generate its own specific Smalltalk code. In addition, some information needs to be added and rearranged to be able to generate Smalltalk code. How this is accomplished is explained in Section 9.3.

Each statement and expression will need a number to generate code for `<slf>` (see Section 9.3.1), which is used to create a reference to the current data object or executing process. The Smalltalk code that has to be generated for a process method call depends on whether it is tail recursive (see Section 9.3.2). Therefore, a process method call must be aware whether it is tail recursive. Variables must be aware of their scope (local or instance) to be able to generate the appropriate Smalltalk code (see Section 9.3.6). Each statement needs a unique number, generated as a parameter of the message `checkPoint: nodeNumber`, to identify the statement. This is used to create a link between the POOSL-specification and the corresponding Smalltalk code (see Section 9.3.11).

The POOSL-simulator requires additional information, i.e. the class name, the instantiation parameters, the instance variables, the initial method call, et cetera. The POOSL-simulator must also be able to retrieve the used data classes, used communication channels and the used message interface of process methods (see Section 9.3.3, 9.3.4 and 9.3.5).
5 The Mapping of POOSL on C++

5.1 Introduction

The POOSL-compiler can only translate a complete POOSL specification, because all POOSL classes must be available for translation to C++.

In Appendix D, a translation table for translating POOSL to C++ is given. The C++ code that has to be generated by the POOSL-compiler can be seen as an intermediate code, because most of the semantic properties of a POOSL specification are implemented by the C++ support library for POOSL [Tet97].

5.2 Required Information

From Appendix D it arises that each POOSL class has to generate its own header and source file. Further a method, statement and expression must be able to generate its own specific C++ code. In addition, some information needs to be added and rearranged to be able to generate C++ code. How this is accomplished is explained in Section 9.3.

The C++ code that has to be generated for a process method call depends on whether it is tail recursive (see Section 9.3.2). Therefore, a process method call must be aware whether it is tail recursive. Variables must be aware of their scope (local, input, return, instantiation or instance) to be able to generate the appropriate C++ code (see Section 9.3.6). In Appendix D is indicated by \texttt{INPUTPARAM}(x), \texttt{RETPARAM}(x) and \texttt{LOCALVAR}(x) whether variables of a specific scope has to be passed to a C++ member function call. This depends on whether a variable is used within the member function (see Section 9.3.7).

Some translations of POOSL constructs have to generate C++ member functions. These member functions have to be generated further in a source file and they must be prototyped in a header file. How this is accomplished is explained in Section 9.3.8.

The channel interconnection structure of a system or a cluster must be computed of their behaviour specification. The channels entries of the instances, channel hiding and channel relabeling within behaviour specifications determine the interconnection of the instances. The interconnection structure is needed to generate C++ code for declaration, creation and interconnection of C++ channel and connector objects (see Section 9.3.13).
6 Compiler Construction

6.1 Introduction

This chapter will explain what the properties of a compiler are, what its constituents are and which stages follow each other during compiling. In addition, the reason is given why a compiler construction tool is used to develop the POOSL-compiler.

6.2 Compiler Properties

In a computer-oriented environment, it is often necessary to translate a structured textual specification either into a different textual representation or into some internal data representation. A special case of this general translation process is compilation, where a source program is translated into some executable machine program or target code. A compiler is a tool for automated translation of such structured textual specifications.

The compiling process consists of three stages:
1) Parsing; building the parse tree
2) Decoration; rearranging information within the parse tree
3) Code Generation; generating target code.

The relation between the compiling stages is visualised in Figure 6-1.

![Diagram of Compiling Stages]

Figure 6-1: Compiling Stages.
A scanner and a parser are parts of a compiler that interact with each other to create a parse tree. A parse tree is an internal data structure that represents the structure of the textual input. After the tree is built, it is decorated, i.e. additional information that is needed for the compilation process is added to the tree. The decorated tree is used for the post-parsing steps, i.e. checking of the context conditions and generating the target code.

During compilation syntax errors and context condition errors can occur and must be handled appropriately. Syntax errors can be found by the scanner or parser, but context conditions must be checked separately by a context checker. This context checker will use the decorated parse tree to determine context condition errors.

The general properties of scanners and the parsers will be explained in the next sections. The parse tree building within the POOSL-compiler will be explained in chapter 8. The last two stages, tree decoration and code generation within the POOSL-compiler will be explained in chapter 9 and 10.

### 6.2.1 Scanner

A scanner is used for dividing the input text up into smallest semantic portions, i.e. tokens, of text. Tokens, also called terminals, can be defined in two varieties, token classes and literals. Token class definitions, which are usually specified by regular expressions, declare a composition of characters that may be accepted. Literals are constants that are composed of one or more fixed characters. The semantic values of the tokens are passed to a parser.

### 6.2.2 Parser

A parser has to determine the structure and validation of the input text. The structure recognised by the parser is usually specified by a context-free grammar (Appendix B). This context-free grammar must be prepared for unambiguous parsing, which means that the structure can only be derived in one unique manner with respect to the semantics of that structure. When a parser can derive a structure in different ways, which are also semantically different, the parser is not able to consider this semantic difference.

To determine if the input text is valid, the parser accumulates the tokens supplied by the scanner. Then the parser determines if a sequence of tokens satisfies one of the grammar rules. Sometimes a parser needs more than one token to determine which structure it is parsing, therefore a scanner can be asked to look ahead in the input. The scanner then is called a look-ahead scanner.

The structure recognised by the parser is stored in an internal data structure, a parse tree. A parse tree consists of nodes that contain an arbitrary number of subnodes and leaves. The leaves of the tree generally contain the required values of tokens.
6.3 Why a Compiler Construction Tool?

There are two major ways to develop a compiler. One way is to write a compiler single-handedly. For a language as complex as POOSL a structured approach, i.e. specifying the context-free grammar and then in accordance with some (standard) rules implement the productions rules, is inevitable. Otherwise, it becomes unmanageable to program a maintainable compiler.

Fortunately, the formalisms behind compiler construction from a context-free grammar specification are well understood. Therefore, it is possible to write computer programs/algorithms that automatically build compilers from context-free grammar specifications. Such programs are called compiler construction tools. So another way to build a compiler is to use a compiler construction tool.

Using a compiler construction tool makes the compiler highly maintainable because changes in the grammar can automatically be adapted by the compiler. Changes in the grammar of POOSL are very likely because POOSL is still under construction. It is for this reason that it is chosen to implement the POOSL compiler with a compiler construction tool. The used compiler construction tool is discussed in chapter 7.
7 T-gen

7.1 Introduction

The compiler construction tool within the Smalltalk programming environment that is used for the POOSL compiler is called T-gen (Translator-Generator, Version 2.1) [Gra92]. T-gen is a general-purpose object-oriented tool for the automatic generation of string-to-object translators. T-gen automatically generates a scanner and a parser from given token class definitions and a grammar specification.

7.2 Features of T-gen

T-gen includes a number of advanced features. T-gen supports:

- Zero, one and two token look-ahead scanner.
- Common grammars: LL(1), SLR(1), LALR(1) and LR(1) [ASU86][FL88].
- Automatic derivation tree generation.
- User-defined parse tree building.
- Creation of T-gen independent scanner and parser classes.
- Interaction within a graphical user-interface.

A negative aspect of T-gen is that it does not support overlapping token classes. Hence, T-gen attaches no significance to token class definition ordering as LEX [Les75], which does support overlapping token classes.

7.3 T-gen Token Class Specification

A T-gen token class specification is a sequence of token class definitions. A token class is defined by its token class terminal, followed by a colon, a regular expression, and an optional scanner directive and terminated by a semicolon.

For example:

```
<ident>    :  [ABCa-zA-Z0-9_]* ;
<whitespace>  :  \s* {ignoreDelimiter} ;
<binNumber>  :  \d* ;
```


Here the square brackets define an alternation of possible characters. For instance, 
\[\text{[ABCa-z]}\] means the characters \text{A}, \text{B}, \text{C} and all ASCII characters between \text{a} and \text{z}. The star symbol, *, means zero or more repetitions of the regular expression preceding it (Kleene Closure). The plus symbol, +, means one or more repetitions of the regular expression preceding it. A backslash is used for escaped characters. These characters are used to define the regular expression (*, -, +, et cetera) and non-printable characters (newline: \v, tab: \t, et cetera). The vertical bar, |, is used for alternation. The Smalltalk method selector between braces (\text{ignoreDelimiter}) is a scanner directive, which represent a message that is sent to the scanner when a token of this class is recognised. With scanner directives, the scanner is able to perform actions on the tokens.

The literal tokens are extracted from the grammar specification. They are treated as if they belong to their own singleton (one-element) token class.

The predefined scanner directive \text{ignoreDelimiter} just skips the current token and scans the next token. This is used to skip white space, i.e. spaces, tabs, new-lines and carriage-returns. To let the scanner be able to scan POOSL comments, one special scanner directive is added namely \text{skipComment}. This directive is used to just skip the comments that are stated within the POOSL code. POOSL comments are started with '"/*' and ended with '"*/', and may be nested within another (see Appendix A). This can not be expressed with a regular expression and therefore it is scanned by the \text{skipComment} algorithm after detection of the first '"/*'.

Additional information about specifying token class definitions in T-gen can be found in [Gra92]. The token class specification of the POOSL language is given in Appendix B.1.

### 7.4 T-gen Grammar Specification

A T-gen grammar specification is a sequence of production rule specifications similar to those used by YACC [Joh75]. A production rule is defined in Backus-Naur Form (BNF) and it specifies the permitted structure of a language. Production rules consist of a (left-hand-side) non-terminal, followed by a colon, one or more right-hand-side specifications separated by vertical bars and terminates by a semicolon. A right-hand-side specification is a regular expression that consists of terminals and non-terminals, optionally followed by a parse-tree-builder directive. A parse-tree-builder directive is a translation symbol, which is either a Smalltalk class identifier or a method selector. The parse-tree-builder class is an instance of (subclass of) the class \text{AbstractSyntaxTreeBuilder}, which interacts with the parser to create the parse tree. The POOSL-compiler uses the class \text{POOSLAbstractSyntaxTreeBuilder} as parse-tree-builder. The parse-tree-builder builds parse trees by creating specific objects for each kind of node as indicated by the translation symbols in grammar productions.
For example:

```plaintext
DataExp : Var MethodCalls {var:calls;}
| 'new' '(' ClassName ')' {NewNode}
| '<ident>' {VarNode};
```

When the translation symbol is a method selector (`var:calls:`), the parser sends the message to the parse-tree-builder. The non-terminals in the right-hand-side of the production are passed as arguments to the method. Therefore, the number of arguments the method expects must be equal to the number of right-hand-side non-terminals.

When the translation symbol is a class identifier (`NewNode`), which is usually a subclass of `ParseTreeNode`, the parse-tree-builder automatically sends the message `new` to that class to create a new node. If the node has subnodes (children) the `addChildInitial: children` message is sent to it. The method `addChildInitial:` has to place the argument `children`, an instance of `OrderedChildren`, in the appropriate instance variables of the node. `OrderedChildren` is a subclass of `OrderedCollection`. However if a right-hand-side of a production only consists of a token class identifier (`<ident>`), the message `setAttribute: string` is sent to the node. The method `setAttribute:` has to place the argument `string`, an instance of `String`, in the appropriate instance variable of the node.

There exist three special predefined parse-tree-builder directives: `nil`, `liftLeftChild` and `liftRightChild`. The selector `nil` is used to associate the value nil with the left-hand-side non-terminal. The other two directives are useful for adding arbitrary numbers of children to parent nodes. The selector `liftLeftChild` instructs the parse-tree-builder not to create a new node when the associated production is recognised. Instead, it stacks the association of the most right non-terminal. When the production is completed all stacked associations are unstacked and added as right-most children of the association of last non-terminal of the production. Consider for example the following grammar fragment:

```plaintext
List : List Ident {liftLeftChild}
| Ident {IdentListNode} ;
Ident : '<ident>' {IdentNode} ;
```

If for example, three successive `<ident>` tokens are given as input, first an instance of `IdentNode` with the first token is created and put in a child collection (`OrderedChildren`) to be passed to its parent node. Then the other two `IdentNodes` are created and they are added to the collection as right-most children. The latter is caused by the `liftLeftChild` directive, which instructs the parse-tree-builder to send the message `addChildLast: children` to the first `IdentNode`. After that an `IdentListNode` is created and the child collection is passed to it. Accordingly, an `IdentListNode` holds a collection of all the `IdentNodes`, its first child being an `IdentNode` containing the left-most token. The directive `liftRightChild` works in an analogue fashion but with the accompanying message `addChildFirst: children`.

Additional information about grammar specification in T-gen can be found in [Gra92]. The complete grammar specification of the POOSL language is given in appendix B.2.
7.5 T-gen Remarks/Modifications

T-gen automatically generates a scanner class (*POOSLScanner*) and a parser class (*POOSLParser*), which are used for parsing within the POOSL-compiler. The generated parser is a LALR(1)-parser, i.e. a type of deterministic bottom-up shift-reduce parser with one token look-ahead. This bottom-up parser is a finite state automaton that builds the parse tree in post-order. Shift and reduce are actions within the parser. A shift action is performed when the input token matches one of the acceptable input tokens for the current state of the parser. The value of that token is stacked and the next state of the parser is determined by the previous state and the token. A reduce action is performed when the parser has succeeded in matching all the elements of a grammar production rule and the next token is not erroneous. The tokens and states belonging to the production on the parser stack are removed and eventually used by some parse-tree-builder action (code). Then the new state of the parser is determined and shifted on the stack.

The POOSL-simulator must be able to trace the execution and report errors with respect to the POOSL code. Because the POOSL-simulator only holds a reference to the parse tree, a link between the parse tree and the corresponding POOSL code is required. To accomplish this link each node contains an instance variable *interval*, which holds the begin position and end position of the POOSL code it corresponds to. The instance variable *interval* of each node is assigned during parsing.

For computation of *interval* of each node, some T-gen scanner and parser classes are modified. In the T-gen scanner classes, *FSABasedScannerWithOneTokenLookahead* and *OptimizedScannerWithOneTokenLookahead* the *scanToken* method is modified with calculation of begin position and end position, i.e. interval, of the token. This interval is passed to the parser. In the T-gen parser class, *LR1Parser* the method *parseWithTreeBuilder:* is modified to push and pop the intervals on an interval stack (*intervalStack*). When the parser performs a shift action, it just pushes the current interval on the interval stack. When the parser performs a reduce action, it pops and unites a number of intervals and then pushes the united interval back on the interval stack. The number of intervals (*nrOfIntervals*) that is combined equals the number of the right-hand side terminals and non-terminals. The super class of all POOSL parse tree nodes, i.e.ystem *POOSLParseTreeNode*, is assigned a reference to the parser so it can access *intervalStack* and *nrOfIntervals*. When a translation symbol results in the creation of a parse tree node, the message *new* is received by *POOSLParseTreeNode*. Within this message a new node is created and initialised by the message *init*, wherein a *nrOfIntervals* of *intervalStack* is popped and united. The united interval is assigned to the instance variable *interval* of the new node and it is pushed back on *intervalStack*.

Consequently, a translation symbol may result in the creation of at most one instance of a subclass of *POOSLParseTreeNode*. Otherwise, the intervals on the stack will be united erroneous.
8 Parse Tree Construction

8.1 Introduction

For some (simple) languages, it can sometimes suffice to generate target code without constructing a parse tree (on-the-fly code generation). In most practical cases however, it is necessary to create a parse tree, so the post-parse stages can use and extend this data structure.

To explain how a parse tree is built first the concept of derivation trees is explained. A derivation tree is a tree that represents each grammar production rule that was applied during parsing of an input text. The root of the derivation tree is the start symbol of the grammar specification. A derivation tree is a precise representation of the parse in that each step of the derivation is represented by some node in the tree. Derivation trees are useful in the early stages of creating and debugging a grammar specification.

However, derivation trees often add a lot of clutter to the structural interpretation of the input text. Parse trees, also called abstract syntax trees, are essential derivation trees without such clutter. They only store the information that is required. Since only selected portions of the derivation will actually be represented in the parse tree, the parser needs to know which rules create nodes and how. This information is specified in the parse-tree-builder directive of each grammar production rule.

Conceptually a parse tree is constructed by performing a post-order traversal of the derivation tree, storing only the required information. For a tree T with root r and subtrees T₁,..., Tₙ, the post-order of T means, the post-order of T₁,..., followed by the post-order of Tₙ followed by r. During the traversal nodes are created to support the structure of the parse tree or previously created nodes move up in the tree along with the traversal to be incorporated as children of future nodes.

8.2 Parse Tree Nodes

The parse tree nodes used by the POOSL-compiler are implemented as Smalltalk classes. The parse tree classes are ordered in a hierarchy in relation to cluster, process and data classes and data expressions. Instance variables of the nodes contain (references to) other nodes, leaves (instances of String) or they can store additional information that is rearranged by tree decoration.
In what way the nodes are created, depends on the translation symbol in the grammar specification. They can be created by means of a translation symbol that is a class name identifier or a method selector. How this is accomplished was explained in section 7.4.

All nodes respond to the message \textit{decorate: parameter}, which is used for adding and retrieving information during tree decoration. Most nodes also respond to the message \textit{emitST: parameter} and \textit{emitCPP: parameter} for respectively generation of Smalltalk and C++ code.

All parse tree node classes used in the POOSL-compiler are given in appendix E.1. The tree node classes are categorised in a hierarchy based on their relation to cluster class, process class data class and data expression.

8.3 Tree Building Strategy

The parse tree must contain all required information that was present in the POOSL specification. This information must be stored in the tree in a manner that it is easy to access, comprehend, and conceive.

The tree-building method of each node, i.e. initialisation method, has been kept as generic as possible. This initialisation method only assigns the instance variables and does nothing else. Therefore, it is possible to see how the tree is constructed by looking (almost only) at the grammar specification.

There are many different parse tree node classes, which represent just a small part of a POOSL specification. Due to this fine-grained division, the behaviour of each node can be apparent and uncomplicated. For parts of a POOSL specification that (probably) need to generate distinct target code, different types of nodes are available. Future translations can take advantage of this.

Nodes that have an instance variable that can refer to multiple subnodes imply that the instance variable always refers to an instance of \textit{OrderedChildren}, even when there is no or just one subnode. With this, the tree structure is kept as generic as possible to facilitate post-parsing steps, i.e. post-parsing steps need less checking for existence of subnodes.
8.4 Tree Structure

To give an idea of how the structure of the parse tree looks like, a visualisation of a parse tree is shown in Figure 8-1. This parse tree is based on the following POOSL code of a data statement.

\[
\text{count} := - \text{count msg1()} \text{msg2()} + 12;
\]

In the rectangular boxes, representing the parse tree nodes, the Smalltalk parse tree node class and its instance variables are shown. The lines between the nodes represent references between them. These references determine the structure of the parse tree. The rounded rectangles represent the leaves of the tree. Leaves contain instances of String, which literally correspond to parts of the POOSL code. The numbers next to the nodes indicate the order of creation (post-order) during parse tree building.

Figure 8-1: Parse Tree Structure Example.
9 Parse Tree Decoration

9.1 Introduction

The parse tree contains all required information that was present in the POOSL specification. However, some information will not be optimally accessible in the way it is useful for the translation to a specific target code. Therefore, the parse tree is traversed to rearrange and move information so it is accessible where it is needed. This post-parsing step is often called parse tree decoration.

Parse tree decoration for Smalltalk and C++ within the POOSL-compiler will be explained in the next sections.

9.2 Tree Traversal

Each parse tree node is aware of how to decorate itself and to provide information for decoration of its children. Therefore, when a parse tree node is requested to decorate itself, the node will perform its (first part of) decoration and provide information for its children and then it subsequently requests its children to decorate themselves. After its children have finished decoration, the decoration procedure returns to the node, which is now able to store information provided by its children.

While traversing the tree a parameter is passed along with the decoration message \texttt{decorate: parameter}. This parameter is an instance of (a subclass of) \texttt{DecorateParm}, which contains information that is needed further down or up in the tree. Information can be added and extracted from this object during traversal, i.e. nodes can retrieve information about nodes above them and they can add information to provide nodes below them. In addition, when the decoration of the children of a node is finished, the decoration message returns to the node, so it can store information provided by its children. In Figure 9-1, the traversal of the parse tree during decoration is visualised.
No instance variables of the nodes that are given a value at tree building are changed during decoration; they hold the same value as before decoration. This is done for other post-parsing steps in order to provide them with the same values.

The hierarchy of class DecorateParm and subclasses (see Appendix E.2) is only to indicate in which parts of the tree the information of the parameter is modified.

### 9.3 Tree Decoration Parameter

The decoration parameter, an instance of (a subclass of) DecorateParm, contains several instance variables that are used for various purposes. The main purpose of the instance variables is to transport information that has to be rearranged.

The properties of the instance variables of parameter will be explained in the next sections. Each section will explain of one of these instance variables how its value is computed and what its purpose is.

#### 9.3.1 Level

During Smalltalk code generation, the String ‘\$lf_i’ is generated in front of each message and within each Smalltalk code block (see Appendix C). The i in ‘\$lf_i’,
stands for the String representation of an Integer. This is used (internally) by the POOSL-simulator to refer to the executing process or to the current data object.

The instance variable level of the decoration parameter is an Integer, which is incremented during the decoration of a ProcessSelectNode, ProcessGuardedCmdNode, ProcessInterruptNode and a ProcessAbortNode. Level is stored in every parse tree node.

9.3.2 TailRecursion

When a method call is known to be tail recursive, alternative code can be generated. The alternative code can be prepared to avoid unbounded stack growth. To determine if a process method call is tail recursive the Boolean, tailRecursion is used. A process method call is tail recursive when the following conditions are all met.

- The method that is performing the method call has no return parameters.
- The called method itself has no return parameters.
- The method call is the last statement of the method that calls it.
- This last statement may not be the last statement of:
  - S1 of an abort statement (S1 abort S2;).
  - S1 or S2 of an interrupt statement (S1 interrupt S2;).
  - S of a while statement (while E do S;)

At the beginning of the decoration of a process method, tailRecursion is initially true. During decoration, it is set to false to satisfy the conditions mentioned above. This is done by the following nodes ProcessMethodNode, ProcessMethodCallNode, ProcessStatementsNode, ProcessAbortNode, ProcessInterruptNode and ProcessWhileNode. In the node, ProcessMethodCallNode the value of tailRecursion is stored so it can be used for generating the alternative code.

9.3.3 DataClasses

The POOSL-simulator must be able to determine the necessary data classes belonging to a (complete) POOSL specification. This is needed for writing a (complete) POOSL specification to a file.

All names of data classes used in expressions, data methods or process methods are stored in a Set of Symbols, called dataClasses. During decoration, DeclarationNodes, created by declarations of variables in a data or process method definition, will add their corresponding data class name (type) to dataClasses. Occurrences of the parse tree node DataNewNode, created as result of the expression new(DataClassName), also add the name of the used data class to dataClasses.
In each `DataExpressionNode`, `DataMethodNode` and `ProcessMethodNode`, `dataClasses` is stored so the POOSL-simulator is able to determine the used data classes of a specification.

### 9.3.4 Channels

The POOSL-simulator needs to determine the channels, which are used within a process class. This is necessary to automatically generate the notation of the communication channels while writing a POOSL process class specification to file.

All channel names that are used within a process method are added to `channels`, which is an `OrderedCollection` of `Symbols`. Each communication statement node, i.e. `ProcessSendNode`, `ProcessReceiveNode` and `ProcessBroadcastNode`, adds its channel name to `channels`. In each `ProcessMethodNode`, `channels` is stored so the POOSL-simulator is able to determine the used channels of a process class.

### 9.3.5 MessageInterface

The POOSL-simulator needs to determine the message interface of a process class. This is necessary to automatically generate the notation of the message interface while writing a POOSL process class specification to file.

All communication messages that are used within a process method are added to `messageInterface`, which is an `OrderedCollection` of `Strings`. Each communication statement node (`ProcessSendNode`, `ProcessReceiveNode` and `ProcessBroadcastNode`) adds a `String` that represents the communication message. `MessageInterface` is stored in each `ProcessMethodNode`.

In the abstract syntax of POOSL, only the number of parameters is stated in the message interface. In the concrete syntax of POOSL the type (data class name) of each parameter is stated. The notation of the type of communication message parameters in the concrete syntax is only for clarification. However, up until now only `ProcessReceiveNode` knows what the types of the parameters are, so only in these cases the appropriate type is added. In case of a `ProcessSendNode` or `ProcessBroadcastNode`, the type ‘UNKNOWN’ will be added for the parameters. Notice however that the matching of communication actions depends on the number of parameters and not their type, according to the semantics of POOSL.

### 9.3.6 Variable Dictionaries

The decoration `parameter` contains several dictionaries, which are used to determine the scope and type of variables that are used within expressions. The type of a variable is the
name of the data class to which it is declared. The scope of the variable indicates in which region of the POOSL code the variable is valid and may be referred to. The type and scope of variables is needed for type casting and scope identification of variables for C++ code generation.

The decoration parameter contains a variable dictionary for each scope, namely instParmDict, instVarDict, inputParmDict, localVarDict, returnParmDict and clusterParmDict. These dictionaries are instances of Dictionary with variable names (Symbols) as keys and their types (Symbols) as associated values.

The declaration of variables in POOSL classes and instance methods are stored in DeclarationNodes. During decoration, these nodes add the declared variable names and the accompanying types to the dictionary with the appropriate scope. The dictionaries are subsequently used during decoration to assign the type and scope to a DataVariableNode (created by variables used in expressions). Therefore, each DataVariableNode is aware of its own type and scope.

9.3.7 VarRef

C++ member function calls may only pass a reference to the appropriate array that contains the variables if it is used within that member function. In Appendix D is indicated by the functions INPUTPARAM(x), RETPARAM(x) and LOCALVAR(x), when and where these references have to be passed.

To accomplish these functions the instance variable varRef is used. VarRef is an instance of VariableReference that consists of three Booleans, refToInputVar, refToReturnVar and refToLocalVar. These booleans indicate whether a process statement or data expression refers to (uses) a variable of a specific scope (input, return or local variable).

During decoration, an occurrence of a DataVariableNode with one of the three scopes sets the appropriate boolean to true. When a statement is composed of multiple statements or expressions, the three booleans of the statement are subjected to the logic or operator with each boolean of these composition statements or expressions. In every process statement node and the in nodes DataExpNode and DataFactorNode, a varRef is stored.

9.3.8 Blocks

The C++ implementation uses threads (lightweight processes, see [Tet97]) to implement some POOSL constructs. The translated POOSL constructs that are implemented in these threads are specified by C++ member functions. In addition, the condition expressions of conditional receive statements and guard statements are specified by member functions. These member functions have to be generated further in the source file and must be prototyped in the header file.
To accomplish this the instance variable Blocks of the decoration parameter is used. Blocks is an OrderedCollection, which contains references to parse tree nodes. During decoration, these references are added by nodes that need to generate such member functions, i.e. ProcessSelectNode, ProcessAbortNode, ProcessInterruptNode, ProcessGuardedCmdNode and ProcessReceiveNode. The references refer to the children of these nodes, so these children can generate code within the member functions. Blocks is stored in each ProcessMethodNode.

During C++ code generation of this node, the member function names (see Section 9.3.9) are stated in the code as function parameters (see Appendix D). Therefore, the implementation is able to call these member functions (and let them be performed by threads). When the statements of the ProcessMethodNode have finished their code generation, the member function code can be generated. The ProcessMethodNode generates the code in the source file for the member functions by sending the message emitCPPBlock: parameter to each element of blocks. This message also generates the accompanying prototyping in the header file.

9.3.9 BlockNr

The C++ member functions, which have to be generated as stated in Section 9.3.8 have to have unique names to distinguish them.

For creating these unique member function names blockNr, an instance of String, is used. BlockNr contains a String that consists of the nested numbering of the statements, separated by underscores ('_1_2'). During decoration, each statement node that has to generate member functions will append the appropriate number to blockNr and pass it to its children. BlockNr is stored in each process statement node, DataExpNode and DataFactorNode.

During C++ code generation, the contents of blockNr are appended to the POOSL method names (concatenated with '_Block_' or '_Cond_') to create member function names, so they all have unique names.

9.3.10 ParentNode

During decoration, a (parent) node passes a reference to itself to its children, using the instance variable parentNode of the decoration parameter. Every parse tree node stores this reference in its instance variable parent. This parent is used for searching somewhere else in the tree to find:

- The communication channels of a class.
- The node with a specific number.

The communication channels of the instances within a behaviour specification are needed to compute the interconnection structure (see Section 9.3.13). To retrieve these
channels the message `getChannelsOfClass: className` is sent by a `BehaviourSpecInstNode`. A `BehaviourSpecInstNode` is the parse tree node that contains the information of the instances within a behaviour specification. The method climbs up in the tree to `ClassDefinitionsNode` using `parent` and then searches for the needed class. When the class is found the method returns an `OrderedCollection` of channel names (`Symbols`), as they where stated in communication channels of a class definition.

*Parent* is also used to find a node with a specific number. How this is accomplished and what it is used for is explained in the next section.

### 9.3.11 NodeNumber

`NodeNumber` is an `Integer` that is stored in every statement node. It is used to create a link between the generated Smalltalk code (a `String`) and the decorated parse tree. With this link, it is possible to find the part of the parse tree belonging to accompanying the Smalltalk code.

During decoration, `nodeNumber` is incremented by each statement node, except for a `ProcessStatementsNode`. Each statement node is assigned a `nodeNumber` in post order, so the higher the node the higher the number. In this manner, the parse tree can be seen as a heap structure with respect to these `nodeNumbers`. In a heap structure, all nodes satisfy the heap condition, which indicates that the number of the parent is greater or equal to the number that its children contain. With this condition, it is possible to search very fast for a node with a specific number. In Figure 9-2, this heap tree structure is visualised based on the following POOSL code. The class name of each node and its value of `nodeNumber` are noted Figure 9-2.

```smalltalk
methOO
| a : Integer |
a := 1;
(ch1 ! msg1; loopOO)
interrupt
  sel
    ch2 ? msg2; a := 2;
or
    ch3 ? msg3; a := 3;
  les.
```

9 Parse Tree Decoration 43
In the Smalltalk code, nodeNumber is generated as a parameter of the message checkPoint: nodeNumber. This message is generated in front of each process statement and data statement (the latter dependent on debugFlag, see Section 10.3).

This additional code is used by the POOSL-simulator to trace the execution within the POOSL code. To create the link to the parse tree the POOSL-simulator sends the message getNodeWithNr: nodeNumber to a ProcessMethodNode, a DataMethodNode or a statement node. This message climbs up in the parse tree, using parent, to the ProcessMethodNode or DataMethodNode. Then it goes down the tree searching (in pre-order) for the node with the nodeNumber according to the heap condition. When the

---

**Figure 9-2: Searching a Node with a Specific Number in a Heap Tree Structure.**

In the Smalltalk code, nodeNumber is generated as a parameter of the message checkPoint: nodeNumber. This message is generated in front of each process statement and data statement (the latter dependent on debugFlag, see Section 10.3).

This additional code is used by the POOSL-simulator to trace the execution within the POOSL code. To create the link to the parse tree the POOSL-simulator sends the message getNodeWithNr: nodeNumber to a ProcessMethodNode, a DataMethodNode or a statement node. This message climbs up in the parse tree, using parent, to the ProcessMethodNode or DataMethodNode. Then it goes down the tree searching (in pre-order) for the node with the nodeNumber according to the heap condition. When the
node is found the method returns the reference to the node. This procedure is also visualised with the arrows in Figure 9-2.

With this reference, the POOSL-simulator is able to determine the position in POOSL code of the corresponding node, using the instance variable \textit{interval} of the node (see Section 7.5).

\section*{9.3.12 InstanceCollection}

The instances used within a behaviour specification are needed for C++ code generation for declaration and creation of process objects. The instantiation expression of each instance is needed to initialise the process objects. The instances of the same class must have a unique identification to distinguish them.

\textit{InstanceCollection} is an \textit{OrderedCollection} of \textit{ClassInstantiations}. A \textit{ClassInstantiation} consists of a class name (\texttt{Symbol}), the instantiation expressions and the number of the instance. The instantiation expressions are stored in an \textit{OrderedCollection} of \textit{DataExpNodes} and/or \textit{DataFactorNodes}. They are used to generate C++ code to initialise the process objects. The number of the instance is an \texttt{Integer}, which indicates the numbering of instances of the same class within the behaviour specification.

During decoration of a behaviour specification, a \textit{ClassInstantiation} is created and added to \textit{instanceCollection} by a \texttt{BehaviourSpecInstNode}. A \texttt{BehaviourSpecInstNode} is the parse tree node that contains the information of the instances within a behaviour specification. \textit{InstanceCollection} is stored in the \texttt{SystemSpecNode} and each \texttt{ClusterClassNode}.

\section*{9.3.13 ConnectionDict and HiddenConnectionDict}

The interconnection structure of a system or a cluster must be computed of their behaviour specification. The channels entries of the instances, channel hiding and channel relabeling within behaviour specifications determine the interconnection structure of the instances. The interconnection structure is needed to generate C++ code for declaration, creation and interconnection of C++ channel and connector objects.

To determine the interconnection structure of a behaviour specification, the instance variables \textit{connectionDict} and \textit{hiddenConnectionDict} of the decoration parameter are used. In general, \textit{connectionDict} contain the connections within a behaviour specification that are not hidden, i.e. observable. \textit{HiddenConnectionDict} contain the hidden connections.

These \textit{connectionDict} and \textit{hiddenConnectionDict} are instances of the class \texttt{ConnectionDictionary} and respectively \texttt{HiddenConnectionDictionary}. These classes are both subclasses of \texttt{Dictionary}. A \texttt{ConnectionDictionary} has channel names (\texttt{Symbols}) as
keys and *OrderedCollections* of *Connectors* as their associated values. A *HiddenConnectionDictionary* has channel names (*Symbols*) as keys and *OrderedCollections* of *OrderedCollections* of *Connectors* as their associated values. These *OrderedCollections* in the *OrderedCollection* of an association value of a *HiddenConnectionDictionary* is used to distinguish channels within a behaviour specification with same names, which are not connected to each other.

To hold the channel entry belonging to an instance, the class *Connector* is used. An instance of *Connector* consists of a *ClassInstantiation* and a channel entry name (*Symbol*). *ClassInstantiation* is used for unique identification of the instances (see Section 9.3.12). The channel name is a name of the channel entry that is possessed the instance, i.e. stated in communication channels of its class definition. These channels are retrieved with the message *getChannelsOfClass: className* (see Section 9.3.10).

During decoration, a *BehaviourSpecInstNode* creates a *ConnectionDictionary* that contains all the channels of the instance as keys. The associated values contain an *OrderedCollection* of *Connectors* of these channels. A *BehaviourSpecInstNode* also creates an empty *HiddenConnectionDictionary*. These dictionaries are subjected to the manipulations of *RelabelListNodes* and *HidingListNodes*.

A *BehaviourSpecNode* also creates a new *ConnectionDictionary* and a *HiddenConnectionDictionary*, which contain the composition of the *ConnectionDictionaries* and respectively the *HiddenConnectionDictionaries* of its children. The composition of the *ConnectionDictionaries* implies that the dictionaries are combined in the new dictionary in the following manner. When a channel name does not exist as key in the new dictionary, the key and the associated value are stored in the new dictionary. Otherwise, if a key already exists in the new dictionary, the associated *Connectors* are added to the present associated *OrderedCollection* of *Connectors*. The composition of the *HiddenConnectionDictionaries* implies that the dictionaries are combined in the new dictionary in the following manner. When a channel name does not exist as key in the new dictionary, the key and the associated value are stored in the new dictionary. Otherwise, if a key already exists in the new dictionary, the associated *OrderedCollection* of *Connectors* are added to the present associated *OrderedCollection* of *Connectors*. These dictionaries are also subjected to the manipulations of *RelabelListNodes* and *HidingListNodes*.

A *RelabelListNode* contains an *OrderedCollection* of channel names, which are alternately the new name, and the old name of a channel. If the old channel name is a key in the *ConnectionDictionary* that is passed by the parent of *RelabelListNode*, the key and the associated value are removed. In a temporal *ConnectionDictionary*, the associated value is put at the newly created key with the new channel name. After all relabels, the temporal *ConnectionDictionary* is added to the dictionary that was passed by the parent. This is done to make sure that all the relabels, which are stated between one pair of square brackets, are performed as an atomic action, i.e. they are all performed simultaneously.

A *HidingListNode* contains an *OrderedCollection* of channel names. If one of these channel names is contained as key in the *ConnectionDictionary*, the *HidingListNode*
moves the key and associated value to the $HiddenConnectionDictionary$. The associated value, an $OrderedCollection$ of $Connectors$, is put in the $OrderedCollection$ of the association value of the $HiddenConnectionDictionary$. However, if the $OrderedCollection$ of $Connectors$ that is associated with the key of the $ConnectionDictionary$ only contains one $Connector$, it is not put in the $HiddenConnectionDictionary$. Because when it contains just one $Connector$ it implies that its channel is not connected to any other instance and therefore it is not required to determine the interconnection structure.

The $ConnectionDictionary$ and $HiddenConnectionDictionary$ of the root of the behaviour specification tree are stored in each $ClusterClassNode$. The $SystemSpecNode$ only stores the $HiddenConnectionDictionary$, because no channel is observable from outside the system. Therefore, the $SystemSpecNode$ first adds the $ConnectionDictionary$ to the $HiddenConnectionDictionary$, and then stores the latter.

To clarify the procedure of computation of the interconnection structure of a behaviour specification an example of an instance structure diagram is showed in Figure 9-3. The parse tree of the accompanying behaviour specification is visualised in Figure 9-4.

![Diag1](image.png)

**Figure 9-3**: Instance Structure Diagram of: $(A || (B[a/b] || C[e/d])[f/e])\{a\}$. 

---

*Figure 9-3: Instance Structure Diagram of: $(A || (B[a/b] || C[e/d])[f/e])\{a\}$.*
Figure 9-4: Parse Tree of Behaviour Specification: (A || (B[a/b] || C[e/d])f[e]{a}).

The contents of the connectionDict and hiddenConnectionDict for each decoration step, indicated by numbers, is stated below. The a after the numbers indicate the contents of the dictionaries before relabeling and hiding, b means after relabeling and hiding. For the representation of connectionDict and hiddenConnectionDict, the following notation is used.

key ➔ value = Dictionary
[instance, channel ] = Connector
{} = OrderedCollection
The contents of the `connectionDict` and `hiddenConnectionDict` during decoration:

1a/b  
`connectionDictA`: \[ a \rightarrow \{ [A, a]\} \]
`hiddenConnectionDictA`: \[ \emptyset \]

2a  
`connectionDictB`: \[ b \rightarrow \{ [B, b]\}, d \rightarrow \{ [B, d]\} \]
`hiddenConnectionDictB`: \[ \emptyset \]

2b  
`connectionDictB`: \[ a \rightarrow \{ [B, b]\}, d \rightarrow \{ [B, d]\} \]
`hiddenConnectionDictB`: \[ \emptyset \]

(relabeling)

3a/b  
`connectionDictC`: \[ a \rightarrow \{ [C, a]\}, d \rightarrow \{ [C, d]\} \]
`hiddenConnectionDictC`: \[ \emptyset \]

4a  
`connectionDictBC`: \[ a \rightarrow \{ [B, b] [C, a]\}, d \rightarrow \{ [B, d] [C, d]\} \]
`hiddenConnectionDictBC`: \[ \emptyset \]

(composition)

4b  
`connectionDictBC`: \[ a \rightarrow \{ [B, b] [C, a]\}, e \rightarrow \{ [B, d] [C, d]\} \]
`hiddenConnectionDictBC`: \[ \emptyset \]

(relabeling)

5a  
`connectionDictABC`: \[ a \rightarrow \{ [A, a] [B, b] [C, a]\}, e \rightarrow \{ [B, d] [C, d]\} \]
`hiddenConnectionDictABC`: \[ \emptyset \]

(composition)

5b  
`connectionDictABC`: \[ f \rightarrow \{ [B, d] [C, d]\} \]
`hiddenConnectionDictABC`: \[ a \rightarrow \{ [A, a] [B, b] [C, a]\} \]

(relabeling)

(hiding)
10 Code Generation

10.1 Introduction

All POOSL specification constructs must be translated in a corresponding target specification that is semantically equivalent. The translation tables of both Smalltalk and C++ (Appendix C and D) indicate the target code that has to be generated for a corresponding POOSL construct. The mapping of POOSL on Smalltalk is designed and formulated by Mark Geilen. The mapping for translating POOSL to C++ is designed and formulated in [Tet97]. Translations to both Smalltalk and C++ can be seen as a translation to an intermediate code, because in both cases supporting target code is required to implement the semantics of POOSL. In Smalltalk, this support is implemented within the POOSL-simulator and for C++, it is implemented by the C++ library of [Tet97].

10.2 Tree Traversal

Each parse tree node itself is aware of what target code it has to generate. Therefore, it is possible to request a node of the decorated parse tree to generate its code. Then the node will generate the first part of the code it has to generate and after that it subsequently requests its children to generate code. After its children are finished, the node will generate the last part of its code. However, for C++ code generation there is an exception to this for parts of code that have to be generated elsewhere. In these cases, references to the parts of the parse tree that still have to generate code will be stored. These parts are requested to generate code later (see Section 10.3).

While traversing the tree a parameter is passed along with the code generation message emitST: parameter and emitCPP: parameter for Smalltalk and C++ code generation respectively. This parameter is an instance of a subclass of EmitParm (see Appendix E.2). It contains information needed somewhere else in the tree or compiler directives/options. These compiler directives instruct the compiler to generate some optional code or perform optimisations.

10.3 Smalltalk Code Generation

Smalltalk code generation is very straightforward, i.e. the code can be generated the same order as in the POOSL specification. This is (partly) due to the fact that Smalltalk
code only has to be generated for methods and expressions, because the classes are stored within the POOSL-simulator.

All nodes that can be part of the parse tree of a process method or a data method respond to the message `emitST: parameter`. This message returns an instance of `String` that contains the Smalltalk translation of the POOSL specification. Formatting of the Smalltalk code in a readable way during code generation is not necessary because the Smalltalk compiler has a method built-in that is able to perform this (format: string).

The parameter, an instance of `EmitSTParm`, only consists of the compiler directive `debugFlag`, an instance of `Boolean`. This `debugFlag` instructs the POOSL-compiler whether to generate the messages `checkPoint: nodeNumber` and `statementRequest` in front of each data statement (see Appendix C). For what reason this is used is explained later in this section.

As stated in section 9.3.2 a `ProcessMethodCallNode` contains an instance variable `tailRecursive` that indicates whether a method call is tail recursive. Therefore, the POOSL-compiler is able to generate alternative Smalltalk code for a method call that is tail recursive and one that is not (see $P_{2ST_1}(\text{statement}^p)$ in Appendix C). The alternative code can be prepared to avoid unbounded stack growth.

To let the POOSL-simulator be able to read POOSL specifications that are stored in a file, several nodes have to be accessible for the simulator. The nodes `SystemSpecNode`, `ClusterClassNode`, `ProcessClassNode`, `DataClassNode`, `ProcessMethodNode`, `DataMethodNode`, `ProcessMethodCallNode` and `DataExpressionNode` respond to several messages sent by the POOSL-simulator for installing classes and methods in its class browser. These messages deliver the POOSL-simulator the required information, i.e. the class name, the instantiation parameters, the instance variables, the initial method call, et cetera. The POOSL-simulator is also able to retrieve the used data classes, used communication channels and the used message interface, which are determined during decoration as explained in section 9.3.3, 9.3.4 and 9.3.5.

After decoration, each `DataVariableNode` knows its scope (see section 9.3.6). Therefore, the POOSL-compiler is able to generate Smalltalk code indicating the scope of the variable (see $P_{2ST_1}(E)$ in Appendix C). For the POOSL-simulator, it is then easier to find the variable because it needs less searching.

As stated in section 9.3.11 the Smalltalk code `checkPoint: nodeNumber` is generated in front of each process statement. In addition, depending on `debugFlag`, it can be generated in front of each data statement. This additional code is used to create a link between the Smalltalk code and the corresponding part of the parse tree. It is used by the POOSL-simulator to trace the execution within the POOSL code.

In SHE a behaviour-preserving transformation exists that removes the boundaries of a cluster. The instantiation expressions of a removed cluster have to be substituted in the corresponding instantiation parameters. These parameters can occur in instantiation expressions of the instances within the behaviour specification of a cluster.
To facilitate this transformation for the POOSL-simulator the parse tree nodes for expression have additional behaviour. The part of the parse tree that contains the instantiation expression of the removed cluster is substituted in the instantiation expression of the instances within the cluster. This substitution takes place at the positions of DataVariableNodes of the corresponding instantiation parameters. In Figure 10-1, this procedure is visualised for the instantiation parameter $ip$ by the dotted line.

The POOSL-simulator can send the message withSubstitutions: paramDictionary to a DataExpNode or a DataFactorNode (root of parse tree of an expression). The parameter paramDictionary contains a Dictionary with instantiation parameter names as keys and references to the parse tree of their expression as associated values. The message will use the message substituteVar: varName with: expression for substitution of each instantiation parameter. Therefore, the result of withSubstitutions: paramDictionary will be the parse tree of the instantiation expression with the parse trees of instantiation parameters substituted.

Figure 10-1: Substitution of instantiation parameter $ip (= 1)$ in expression $ip + 2$. 
All expression nodes respond to the message emitPOOSL, which generate a String that contains POOSL code. Therefore, the substituted tree can generate POOSL code for substituted instantiation parameters in the instantiation expressions. Therefore, the POOSL-simulator is able to automatically generate the POOSL code for these instantiation expressions.

10.4 C++ Code Generation

All nodes that have to generate C++ code respond to the message emitST: parameter, which writes to a header file and source file. For each POOSL class specification and for the system’s behaviour specification, a header file and a source file is created. The parameter, an instance of EmitCPPParm, consists of a headerStream and a sourceStream, which are used for writing to the header file and the source file. These streams are instances of FormattedExternalWriteStream, which connect a stream to a file. The stream has some methods for formatting the generated code in a readable way.

The parameter also contains the instance variables path, headerExt, sourceExt and systemName, which are assigned during user interaction with the C++ code generation menu. Path, headerExt and sourceExt are Strings that contain respectively the path where the files have to be written, the extension of the header files and the extension of the source files. SystemName is a String containing the name of the complete POOSL specification needed for inclusion in all header files. In addition, parameter contains currentClass and currentMethod. CurrentClass is a String that contains the class name that is currently generating code. This is used by all nodes that generate member functions because they need to know their associated class name for generating code (see Appendix D). CurrentMethod contains a reference to the method node that is currently generating code. This is used for inclusion of the method name in the member functions names that will be generated within the method. It is also possible to store path, headerExt, sourceExt, SystemName, currentClass and currentMethod during decoration in the nodes that required them. However, because many nodes need this information, which is identical for many nodes, this information is passed during code generation.

Further parameter contains a compiler directive varDebug, an instance of Boolean. This varDebug instructs the POOSL-compiler whether to generate index number for the array that contains the variables or generate the variable names (see the boxed text and note in Appendix D). This compiler optimisation prevent searching for variable names during execution and therefore makes the C++ implementation (slightly) faster. However when printing debug information while running the C++ implementation no variable names will be visible.

As stated in section 9.3.2 a ProcessMethodCallNode contains an instance variable tailRecursive that indicates whether a method call is tail recursive. Therefore, the POOSL-compiler is able to generate alternative C++ code for a method call that is tail recursive and one that is not (see P2CPP(statement\_{P_{s_1,...,s_j}}) in Appendix D). The alternative code can be prepared to avoid unbounded stack growth.
Each `DataVariableNode` knows its scope and type (see section 9.3.6). Therefore, the POOSL-compiler is able to generate C++ code indicating the scope of the variable by referring to the accompanying C++ array that contains the variable. In addition, the type is required for type casting, i.e. indication the type that the variable contains (see \( \text{P2CPP(E)} \) and \( \text{P2CPP(statement)}^{5} \) in Appendix D).

Only when variables of a specific scope are used within C++ member functions, a reference to the accompanying C++ array that contains the variables of the scope must be passed in the member function call. When and where these references have to be passed is indicated by `INPUTPARAM(x)`, `RETPARAM(x)` and `LOCALVAR(x)` in Appendix D. To accomplish this every process statement node and the nodes `DataExpNode` and `DataFactorNode` has an instance variable `varRef` that indicates whether variables of a specific scope occurs in the underlying tree (see Section 9.3.7). The content of varRef determines whether the name of the C++ array is generated as a parameter of the member function call.

Some translated POOSL constructs are implemented in threads, which are specified by C++ member functions. These member functions have to be generated further in the source file. In addition, each member function must be prototyped in the header file. To accomplish this references to nodes that have to generate code in such member functions are stored in the instance variable `Blocks` of each `ProcessMethodNode` (see Section 9.3.8). During C++ code generation of this node, the member function names (see Section 9.3.9) are stated in the code as function parameters (see \( \text{P2CPP(statement)}^{p} \) Appendix D). Therefore, the implementation is able to call these member functions and let them be performed by threads. When the statements of the `ProcessMethodNode` have finished their code generation, the member function code can be generated. The `ProcessMethodNode` generates the code in the source file for the member functions by sending the message `emitCPPBlock: parameter` to each element of `blocks`. This message also generates the accompanying prototyping in the header file.

For declaration, creation and initialisation of the system, clusters and processes the `InstanceCollection` is used. It is stored in `SystemSpecNode` and each `ClusterClassNode` (see Section 9.3.12). `InstanceCollection` is an `OrderedCollection` of `ClassInstantiations`. A `ClassInstantiation` contains the name of the class and the number of the instance, which are combined to create unique C++ identifiers for the instances. The declaration and creation of the instances is indicated by `PROTOTYPE(BSpec)` and `CONSTRUCT(BSpec)` in Appendix D. A `ClassInstantiation` also contains the instantiation expressions of the instance, which is used for initialisation. This initialisation code is indicated by `STARTUP(BSpec)` in Appendix D.

The processes and clusters have to be interconnected with each other according to the behaviour specification of which they are part. Therefore, each `ClusterClassNode` contains a `ConnectionDictionary` and `HiddenConnectionDictionary`, which contain respectively the observable connections and the hidden connections of its behaviour specification. The `SystemSpecNode` only contains a `HiddenConnectionDictionary`, because no channel is observable from outside the system. In these dictionaries, a channel name is associated with the collection channel entries of the instances that are connected to the channel. These dictionaries are used for C++ code generation of the
interconnection structure of behaviour specification of the system and clusters. In Appendix D the C++ code generation for these interconnections is indicated by CONNECT(BSpec).

All identifiers used in the POOSL specification are concatenated with underscores to create the corresponding C++ identifiers. This is done to avoid conflicts with C++ keywords. All instances of processes and clusters and all channels defined in POOSL have besides the underscore also a number concatenated to their class name for the corresponding C++ identifier. This is for unique identification to distinguish process and cluster instances of the same class and to distinguish channels with the same name that are not connected to each other.

To overcome problems with respect to the limitation of the filename length in DOS (eight characters) all generated header and source files have numbers as filenames. However, the POOSL-compiler generates a POOSL specification dependent UNIX makefile wherein these files and their original POOSL class name are listed. The common UNIX makefile, which is used for all POOSL specifications, is able to rename the files to their original POOSL class name.
11 Conclusions, Results and Future Work

11.1 Conclusions

The syntax and grammar of POOSL formulated in [Kup96], the mapping on Smalltalk formulated by Mark Geilen (implementing the POOSL-simulator) and the mapping on C++ formulated in [Tet97] are used as a starting-point for the implementation of the POOSL-compiler.

The concrete syntax and the accompanying grammar specification of POOSL of [Kup96] have been modified to correspond more to the abstract syntax and to remove some impractical syntax constructs. To reflect these modifications, new POOSL syntax diagrams are drawn up (see Appendix A) and the accompanying grammar is formulated (see Appendix B).

The POOSL-compiler has been implemented within the Smalltalk environment because of the required interaction with the POOSL-simulator. Changes in the grammar of POOSL are still very likely because POOSL is still under construction. Therefore, the choice was made to use the compiler construction tool T-gen (Translator-Generator, see Chapter 7). A compiler construction tool makes the compiler highly maintainable because changes in the grammar can automatically be adapted by the compiler.

The compiler construction tool (T-gen) is able to construct a parse tree that contains all required information that was present in the POOSL specification. However, some information in the parse tree will not be optimally accessible in the way it is useful for the translation to a specific target code. Therefore, the parse tree has to be decorated, i.e. traversed to add and rearrange information so it is accessible where it is needed. After decorating the parse tree, the tree can be traversed to generate the target code.

11.2 Results

A POOSL-compiler has been developed for translating POOSL to Smalltalk as well as to C++. The POOSL-compiler covers the complete translation of all POOSL specifications to C++. To Smalltalk, only the parts of a POOSL specification, which are required by the POOSL-simulator, can be translated. For both translations, supporting target code is required to implement the semantics of POOSL. In Smalltalk, this support has implemented within the POOSL-simulator by Mark Geilen and for C++, it has been implemented by [Tet97] with a C++ library. The POOSL-compiler is not (yet) able to check the context conditions of POOSL.
Several test cases have been performed to test the functionality of the POOSL-compiler. With these tests, the translation to Smalltalk and C++ with their support implementations could also be tested thoroughly. Among several other test cases, the POOSL specifications of the PAR-protocol (see [Voe95a]) and the elevator system (see [PV97]) were compiled and executed successfully.

### 11.3 Future Work

**Extending/Changing POOSL**
Due to the use of the compiler construction tool (T-gen), future extensions or changes of POOSL will be very easy to realise. After modifying the grammar specification a new scanner and parser can be generated automatically.

**Context Conditions Checking**
Checking the context conditions of POOSL can be performed by traversing the decorated parse tree and determine whether the conditions are met. Some information needed for context checking will already be available. However, it will be necessary to extend the parse tree decoration to have all required information at the places it is needed.

Due to the introduction of an early return statement one context condition has been added to the context conditions of POOSL, formulated in [PV97]. This context condition implies that each data method has to terminate with a return statement.

**Translations to other Target Codes**
The decorated parse tree will be very useful for future translations to other target codes, because a lot of information, which is needed for generating target code will already be available at the required places. Therefore, translation to other target code will need just little (or no) extra decoration.

**Compiler Optimisations**
The alternatives of a select statement don not necessarily have to be performed by threads. The POOSL-compiler must determine all (nested) statements that want to perform the first execution step. The POOSL-compiler is then able to generate code for submitting the appropriate requests simultaneously without using threads.
Acknowledgements

Finally, I want to thank all people that supported me during this educational graduation project. There are some people that I want to thank explicitly.

First, I want to thank the supervisor of the project, M.P.J. Stevens. He advised me and gave me the opportunity to accomplish this compelling graduation project.

Jeroen Voeten and Piet van der Putten I want to thank them for explaining SHE and POOSL. I also thank them for their support, reading earlier drafts of this thesis and their enthusiasm when the POOSL-compiler started to work.

For the perfect co-operation throughout the project, I thank Mark Geilen. I also thank him for explaining POOSL details, testing the POOSL-compiler and reading earlier drafts of this thesis.

I thank Niels van Tetrode for explanation of the C++ mapping and co-operation during implementation and testing of POOSL-compiler for C++.

At last, I thank Harald Vranken for supporting the modification of the POOSL specification of the elevator system for testing the POOSL-compiler.
References

[ASU86] Aho, A.C. and R. Seth, J.D. Ullman
Addison-Wesley, 1986.

[Fel96] Felius van, J.D.
From POOSL to C++.  
Eindhoven: Eindhoven University of Technology, Faculty of Electrical  
Engineering, 1996.  
Master’s Thesis.

[FL88] Fischer, C.N. and R.J. LeBlanc
Crafting a Compiler.  

Gainesville, Florida: University of Florida, Computer and Information  
Author’s E-mail Address: graver@ufl.edu.

[Joh75] Johnson, S.C.
YACC – yet another compiler compiler.  

[Kup96] Kuppens, P.W.
Design of a Verification Tool for Complex System Design.  
Eindhoven: Eindhoven University of Technology, Institution for  
Continuing Education, 1996.  
Graduation Report.

[Les75] Lesk, M.E.
LEX – a lexical analyser generator.  

[Mil80] Milner, R.
A Calculus of Communicating Systems.  
Lecture Notes in Computer Science 92.  
Berlin: Springer-Verlag, 1980.
[Man96] Mankoe, I.  
*Designing a Simulator for POOSL.*  
Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering, 1996.  
Master's Thesis.  

[PPS92] ParcPlace Systems.  
/Objectworks\Smalltalk Release 4.1.  
User's Guide.  

*Object-Oriented Co-Design for Hardware/Software Systems.*  

[VPS96] Voeten J.P.M. and P.H.A. van der Putten, M.P.J. Stevens  
*Behaviour-Preserving Transformations in SHE, A Formal Approach to Architecture Design.*  

[PV97] Putten van der, P.H.A. and J.P.M. Voeten.  
*Specification of Reactive Hardware/Software Systems.*  

[Str92] Stroustrup, B.  
*The C++ Programming Language.*  

[Tet97] Tetrode van, J.N.  
*Implementing POOSL in C++.*  
Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering, August 1997.  
Master's Thesis.  

[Voe95a] Voeten, J.P.M.  
*POOSL: An Object-Oriented Specification Language for the Analysis and Design of Hardware/Software Systems.*  
EUT report 95-E-290.  

---

References
Voeten, J.P.M.


Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering, October 1995.


EUT report 95-E-293.
Appendices

A  POOSL Syntax Diagrams .................................................67
   A.1 POOSL Process Part ..............................................67
   A.2 POOSL Data Part ...............................................74

B  POOSL Grammar ..........................................................81
   B.1 POOSL Token Class Specification ..............................81
   B.2 POOSL Grammar Specification ..................................81

C  Translation Table for Smalltalk .....................................87

D  Translation Table for C++ .............................................93

E  Smalltalk Classes .....................................................113
   E.1 Parse Tree Node Classes .......................................113
   E.2 Other Classes ..................................................115
A POOSL Syntax Diagrams

A.1 POOSL Process Part

POOSL Specification:

The dotted lined box indicate some additional acceptable POOSL specifications, which are only to be used by the POOSL-simulator. The prefix for each part is to avoid some ambiguousness.

BehaviourSpecification:

BehaviourSpecificationTerm:

BehaviourSpecificationTerm:
Channel Manipulations:

- Relabeling
- Hiding

Relabeling:

```
[ ]  \--/  ]
  Ident  \   Ident
    ,
```

Hiding:

```
{ IdentList }  
```

A POOSL Syntax Diagrams
ProcessClass:

- process ➔ class ➔ Ident ➔ Parameters ➔
- instance ➔ variables ➔ DeclarationList ➔
- communication ➔ channels ➔ IdentList ➔
- message ➔ interface ➔ MessageList ➔
- initial ➔ method ➔ call ➔ Ident ➔
  ➔ DataExpressionList ➔)
- instance ➔ methods ➔ ProcessMethodList ➔

ProcessMethodList:

- ProcessMethod ➔

ProcessMethod:

- Ident ➔ Parameters ➔ Parameters ➔
- OptionalVariables ➔
- ProcessStatementList ➔.
DeclarationList:

Declarations:

Parameters:

OptionalParameters:

MessageList:

MessageProto:

A POOSL Syntax Diagrams
ProcessStatementList:

```
ProcessStatement
;  

;  
```

ProcessStatement:

```
ProcessTerm

ProcessTerm abort ProcessTerm

ProcessTerm interrupt ProcessTerm
```
ProcessTerm:

```
[] DataExpression [] ProcessTerm
while DataExpression do ProcessStatementList od
if DataExpression then ProcessStatementList fi
  else ProcessStatementList
sel ProcessStatementList or ProcessStatementList les
delay DataExpression
skip
Ident = DataExpressionList = IdentList
Ident ? Ident
  IdentList
Ident !* Ident
  IdentList
Ident := DataExpression
Ident DataMethodCallList
(` ProcessStatementList `)
```

A POOSL Syntax Diagrams 73
A.2 POOSL Data Part

DataClass:

- data
- class
- Ident
- instance
- variables
- DeclarationList
- instance
- methods
- DataMethodList

DataMethodList:

- DataMethod

DataMethod:

- Ident
- OptionalParameters
- Ident
- OptionalVariables
- DataStatementList
- .
- Ident
- OptionalParameters
- Ident
- primitive
- PrimitiveString
- ;
DataStatementList:

DataStatement:

while DataExpression do DataStatementList od

if DataExpression then DataStatementList fi

else DataStatementList

return DataExpression

Ident ( DataExpressionList )

Ident := DataExpression

Ident DataMethodCallList

self

( DataStatementList )
DataExpression:

DataFactor:

DataMethodCallList:

DataExpressionList:
PrimaryExpression:

- PrimitiveDataObject
  - Ident
  - self
  - new (Ident) (DataExpression)

Operator:

- [\texttt{=<>+-|*/%&}]
  - ==
  - !=
  - <=
  - >=

PrimitiveDataObject:

- PrimitiveBoolean
- PrimitiveInteger
- PrimitiveReal
- PrimitiveChar
- PrimitiveString
- nil
PrimitiveBoolean:

```
true
false
bunk
```

PrimitiveChar:

```
' 
[!-tv
\s\t]
'
cunk
```

PrimitiveReal:

```
[0-9]
.
[0-9]
runk
```

PrimitiveInteger:

```
[0-9]
%
[01]
$
[a-fA-F0-9]
iunk
```

PrimitiveString:

```
"[!-~\n\r\s\t]"
""
""
""
```
IdentList:

```
Ident
, ...
```

Ident:

```
[a-zA-Z] → [a-zA-Z0-9_]
```

OptionalVariables:

```
1 → DeclarationList → 1
```

Comment:

```
/* → [~\n\s\t] → */
```

(1) Note that here the character combination "/" and "/*" are excluded.
In this appendix the token class specification and the grammar specification used by T-­
gen within the POOSL-compiler, are stated. The text files (poosl.tok, poosl.grm) containing these specifications can be found in the directory POOSL-compiler within the SHE-directory.

### B.1 POOSL Token Class Specification

```plaintext
<ident> : [A-Za-z][A-Za-z0-9_]* ;
<int> : [0-9]+ ;
<real> : [0-9]+ . [0-9]+ ;
<char> : '[!^-~\s\t\r\n]' ;
<string> : "" ([!#\s\t\r\n] | \"\")* "" ;
<binNumber> : %[01]+ ;
<hexNumber> : \$[0-9a-fA-F]+ ;
<comment> : /\* \*\* \*\* \*\* \*\* \*\* \*\* \*\* \*\* \*\* \*\* \*\* \*\* \*\* \*\*

//\* {skipComment};

<whitespace> : [\s\t\r\n]* {ignoreDelimiter};

<relOp> : =|==|!=<|><=|>= ;

<factorOp> : \*|\%|& ;
```

### B.2 POOSL Grammar Specification

```plaintext
POOSLSpecification : BehaviourSpecification DefinitionList (SystemSpecNode)
    | DefinitionList
    | 'PROCESSMETHOD_' ProcessMethod
    | 'INITIALMETHODCALL_' Ident '(' DataExpressionList ')' '(' 'J'
    | '{initialMethodCall:parms:}'
    | 'DATAMETHOD_' DataMethod
    | 'DATAEXPRESSION_' DataExpression
    |
    BehaviourSpecification : BehaviourSpecificationList (behaviourSpecs:)
    |
    | BehaviourSpecificationList '||' BehaviourSpecificationTerm (OrderedChildren)
    |
    BehaviourSpecificationTerm : '(' 'BehaviourSpecificationTerm')'
    | ChannelManipulations (BehaviourSpecNode)
    | Ident '(' 'DataExpressionList ')' ChannelManipulations (BehaviourSpecInstNode)
    | Ident ChannelManipulations (class:chMan:)
    |
    ChannelManipulations : 'empty'
    | ChannelManipulations Relabeling (OrderedChildren)
    | ChannelManipulations Hiding (OrderedChildren)
    |
    Relabeling : '[' 'RelabelList ']' (BehaviourRelabelListNode)
```

B POOSL Grammar

81
RelabelList
  :  "empty"  {OrderedChildren}
  |  NotEmptyRelabelList
;
NotEmptyRelabelList
  :  Ident '/' Ident  {OrderedChildren}
  |  RelabelList ',' Ident '/' Ident  {liftLeftChild}
;
Hiding
  :  '\\' '{' IdentList '}'
  {BehaviourHidingListNode}
;
DefinitionList
  :  Definitions  {ClassDefinitionsListNode}
  ;
Definitions
  :  "empty"  {OrderedChildren}
  |  Definitions  Definition  {liftLeftChild}
  ;
Definition
  :  DataClass
  |  ProcessClass
  |  ClusterClass
;
ClusterClass
  :  'cluster' 'class' Ident Parameters
  'communication' 'channels' IdentList
  'message' 'interface' MessageList
  'behaviour' 'specification' BehaviourSpecification
  {ClusterClassNode}
;
ProcessClass
  :  'process' 'class' Ident Parameters
  'instance' 'variables' DeclarationList
  'communication' 'channels' IdentList
  'message' 'interface' MessageList
  'initial' 'method' 'call' Ident '(' DataExpressionList ')' '(' ')'
  'instance' 'methods' ProcessMethodList
  {ProcessClassNode}
;
DeclarationList
  :  "empty"  {OrderedChildren}
  |  NotEmptyDeclarationList
  |  NotEmptyDeclarationList ';
;
NotEmptyDeclarationList
  :  Declaration  {OrderedChildren}
  |  NotEmptyDeclarationList ';
  ;
Declaration
  :  NotEmptyIdentList ':' Ident
  {DeclarationNode}
;
IdentList
  :  "empty"  {OrderedChildren}
  |  NotEmptyIdentList
;
NotEmptyIdentList
  :  Ident
  |  IdentList ',' Ident
  {OrderedChildren}
  |  NotEmptyIdentList
;
MessageList
  :  "empty"  {OrderedChildren}
  |  NotEmptyMessageList
  |  NotEmptyMessageList ';
;
NotEmptyMessageList
  :  MessageProto
  {OrderedChildren}
  |  NotEmptyMessageList ';
  ;
MessageProto
  :  Ident '?' Ident
  '(' IdentList ')'  {chan:rec:protos:}
  |  Ident '?' Ident
  {chan:rec:}
B POOSL Grammar
B POOSL Grammar

85
; PrimitiveBooleanDataObject
  : 'bunk' {nil} {trueString} {falseString}
  | 'true' {trueString}
  | 'false' {falseString}
;
PrimitiveIntegerDataObject
  : 'iunk' {nil}
  | <int> {asString:}
  | <binNumber> {asString:}
  | <hexNumber> {asString:}
;
PrimitiveRealDataObject
  : 'runk' {nil}
  | <real> {asString:}
;
PrimitiveCharDataObject
  : 'cunk' {nil}
  | <char> {asString:}
;
PrimitiveStringDataObject
  : 'sunk' {nil}
  | <string> {asString:}
;
Ident
  : <ident> {asString:}
;
The translation function \( P2ST_i \) (argument) represents the function that translates the argument, which represents POOSL code, to Smalltalk code. The bold text stands for the substitution of the corresponding syntax definition or translation function. All identifiers that are denoted italic imply that they are substituted literally in the Smalltalk translation.

In POOSL, methods are defined in a method definition. Process method definition and the data method definition have the following syntax (methodDef\(^p\), methodDef\(^d\)).

### Syntax Definitions

**methodDef\(^p\)**

```plaintext
methodDef\(^p\) ::= method(inputParam\(_i\),iType,. . . ,inputParam\(_n\),iType\(_n\))
                    (retParam\(_i\),rType,. . . ,retParam\(_n\),rType\(_n\))
                    | localVar\(_i\),lType,. . . ,localVar\(_n\),lType\(_n\)|
                    statement\(^p\).
```

**statement\(^p\)**

```plaintext
statement\(^p\) ::= E
                | statement\(^d\); statement\(^p\)
                | ch!msg(E,. . . ,E\(_n\))
                | ch!*msg(E,. . . ,E\(_n\))
                | ch?msg(P,. . . ,P\(_n\))
                | ch?msg(P,. . . ,P\(_n\)|E)
                | method(E,. . . ,E\(_n\))(P,. . . ,P\(_n\))
                | sel statement\(^p\)| or . . . or statement\(^p\), les
                | [E] statement\(^p\)
                | statement\(^p\), abort statement\(^p\)
                | statement\(^p\), interrupt statement\(^p\)
                | delay E
                | skip
```

**methodDef\(^d\)**

```plaintext
methodDef\(^d\) ::= method(inputParam\(_i\),iType,. . . ,inputParam\(_n\),iType\(_n\)):retType
                    | localVar\(_i\),lType,. . . ,localVar\(_n\),lType\(_n\)|
                    statement\(^d\).
```

**statement\(^d\)**

```plaintext
statement\(^d\) ::= instVar := E
                | localVar := E
                | inputParam := E
                | retParam := E
                | instParam := E
                | statement\(^d\); statement\(^d\)
                | E
                | while E do statement\(^d\) od
                | if E then statement\(^d\) fi
                | if E then statement\(^d\), else statement\(^d\), fi
                | return(E)
```

C Translation Table for Smalltalk

C Translation Table for Smalltalk
\[ \text{E} \quad ::= \quad \text{instVar} \\
| \quad \text{localVar} \\
| \quad \text{inputParam} \\
| \quad \text{retParam} \\
| \quad \text{instParam} \\
| \quad \text{new} \quad \text{(dataClass)} \\
| \quad \text{self} \\
| \quad \text{E} \quad \text{method(}E, \ldots, E_n\text{)} \\
| \quad \text{E}, \quad \text{binaryOperator} \quad E_2 \\
| \quad \text{unaryOperator} \quad E \\
| \quad \text{constBoolean} \\
| \quad \text{bunk} \\
| \quad \text{constInteger} \\
| \quad \text{iunk} \\
| \quad \text{constReal} \\
| \quad \text{runk} \\
| \quad \text{constChar} \\
| \quad \text{cunk} \\
| \quad \text{constString} \\
| \quad \text{nil} \] only used within a process method

\[ \text{P} \quad ::= \quad \text{instVar} \\
| \quad \text{localVar} \\
| \quad \text{inputParam} \\
| \quad \text{retParam} \\
| \] instParam

### Method Translation Function

\[ \text{PST}_{1}(\text{methodDef}^P) \quad ::= \quad \text{The complete translation of a process method can be accomplished as follows. The POOSL code } \text{methodDef}^P \text{ is passed as a Smalltalk String to the message } \text{parse: methodDef}^P, \text{ which is send to an instance of } \text{POOSLLParser}. \text{ This message returns a reference to the root of the parse tree, i.e. a } \text{ProcessMethodNode}. \text{ The parse tree is then decorated by sending } \text{decorate} \text{ to the } \text{ProcessMethodNode}. \text{ After that, the root } \text{ProcessMethodNode} \text{ can be requested to generate Smalltalk code by sending } \text{stBehaviour} \text{ (which calls } \text{emitST: parameter}). \text{ This message returns a String, which contains the translation of the statements of the method according to their translation function (see below), enclosed in a Smalltalk code block (}\text{[:slf0 | PST}_{2}(\text{statement}^P))}. \text{ The remaining information of the process method (instance variables, used data classes, used channels, et cetera) can be retrieved by sending the appropriate messages to the } \text{ProcessMethodNode}. \]

\[ \text{PST}_{2}(\text{methodDef}^D) \quad ::= \quad \text{The complete translation of a data method can be accomplished as follows. The POOSL code } \text{methodDef}^D \text{ is passed as a Smalltalk String to the message } \text{parse: methodDef}^D, \text{ which is send to an instance of } \text{POOSLLParser}. \text{ This message returns a reference to the root of the parse tree, i.e. a } \text{DataMethodNode}. \text{ The parse tree is then decorated by sending } \text{decorate} \text{ to the } \text{DataMethodNode}. \text{ After that, the root } \text{DataMethodNode} \text{ can be requested to generate Smalltalk code by sending } \text{stBehaviour} \text{ (which calls } \text{emitST: parameter}). \text{ This message returns a String, which contains the translation of the statements of the method according to their translation function (see below), enclosed in a Smalltalk code block (}\text{[:slf0 | PST}_{2}(\text{statement}^D))}. \text{ The remaining information of the data method (instance variables, used data classes, et cetera) can be retrieved by sending the appropriate messages to the } \text{DataMethodNode}. \]
Supporting Translation Functions

\[ \text{ExprArray}(_1, \ldots, _n) ::= \left( \text{Array withAll:} \right. \]
\[ \left. \text{(OrderedCollection new add: } [\text{PST}(_1)] ; \ldots \right. \]
\[ \left. \text{add: } [\text{PST}(_n)]; \text{yourself)} \right) \]

\[ \text{ParamArray}(_1, \ldots, _n) ::= \left( \text{Array withAll:} \right. \]
\[ \left. \text{(OrderedCollection new add: } \text{PST}(_1) ; \ldots \right. \]
\[ \left. \text{add: } \text{PST}(_n); \text{yourself)} \right) \]

\(<\text{slf}> ::= \text{represents a String, which contains 'slf' concatenated with the String} \]
\[ \text{representation of the Integer i. This Integer i is assigned during} \]
\[ \text{decoration and incremented according to the translation table} \]
\[ \text{PST/(statement P) (select, guard, abort, interrupt). } <\text{slf}> \text{ is used by} \]
\[ \text{the POOSL-simulator to refer to the current data object or the executing} \]
\[ \text{process.} \]

\[ \text{CONST(constBoolean)} ::= \text{The Smalltalk representation of constBoolean. In POOSL} \]
\[ \text{constBoolean can be can be true or false in POOSL, which is} \]
\[ \text{the same in Smalltalk, so it is translated literally.} \]

\[ \text{CONST(constInteger)} ::= \text{The Smalltalk representation of constInteger. In POOSL} \]
\[ \text{constInteger can be noted decimal, binary and hexadecimal (123, %101110, $FF3E). The decimal noted integer is translated} \]
\[ \text{literally. The binary and hexadecimal noted integer are translated to the Smalltalk equivalent binary or hexadecimal} \]
\[ \text{notation (2r101110, 16rFF3E).} \]

\[ \text{CONST( constReal )} ::= \text{The Smalltalk representation of constReal. ConstReal is} \]
\[ \text{translated literally to Smalltalk.} \]

\[ \text{CONST( constChar )} ::= \text{The Smalltalk representation of constChar. In POOSL constChar} \]
\[ \text{is a character noted between apostrophes ('A'). For Smalltalk} \]
\[ \text{this is translated by a dollarsign followed by the character ($A)}. \]

\[ \text{CONST( constString) ::= The Smalltalk representation of constString. In POOSL} \]
\[ \text{constString contains one or more characters enclosed in quotes. For the Smalltalk translation the quotes are} \]
\[ \text{translated to apostrophes and all apostrophes within constString are doubled. ConstString can contain double} \]
\[ \text{quotes to represent a quote within the string. These quotes are} \]
\[ \text{translated to one single quote.} \]

Note that all binaryOperators, binary primitive data class operators, are translated literally to Smalltalk. The unaryOperator can be the plus or the minus sign. The plus sign is omitted and the minus sign is translated to -negate- to distinguish it from the binary minus operator.

Before each process statement the code \(<\text{slf}> \text{ checkPoint: nodeNumber} \) is generated for debugging a process method within the POOSL-simulator. Before each data statement the code \(<\text{slf}> \text{ checkPoint: nodeNumber} \) can be generated, depending on the compiler directive \text{debugFlag}. This is used within the POOSL-simulator for debugging a data method. \text{NodeNumber} contains the number of the node, which is assigned during decoration.
**Process Statement Translation Function**

<table>
<thead>
<tr>
<th>Statement(^p)</th>
<th>(P2ST_1(\text{statement}^p))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E)</td>
<td>(&lt;\text{self}&gt;\text{statementRequest.})</td>
</tr>
<tr>
<td>(\text{Statement}^4)</td>
<td>(&lt;\text{self}&gt;\text{evaluate: }{P2ST_1(E)})</td>
</tr>
<tr>
<td>(\text{ch!msg}(E_1, \ldots, E_n))</td>
<td>(&lt;\text{self}&gt;\text{send: 'msg' on: 'ch' with: }{\text{ExprArray}(E_1, \ldots, E_n)})</td>
</tr>
<tr>
<td>(\text{ch!*msg}(E_1, \ldots, E_n))</td>
<td>(&lt;\text{self}&gt;\text{broadcast: 'msg' on: 'ch' with: }{\text{ExprArray}(E_1, \ldots, E_n)})</td>
</tr>
<tr>
<td>(\text{ch?msg}(P_1, \ldots, P_n))</td>
<td>(&lt;\text{self}&gt;\text{receive: 'msg' on: 'ch' with: }{\text{ParamArray}(P_1, \ldots, P_n)})</td>
</tr>
<tr>
<td>(\text{ch?msg}(P_1, \ldots, P_n</td>
<td>E))</td>
</tr>
<tr>
<td>(\text{Method}(E_1, \ldots, E_n)(P_1, \ldots, P_n))</td>
<td>(&lt;\text{self}&gt;\text{execute: 'method' with: }{\text{ExprArray}(E_1, \ldots, E_n)}\text{return: }{\text{ParamArray}(P_1, \ldots, P_n)})</td>
</tr>
<tr>
<td>(\text{method}(E_1, \ldots, E_n)())</td>
<td>(&lt;\text{self}&gt;\text{statementRequest. with: }{\text{ExprArray}(E_1, \ldots, E_n)})</td>
</tr>
<tr>
<td>(\text{sel statement}^1)</td>
<td>(&lt;\text{self}&gt;\text{selectOneOf: (OrderedCollection new add: }{&lt;\text{self}&gt;, {P2ST_1(\text{statement}^p)}}; \ldots \text{add: }{&lt;\text{self}&gt;, {P2ST_1(\text{statement}^p)}}; \text{yourself})}))</td>
</tr>
<tr>
<td>(\text{or ... statement}^n)</td>
<td>(&lt;\text{self}&gt;\text{guard: }{&lt;\text{self}&gt;, {P2ST_1(\text{statement}^p)}}\text{with: }{P2ST_1(E)})</td>
</tr>
<tr>
<td>([E] \text{statement}^p)</td>
<td>(&lt;\text{self}&gt;\text{abort: }{&lt;\text{self}&gt;, {P2ST_1(\text{statement}^p)}}\text{with: }{&lt;\text{self}&gt;, {P2ST_1(\text{statement}^p)}})</td>
</tr>
<tr>
<td>(\text{statement}^p_1 \text{ abort statement}^p_2)</td>
<td>(&lt;\text{self}&gt;\text{interrupt: }{&lt;\text{self}&gt;, {P2ST_1(\text{statement}^p_1)}}\text{with: }{&lt;\text{self}&gt;, {P2ST_1(\text{statement}^p_2)}})</td>
</tr>
<tr>
<td>(\text{statement}^p_1 \text{ interrupt statement}^p_2)</td>
<td>(&lt;\text{self}&gt;\text{delay: }{P2ST_1(E)})</td>
</tr>
<tr>
<td>(\text{delay E})</td>
<td>(&lt;\text{self}&gt;\text{skip})</td>
</tr>
</tbody>
</table>
### Data Statement Translation Function

<table>
<thead>
<tr>
<th>Statement(^d)</th>
<th>(\text{P2ST}_1(\text{statement}(^d)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{instVar} := E)</td>
<td>(&lt;\text{alf}&gt;, \text{assign: } [\text{P2ST}_1(E)]) (\text{toInstVar: } #'\text{instVar}')</td>
</tr>
<tr>
<td>(\text{localVar} := E)</td>
<td>(&lt;\text{alf}&gt;, \text{assign: } [\text{P2ST}_1(E)]) (\text{toLocalVar: } #'\text{localVar}')</td>
</tr>
<tr>
<td>(\text{InputParam} := E)</td>
<td>(&lt;\text{alf}&gt;, \text{assign: } [\text{P2ST}_1(E)]) (\text{toLocalVar: } #'\text{inputParam}')</td>
</tr>
<tr>
<td>(\text{RetParam} := E)</td>
<td>(&lt;\text{alf}&gt;, \text{assign: } [\text{P2ST}_1(E)]) (\text{toLocalVar: } #'\text{retParam}')</td>
</tr>
<tr>
<td>(\text{InstParam} := E)</td>
<td>(&lt;\text{alf}&gt;, \text{assign: } [\text{P2ST}_1(E)]) (\text{toInstVar: } #'\text{instParam}')</td>
</tr>
<tr>
<td>(\text{statement}(^d); (\text{statement}(^d))</td>
<td>(\text{P2ST}_1(\text{statement}(^d))), (\text{P2ST}_1(\text{statement}(^d)))</td>
</tr>
<tr>
<td>(E)</td>
<td>(\text{P2ST}_1(E))</td>
</tr>
<tr>
<td>while (E) do</td>
<td>(&lt;\text{alf}&gt;, \text{while: } [\text{P2ST}_1(E)]) (\text{do: } [\text{P2ST}_1(\text{statement}(^d))])</td>
</tr>
<tr>
<td>(\text{statement}(^d))</td>
<td>(&lt;\text{alf}&gt;, \text{if: } [\text{P2ST}_1(E)]) (\text{then: } [\text{P2ST}_1(\text{statement}(^d))])</td>
</tr>
<tr>
<td>(\text{fi})</td>
<td>(&lt;\text{alf}&gt;, \text{if: } [\text{P2ST}_1(E)]) (\text{then: } [\text{P2ST}_1(\text{statement}(^d))])</td>
</tr>
<tr>
<td>(\text{else})</td>
<td>(&lt;\text{alf}&gt;, \text{if: } [\text{P2ST}_1(E)]) (\text{then: } [\text{P2ST}_1(\text{statement}(^d))])</td>
</tr>
<tr>
<td>(\text{statement}(^d))</td>
<td>(&lt;\text{alf}&gt;, \text{if: } [\text{P2ST}_1(E)]) (\text{then: } [\text{P2ST}_1(\text{statement}(^d))])</td>
</tr>
<tr>
<td>(\text{else})</td>
<td>(&lt;\text{alf}&gt;, \text{if: } [\text{P2ST}_1(E)]) (\text{then: } [\text{P2ST}_1(\text{statement}(^d))])</td>
</tr>
<tr>
<td>(\text{fi})</td>
<td>(&lt;\text{alf}&gt;, \text{if: } [\text{P2ST}_1(E)]) (\text{then: } [\text{P2ST}_1(\text{statement}(^d))])</td>
</tr>
<tr>
<td>(\text{return}(E))</td>
<td>(\text{P2ST}_1(E))</td>
</tr>
</tbody>
</table>

C Translation Table for Smalltalk
### Expression Translation Function

<table>
<thead>
<tr>
<th>E</th>
<th>(P_{2ST}(E))</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstVar</td>
<td>(&lt;\text{self}&gt;) instVar: '#instVar'</td>
</tr>
<tr>
<td>InstParam</td>
<td>(&lt;\text{self}&gt;) instVar: '#InstParam'</td>
</tr>
<tr>
<td>LocalVar</td>
<td>(&lt;\text{self}&gt;) localVar: '#localVar'</td>
</tr>
<tr>
<td>InputParam</td>
<td>(&lt;\text{self}&gt;) localVar: '#inputParam'</td>
</tr>
<tr>
<td>RetParam</td>
<td>(&lt;\text{self}&gt;) localVar: '#retParam'</td>
</tr>
<tr>
<td>new(dataClass)</td>
<td>(PDataClass withName: '#dataClass') new</td>
</tr>
<tr>
<td>Self</td>
<td>(&lt;\text{self}&gt;)</td>
</tr>
<tr>
<td>E method(E₁,...,Eₙ)</td>
<td>(P_{2ST}(E)) execute: '#method'</td>
</tr>
<tr>
<td></td>
<td>with: ExprArray(E₁,...,Eₙ)</td>
</tr>
<tr>
<td>E₂ binaryOperator E₃</td>
<td>(P_{2ST}(E₂)) execute: '#binaryOperator'</td>
</tr>
<tr>
<td></td>
<td>with: (Array with: (P_{2ST}(E₃)))</td>
</tr>
<tr>
<td>UnaryOperator E</td>
<td>(P_{2ST}(E)) execute: '#unaryOperator'</td>
</tr>
<tr>
<td>ConstBoolean</td>
<td>PBoolean with: (\text{CONST}(\text{constBoolean}))</td>
</tr>
<tr>
<td>Bunk</td>
<td>PBoolean unknown</td>
</tr>
<tr>
<td>ConstInteger</td>
<td>PInteger with: (\text{CONST}(\text{constInteger}))</td>
</tr>
<tr>
<td>Int</td>
<td>PInteger unknown</td>
</tr>
<tr>
<td>ConstReal</td>
<td>PReal with: (\text{CONST}(\text{constReal}))</td>
</tr>
<tr>
<td>Real</td>
<td>PReal unknown</td>
</tr>
<tr>
<td>ConstChar</td>
<td>PChar with: (\text{CONST}(\text{constChar}))</td>
</tr>
<tr>
<td>Char</td>
<td>PChar unknown</td>
</tr>
<tr>
<td>ConstString</td>
<td>PString with: (\text{CONST}(\text{constString}))</td>
</tr>
<tr>
<td>String</td>
<td>PString instance</td>
</tr>
<tr>
<td>Nil</td>
<td>PNil instance</td>
</tr>
</tbody>
</table>

### Parameter Translation Function

<table>
<thead>
<tr>
<th>P</th>
<th>(P_{2ST}(P))</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstVar</td>
<td>'#instVar'</td>
</tr>
<tr>
<td>InputParam</td>
<td>'#inputParam'</td>
</tr>
<tr>
<td>RetParam</td>
<td>'#retParam'</td>
</tr>
<tr>
<td>LocalVar</td>
<td>'#localVar'</td>
</tr>
<tr>
<td>InstParam</td>
<td>'#instParam'</td>
</tr>
</tbody>
</table>
The translation function $P2CPP(SSpec)$ represents the function that translates a POOSL system specification into C++. The **bold** text stands for the substitution of the corresponding syntax definition or translation function. Text printed in *italic* stands for the substitution of the corresponding POOSL identifier.

POOSL variable and parameter identifiers are translated into strings in the C++ code. This enables the C++ implementation to print debugging information during program execution. To increase run-time performance, the strings used in data expressions (E) may be replaced by ID numbers. The ID number of a variable/parameter is its position number (starting at position 0) in the corresponding variable/parameter list. Instance variables and instantiation parameters must be in the same list (of instance variables) starting with the instantiation parameters.

So, to disable variable/parameter debugging information and increase run-time performance, apply the following to the POOSL to C++ mapping:

- Replace all the strings in solid lined boxes by the corresponding identifier ID numbers.
- Leave out the code in dotted lined boxes.

Blocks are used to indicate a composition of process statements. A condition indicates a Boolean expression used for a receive or guard condition. Each block or condition is translated into a separate C++ member function. Blocks can be nested in select, abort and interrupt statements. A block or condition is designated by a variable number of indices. The first index indicates sequence number of the statement at nesting level 1 (the base level) where the block or condition is nested in. The second index indicates its sequence number at nesting level 2, and so on. This way it is always possible to find the indices of a block or condition when its location in the POOSL specification is known.
Syntax Definitions

**SSpec** ::= \{ BSpec, Sys^p, Sys^q, Sys^r \}

**Sys^p** ::= classDef^p; ...; classDef^r

**Sys^q** ::= classDef^q; ...; classDef^r

**Sys^r** ::= classDef^r; ...; classDef^r

classDef^p ::= data class dataClass
instance variables instVar^p:ivType^p; ...; instVar^r:ivType^r
instance methods methodDef^p; ...; methodDef^r

methodDef^p ::= method(inputParam^p:Type^p; ...; inputParam^r:Type^r):mType
[localVar^p:Type^p; ...; localVar^r:Type^r]
statement^p

classDef^q ::= process class
processClass(instParam^p:Type^p; ...; instParam^r:Type^r)
instantiation variables instVar^p:ivType^p; ...; instVar^r:ivType^r
communication channels ch^p; ...; ch^r
message interface l^p; ...; l^r
initial method call initialMethod(E^p; ...; E^r)()
instance methods methodDef^p; ...; methodDef^r

methodDef^q ::= method(inputParam^p:Type^p; ...; inputParam^r:Type^r)
[retParam^p:Type^p; ...; retParam^r:Type^r]
[localVar^p:Type^p; ...; localVar^r:Type^r]
block.

block^p ::= statement^p; ...; statement^p

statement^p ::= statement^q
| ch! msg(E^p; ...; E^r)
| ch" msg(E^p; ...; E^r)
| ch? msg(F^p; ...; F^r)
| [condition^p; ...; condition^q] block^p; ...; block^r
| sel block^p; ...; block^r or ... or block^p; ...; block^r
| if E then block^p; ...; block^r else block^p; ...; block^r fi
| if E then block^p; ...; block^r fi
| while E do block^p; ...; block^r od
| [condition^p; ...; condition^q] block^p; ...; block^r
| abort block^p; ...; block^r
| interrupt block^p; ...; block^r
| delay E
| skip

P ::= instVar
| inputParam
| retParam
| localVar
| instParam

condition^p; ...; condition^q ::= E

classDef^r ::= cluster class
clusterClass(exprParam^p:Type^p; ...; exprParam^r:Type^r)
communication channels ch^p; ...; ch^r
message interface l^p; ...; l^r
behaviour specification SSpec

D Translation Table for C++
BSpec ::= class E₁, ⋯, Eₙ
    class Spec E₁, ⋯, Eₙ
    BSpec | BSpec,
    BSpec \ (hiding₁, ⋯, hidingₙ)
    BSpec [newLabel₁/oldLabel₁, ⋯, newLabelₙ/oldLabelₙ]

statementₜ ::= instVar := E
    localVar := E
    inputParam := E
    retParam := E
    instParam := E
    statementₜ; statementₜ,
    E
    while E do statementₜ od
    if E then statementₜ fi
    if E then statementₜ; else statementₜ; fi
    return (E)

E ::= instVar
    localVar
    inputParam
    retParam
    instParam
    exprParam
    new (dataClass)
    self
    E method E₁, ⋯, Eₙ
    E, binaryOperator E₂
    unaryOperator E
    true
    false
    bunk
    constInteger
    junk
    constReal
    runk
    constCharacter
    cunk
    constString
    nil
Definitions Required for the Translation Function

\[ e, e_1, \cdots, e_n \in \{ \text{block}_{i_1, \cdots, i_j} | j, i_1, \cdots, i_j \in \mathbb{N} \} \cup \{ \text{condition}_{i_1, \cdots, i_j} | j, i_1, \cdots, i_j \in \mathbb{N} \} \]

\( (e_1, \cdots, e_n) + (e_{n+1}, \cdots, e_{m}) = (e_1, \cdots, e_n, e_{n+1}, \cdots, e_{m}) \)

\( \text{getFirst}((e_1, \cdots, e_n)) = e \)

\( \text{discardFirst}((e_1, \cdots, e_n)) = (e_1, \cdots, e_n) \)

\( |(e_1, \cdots, e_n)| = n \)

\[ \text{LIST(block}_{i_1, \cdots, i_j}) := (\text{block}_{i_1, \cdots, i_j}) + \text{SCAN}(\text{block}_{i_1, \cdots, i_j}) \]

\[ \text{SCAN(block}_{i_1, \cdots, i_j}) := \text{LIST}(@\text{statement}_{i_1, \cdots, i_j, 1}) + \cdots + \text{LIST}(@\text{statement}_{i_1, \cdots, i_j, n}) \]

\[
\text{PROTOTYPE}(\text{list}) := \begin{cases} \text{PROTOTYPE}(@\text{getFirst}(\text{list})) \\ \text{PROTOTYPE}(@\text{discardFirst}(\text{list})) \end{cases}, \text{if } |\text{list}| > 0, \text{otherwise}
\]

\[
\text{P2CPP}(\text{list}) := \begin{cases} \text{P2CPP}(@\text{getFirst}(\text{list})) \\ \text{P2CPP}(@\text{discardFirst}(\text{list})) \end{cases}, \text{if } |\text{list}| > 0, \text{otherwise}
\]

\[
\text{IF}(\text{cond}, \text{thenText}, \text{elseText}) := \begin{cases} \text{thenText} \text{,if } \text{cond} \text{ is true} \\ \text{elseText} \text{,otherwise} \end{cases}
\]

\[
\text{IF}(\text{cond}, \text{thenText}) := \text{IF}(\text{cond}, \text{thenText}, \_)
\]

\[ \text{INPUTPARAM}(x) := \begin{cases} \text{true}, \text{if expression/block/condition } x \text{ refers to an } \\ \text{false}, \text{otherwise} \end{cases} \]

\[ \text{RETPARAM}(x) := \begin{cases} \text{true}, \text{if expression/block/condition } x \text{ refers to a } \text{retParam} \\ \text{false}, \text{otherwise} \end{cases} \]

\[ \text{LOCALVAR}(x) := \begin{cases} \text{true}, \text{if expression/block/condition } x \text{ refers to a } \text{localVar} \\ \text{false}, \text{otherwise} \end{cases} \]
**CONST(constInteger) ::= C++ representation of constInteger (a long)**

**CONST(constReal) ::= C++ representation of constReal (a double)**

**CONST(constChar) ::= C++ representation of constChar (a char)**

**CONST(constString) ::= C++ representation of constString (a const char *)**

Note that the C++ class names of POOSL primitive data classes are: Boolean, Integer, Real, and Char. The string is a user primitive data class String.

Note that the conversion by `CONST()` applies constraints to the value range of POOSL constants.

Note that `system` is the name of the system.

Note that the C++ identifiers that represent POOSL identifiers have an underscore (_) appended to their original name. This is done to prevent conflicts with C++ keywords and library functions.

Note that `unaryOperator` and `binaryOperator` stand for the textual representation of primitive data class operators. All operators are translated literally, except for $E_1 = E_2$ which has to be translated into `(P2CPP(E_1).isEqual(P2CPP(E_2)))`.

The behaviour specification (BSpec) of a cluster, must be broken down into instances of clusters, processes and channels and their interconnection. Because this is very complex to specify formally, the following "magic functions" are used:

- **ID(class)** ::= unique identification number for every instance of cluster or process class
- **ID** ::= identifier of the corresponding cluster or process instance
- **CHID()** ::= unique channel identifier ()
- **CHID** ::= identifier of the corresponding channel instance
Translation Function

```
P:2CPP(SSpec) ::= P:2CPP(Sys4) + P:2CPP(Sys5) + P:2CPP(Sys6) + P:2CPP_SYS(SSpec)

P:2CPP(Sys4) ::= P:2CPP(classDef4,) + P:2CPP(classDef4n)

P:2CPP(Sys5) ::= P:2CPP(classDef5,) + P:2CPP(classDef5n)

P:2CPP(Sys6) ::= P:2CPP(classDef6,) + P:2CPP(classDef6n)

P2CPP(SYS4) ::= P2CPP(classDef4,) + P2CPP(classDef4n)

P2CPP(classDef4) ::= HEADERFILE(classDef4) + CPPFILE(classDef4)

P2CPP(classDef5) ::= HEADERFILE(classDef5) + CPPFILE(classDef5)

P2CPP(classDef6) ::= HEADERFILE(classDef6) + CPPFILE(classDef6)

P2CPP(SYS5) ::= P2CPP(classDef5,) + P2CPP(classDef5n)

P2CPP_SYS(SSpec) ::= HEADERFILE(SSpec) + CPPFILE(SSpec)

HEADERFILE(classDef2)  File dataClass.h

#include "system.h"

class dataClass_ : public NonPrimitiveDataObject
{
dataClass_() : NonPrimitiveDataObject() { }
public:
static dataClass_& New() { return *(new dataClass_); }
const char* GetClassName() { return "dataClass"; }
sizet GetClassSize() { return sizeof(dataClass_); }
PROTOTYPE(methodDef4,)
...  PROTOTYPE(methodDef5,)
};
#endif
```
```cpp
CPPFILE(classDef d)

#include "system.h"

P2CPP(methodDef d);
...
P2CPP(methodDef d);

HEADERFILE(classDef p)

#ifndef __processClass__
define __processClass__
#endif

#include "system.h"

I File dataClass.cc
I File processClass.h

class processClass_ : public ProcessIF(v+u > 0, public VarArray)
{
    public:
        ProcessConnector *ch_;
        ...;
        ProcessConnector *ch_;
        processClass_(Cluster *, const char *);
    void StartUp(IF(n > O, ExprArray &));
    PROTOTYPE(methodDef p);
    ...
    PROTOTYPE(methodDef p);
};
#endif

CPPFILE(classDef p)

#include "system.h"

processClass_::processClass_(Cluster *owner, const char *name)
    : Process(owner, name)IF(v+u > 0,
    VarArray(v+u,"instParam","...","instParam","instVar","...","instVar"))
    {
        ch_ = new ProcessConnector(this,"ch_");
        ...;
        ch_ = new ProcessConnector(this,"ch_");
    }

void processClass_::StartUp(IF(u > O, ExprArray &)I)
    {
        IF(u > 0,Assign(i.DeepCopy()));
        InitialMethodCall((ProcessMethod)&processClass_::initialMethod_IF(n > 0,
            new ExprArray(n, &P2CPP(E),
            ...,
            &P2CPP(E)));
    }

P2CPP(methodDef p);
...
P2CPP(methodDef p);
```
```cpp
#ifndef _clusterClass_
#define _clusterClass_

#include "system.h"

class clusterClass_ : public Cluster
{
    public:
    ClusterConnector *ch_; 
    ...;
    ClusterConnector *ch_; 
    Channel *CHID(); 
    ...;
    Channel *CHID();
    PROTOTYPE(BSpec)
    clusterClass_(Cluster *const char *); 
    void StartUp(IF(u > 0,ExprArray &)); 
};
#endif

#include "system.h"

clusterClass_::clusterClass_(Cluster *owner, const char *name) : Cluster(owner, name)
{
    ch_ = new ClusterConnector(this,"ch_" );
    ...;
    ch_ = new ClusterConnector(this,"ch_" );
    CHID = new Channel(this,"CHID" );
    ...;
    CHID = new Channel(this,"CHID" );
    CONSTRUCT(BSpec)
    CONNECT(BSpec)
}

clusterClass_::StartUp(IF(u > 0,ExprArray &))
{
    IF(u > 0,i.SetNames("exprParam",...,"exprParam");)
    STARTUP(BSpec)
}
```
```c++
#ifndef __system__
#define __system__

class dataClass_; 
...; 
class dataClass_; 

for every data class in Sys

class processClass_; 
...; 
class processClass_; 

for every process class in Sys

class clusterClass_; 
...; 
class clusterClass_; 

for every cluster class in Sys

#include "process.h"

#include "dataClass_"
...;
#include "dataClass_"

for every data class in Sys

#include "processClass_"
...;
#include "processClass_"

for every process class in Sys

#include "clusterClass_"
...;
#include "clusterClass_"

for every cluster class in Sys

#endif
```
```cpp
#include "system.h"

class System : public Cluster
{
    public:
        Channel *CHID();
        ...;
        Channel *CHID();
        PROTOTYPE(BSpec)
        System(const char *);  
        void StartUp();
    }

    System::System(const char *name)
    {
        Cluster(NULL, name)
        {
            CHID = new Channel(this,"CHID");
            ...;
            CHID = new Channel(this,"CHID");
            CONSTRUCT(BSpec)
            CONNECT(BSpec)
        }
        System::StartUp()
        {
            STARTUP(BSpec)
        }

    int main(void)
    {
        System Sys("system");
        Sys.StartUp();
        scheduler.Run();
        return ( 0 );
    }

PROTOTYPE(methodDef^)

mType_& method_(IF(p > 0,ExprArray));

P2CPP(methodDef^)

mType_& dataClass_::method_(IF(p > 0,ExprArray i))
{
    LocalArray 1(k + 1,self,"localVar","localVar");
    IF(p > 0,::SetNames("inputParam","inputParam"));
    P2CPP(statements);
}

PROTOTYPE(methodDef^)

ReturnObject *method_(ProcessNode *IF(p > 0,, ExprArray &));

list := LIST(block)
PROTOTYPE(list)
```
P2CPP(methodDef P)

list := LIST(block)

ReturnObject *processClass::method_(ProcessNode *MyNodeIF(p > 0, ExprArray &i))
{ ReturnObject *ret = NULL;
  IF(k > 0,VarArray &i = *(new VarArray(k,"localVar","localVar"));)
  IF(q > 0,RetArray &r = *(new RetArray(q,"retParam","retParam"));)
  IF(p > 0,i.SetNames("inputParam","inputParam");)
  IF(k > 0,MyNode->Push(&i);)
  IF(q > 0,MyNode->Push(&r);)
  P2CPP(getFirst(list))
  IF(q > 0,ret = &r;)
  IF(q > 0,MyNode->Pop();)
  IF(k > 0,delete MyNode->Pop();)
  return ( ret );
}

P2CPP(discardFirst(list))

<table>
<thead>
<tr>
<th>BSpec</th>
<th>PROTOTYPE (BSpec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>class^E,\ldots,E_n</td>
<td>class^E *class^E_ID(class^E);</td>
</tr>
<tr>
<td>class^E,\ldots,E_n</td>
<td>class^E *class^E_ID(class^E);</td>
</tr>
<tr>
<td>BSpec_1</td>
<td></td>
</tr>
<tr>
<td>BSpec \ [hiding_1,\ldots,hiding_n]</td>
<td>PROTOTYPE(Bspec)</td>
</tr>
<tr>
<td>BSpec[newLabel_1/oldLabel_1, \ldots,newLabel_n/oldLabel_n]</td>
<td>PROTOTYPE(Bspec)</td>
</tr>
<tr>
<td>BSpec</td>
<td>CONSTRUCT(BSpec)</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td><code>class^{(E_1, \ldots , E_n)}</code></td>
<td><code>class^{_ID} = new class^{_(this,&quot;class^{_ID&quot;});</code></td>
</tr>
<tr>
<td><code>class^{(E_1, \ldots , E_n)}</code></td>
<td><code>class^{_ID} = new class^{_(this,&quot;class^{_ID&quot;});</code></td>
</tr>
<tr>
<td>`BSpec_1</td>
<td></td>
</tr>
<tr>
<td><code>BSpec \ \{\text{hiding}_1, \ldots ,\text{hiding}_n\}</code></td>
<td><code>CONSTRUCT(Bspec)</code></td>
</tr>
<tr>
<td><code>BSpec[newLabel_1/oldLabel_1,</code></td>
<td><code>CONSTRUCT(Bspec)</code></td>
</tr>
<tr>
<td><code>\ldots , newLabel_n/oldLabel_n]\</code></td>
<td></td>
</tr>
</tbody>
</table>
### CONNECT (BSpec)

| Channel ch must be connected to process connector pc of process instance class\_ID | ch->Connect(class\_ID->pc); |
| Channel ch must be connected to cluster connector cc of cluster instance class\_ID | ch->Connect(class\_ID->cc); |
| Channel ch must be connected to cluster connector cc of this cluster | ch->Connect(cc); |

### STARTUP (BSpec)

| Class \( E, \ldots, E_n \) | class\_ID->StartUp(IF(n > 0, *(new ExprArray(n, \&P2CPP(E), \ldots, \&P2CPP(E_n) ) ))); |
| Class \( E, \ldots, E_n \) | class\_ID->StartUp(IF(n > 0, *(new ExprArray(n, \&P2CPP(E), \ldots, \&P2CPP(E_n) ) ))); |

<p>| BSpec _ <strong>|</strong> BSpec2 | STARTUP (BSpec_1) STARTUP (BSpec_2) |
| BSpec \ Backslash (hiding_1, \ldots, hiding_n) | STARTUP (BSpec) |
| BSpec [newLabel_1/oldLabel_1, \ldots, newLabel_n/oldLabel_n] | STARTUP (BSpec) |</p>
<table>
<thead>
<tr>
<th>statement(^{d})</th>
<th>P2CPP(statement(^{d}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>instVar := E</td>
<td>self.Assign(&quot;instVar&quot;, P2CPP(E))</td>
</tr>
<tr>
<td>localVar := E</td>
<td>i.Assign(&quot;localVar&quot;, P2CPP(E))</td>
</tr>
<tr>
<td>inputParam := E</td>
<td>i.Assign(&quot;inputParam&quot;, P2CPP(E))</td>
</tr>
<tr>
<td>retParam := E</td>
<td>r.Assign(&quot;retParam&quot;, P2CPP(E))</td>
</tr>
<tr>
<td>instParam := E</td>
<td>self.Assign(&quot;instParam&quot;, P2CPP(E))</td>
</tr>
<tr>
<td>statement(^{d}<em>{1}); statement(^{d}</em>{2})</td>
<td>P2CPP(statement(^{d}<em>{1})); P2CPP(statement(^{d}</em>{2}))</td>
</tr>
<tr>
<td>E</td>
<td>P2CPP(E).Cleanup()</td>
</tr>
<tr>
<td>while E do</td>
<td>while (P2CPP(E).IsTrue())</td>
</tr>
<tr>
<td>statement(^{d})</td>
<td>{ P2CPP(statement(^{d})); }</td>
</tr>
<tr>
<td>od</td>
<td></td>
</tr>
<tr>
<td>if E then</td>
<td>if (P2CPP(E).IsTrue())</td>
</tr>
<tr>
<td>statement(^{d})</td>
<td>{ P2CPP(statement(^{d})); }</td>
</tr>
<tr>
<td>fi</td>
<td></td>
</tr>
<tr>
<td>if E then</td>
<td>if (P2CPP(E).IsTrue())</td>
</tr>
<tr>
<td>statement(^{d}_{1})</td>
<td>{ P2CPP(statement(^{d}_{1})); }</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>statement(^{d}_{2})</td>
<td>{ P2CPP(statement(^{d}_{2})); }</td>
</tr>
<tr>
<td>fi</td>
<td></td>
</tr>
<tr>
<td>return(E)</td>
<td>return (mType_&amp;) Return(P2CPP(E))</td>
</tr>
<tr>
<td><strong>statement</strong>[^1]</td>
<td><strong>LIST(statement)</strong>[^2]</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>( \text{statement}' )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{ch!msg}(E_1, \ldots, E_n) )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{ch!*msg}(E_1, \ldots, E_n) )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{ch?msg}(P_1, \ldots, P_n) )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{ch?msg}(P_1, \ldots, P_n \mid \text{condition}_{a_1, \ldots, s_j}) )</td>
<td>( \text{(condition}_{a_1, \ldots, s_j}) )</td>
</tr>
<tr>
<td>( \text{method}(E_1, \ldots, E_n \mid (P_1, \ldots, P_n) )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{sel} )</td>
<td>( \text{LIST(block}<em>{a_1, \ldots, s_j, 1} ) + \cdots + ( \text{LIST(block}</em>{a_1, \ldots, s_j, n} )</td>
</tr>
<tr>
<td>( \text{or} )</td>
<td>( \text{LIST(block}<em>{a_1, \ldots, s_j, 1} ) + \text{LIST(block}</em>{a_1, \ldots, s_j, s} )</td>
</tr>
<tr>
<td>( \text{or} )</td>
<td>\text{LIST(block}_{a_1, \ldots, s_j, 1} )</td>
</tr>
<tr>
<td>( \text{[condition}<em>{a_1, \ldots, s_j} } \text{block}</em>{a_1, \ldots, s_j} )</td>
<td>( \text{(condition}<em>{a_1, \ldots, s_j}) ) + \text{LIST(block}</em>{a_1, \ldots, s_j} )</td>
</tr>
<tr>
<td>( \text{if } E \text{ then} )</td>
<td>\text{SCAN(block}<em>{a_1, \ldots, s_j, 1} ) + \text{SCAN(block}</em>{a_1, \ldots, s_j, 2} )</td>
</tr>
<tr>
<td>( \text{block}_{a_1, \ldots, s_j, 1} )</td>
<td>( \text{SCAN(block}_{a_1, \ldots, s_j} )</td>
</tr>
<tr>
<td>( \text{else} )</td>
<td>( \text{LIST(block}<em>{a_1, \ldots, s_j, 1} ) + \text{LIST(block}</em>{a_1, \ldots, s_j, 1} )</td>
</tr>
<tr>
<td>( \text{block}_{a_1, \ldots, s_j, 2} )</td>
<td>( \text{LIST(block}<em>{a_1, \ldots, s_j, 1} ) + \text{LIST(block}</em>{a_1, \ldots, s_j, 1} )</td>
</tr>
<tr>
<td>( \text{fi} )</td>
<td>( \text{LIST(block}_{a_1, \ldots, s_j} )</td>
</tr>
<tr>
<td>( \text{if } E \text{ then block}_{a_1, \ldots, s_j} \text{ fi} )</td>
<td>( \text{SCAN(block}_{a_1, \ldots, s_j} )</td>
</tr>
<tr>
<td>( \text{while } E \text{ do block}_{a_1, \ldots, s_j} \text{ od} )</td>
<td>( \text{SCAN(block}_{a_1, \ldots, s_j} )</td>
</tr>
<tr>
<td>\text{Block}<em>{a_1, \ldots, s_j, 1} \text{ abort block}</em>{a_1, \ldots, s_j, 2} )</td>
<td>( \text{LIST(block}<em>{a_1, \ldots, s_j, 1} ) + \text{LIST(block}</em>{a_1, \ldots, s_j, 1} )</td>
</tr>
<tr>
<td>\text{Block}<em>{a_1, \ldots, s_j, 1} \text{ interrupt block}</em>{a_1, \ldots, s_j, 2} )</td>
<td>( \text{LIST(block}<em>{a_1, \ldots, s_j, 1} ) + \text{LIST(block}</em>{a_1, \ldots, s_j, 1} )</td>
</tr>
<tr>
<td>\text{Delay } E )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

---

[^1]: \( s_j \)\( \text{ is a subscript.} \)

[^2]: \( \text{LIST is a function.} \)
PROTOTYPE(block_{s1, ..., sj})
Tail *method_Block_s, __ sj(ProcessNode *, ExprArray &, RetArray &, VarArray &);

PROTOTYPE(condition_{s1, ..., sj})
Boolean &method_Cond_s, __ ._sj(ExprArray &, RetArray &, VarArray &);

P2CPP(condition_{s1, ..., sj})
Boolean &processClass_::method_Cond_s, sj(ExprArray &, RetArray &), VarArray &)
{
    return ( P2CPP(E) );
}

P2CPP(block_{s1, ..., sj})
Tail *processClass_::method_Block_s, __ sj(ProcessNode *MyNode,
    ExprArray &IF(INPUTPARAM(E),i), RetArray &IF(RETPARAM(E),r), VarArray &IF(LOCALVAR(E),i))
{
    Tail *ret = NULL;
    P2CPP(statement_{s1, ..., sj})
    ...;
    P2CPP(statement_{s1, ..., sj})
    return ( ret );
}
<table>
<thead>
<tr>
<th>E</th>
<th>P2CPP(E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>instVar</td>
<td>((ivType_&amp;) self(&quot;instVar&quot;))</td>
</tr>
<tr>
<td>localVar</td>
<td>((lType_&amp;) 1(&quot;localVar&quot;))</td>
</tr>
<tr>
<td>inputParam</td>
<td>((iType_&amp;) i(&quot;inputParam&quot;))</td>
</tr>
<tr>
<td>retParam</td>
<td>((rType_&amp;) r(&quot;retParam&quot;))</td>
</tr>
<tr>
<td>instParam</td>
<td>((ipType_&amp;) self(&quot;instParam&quot;))</td>
</tr>
<tr>
<td>exprParam</td>
<td>((epType_&amp;) e(&quot;exprParam&quot;) . DeepCopy())</td>
</tr>
<tr>
<td>new(dataClass)</td>
<td>dataClass_::New()</td>
</tr>
<tr>
<td>self</td>
<td>self</td>
</tr>
</tbody>
</table>

**E method(E, ... , E)**

\[
P2CPP(E).method_ (IF(n > 0, 
ExprArray(n, &P2CPP(E), ..., &P2CPP(E_n)) 
))
\]

**E, binaryOperator E_2**

\[
(P2CPP(E_1) binaryOperator P2CPP(E_2))
\]

**unaryOperator E_1**

\[
(unaryOperator P2CPP(E_1))
\]

**true**

Boolean::New(TRUE)

**false**

Boolean::New(FALSE)

**bunk**

Boolean::Unk()

**constInteger**

Integer::New(CONST(constInteger))

**iunk**

Integer::Unk()

**constReal**

Real::New(CONST(constReal))

**runk**

Real::Unk()

**constChar**

Char::New(CONST(constChar))

**cunk**

Char::Unk()

**constString**

String_::New(CONST(constString))

**nil**

DataObject::Nil()
<table>
<thead>
<tr>
<th><strong>statement^*_{i, \ldots, n}</strong></th>
<th><strong>P2CPP(statement^*_{i, \ldots, n})</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>statement^*</code></td>
<td>MyNode-&gt;Statement(); P2CPP(statement^*);</td>
</tr>
<tr>
<td><code>ch!msg(E, \ldots, E_n)</code></td>
<td>MyNode-&gt;Send(ch_, &quot;msg(n)&quot; IF(n &gt; 0, new ExprArray(n, &amp;P2CPP(E), \ldots, &amp;P2CPP(E_n))) );</td>
</tr>
<tr>
<td><code>ch?msg(E, \ldots, E_n)</code></td>
<td>MyNode-&gt;Broadcast(ch_, &quot;msg(n)&quot; IF(n &gt; 0, new ExprArray(n, &amp;P2CPP(E), \ldots, &amp;P2CPP(E_n)) );</td>
</tr>
<tr>
<td><code>ch?msg(P, \ldots, P_n)</code></td>
<td>MyNode-&gt;Receive(ch_, &quot;msg(m)&quot; IF(m &gt; 0, new ParamArray(m, P2CPP(P), \ldots, P2CPP(P_n)) );</td>
</tr>
<tr>
<td><code>ch?msg(P, \ldots, P_n)</code></td>
<td>MyNode-&gt;Receive(ch_, &quot;msg(m)&quot; IF(m &gt; 0, new ParamArray(m, P2CPP(P), \ldots, P2CPP(P_n)) );</td>
</tr>
<tr>
<td><code>ch?msg(P, \ldots, P_n)</code></td>
<td>MyNode-&gt;Receive(ch_, &quot;msg(m)&quot; IF(m &gt; 0, new ParamArray(m, P2CPP(P), \ldots, P2CPP(P_n)) );</td>
</tr>
<tr>
<td><code>method(E, \ldots, E_n)</code> (P, \ldots, P_n) (non tail-recursive)</td>
<td>MyNode-&gt;Statement(); MyNode-&gt;Call(this,(ProcessMethod)&amp;processClass_::method_, IF(n &gt; 0, new ExprArray(n, &amp;P2CPP(E), \ldots, &amp;P2CPP(E_n)), NULL), IF(m &gt; 0, new ParamArray(m, P2CPP(P), \ldots, P2CPP(P_n)), NULL));</td>
</tr>
</tbody>
</table>

D Translation Table for C++
method(E, \ldots, E_n)() \hfill MyNode->Statement(); \hfill \text{(tail-recursive)}

\begin{align*}
\text{ret} &= \text{new} \\
\text{T} \text{ail}((\text{ProcessMethod}) & \text{processClass::method_}, \\
\text{IF}(n > 0, \text{new ExprArray\{n,} \\
& \text{P2CPP(E)}, \\
\ldots, \\
& \text{P2CPP(E_n),} \\
\text{NULL}) \\)
\end{align*}

```
if E then \\
\text{block}_s, \ldots, \text{block}_n \\
\text{else} \\
\text{block}_s, \ldots, \text{block}_n \\
\text{fi}
```

```cpp
MyNode->Statement(); \\
if ( \text{P2CPP(E).IsTrue}() ) \\
{
  \text{P2CPP(block}_s, \ldots, \text{block}_n,)
} \\
else \\
{
  \text{P2CPP(block}_s, \ldots, \text{block}_n)
}
```

```cpp
\text{if} \ E \ \text{then} \\
\text{block}_s, \ldots, \text{block}_n \\
\text{fi}
```

```cpp
\text{if} \ E \ \text{then} \\
\text{block}_s, \ldots, \text{block}_n \\
\text{fi}
```

```cpp
\text{while} \ E \ \text{do} \ \text{block}_s, \ldots, \text{block}_n \ \text{od}
```

\text{Translation Table for C++}

```cpp
MyNode->Statement(); \\
\text{if} \ ( \text{P2CPP(E).IsTrue}() ) \\
\{ \\
  \text{P2CPP(block}_s, \ldots, \text{block}_n) \\
\} \\
```

```cpp
\text{while} \ ( \text{P2CPP(E).IsTrue}() ) \\
\{ \\
  \text{P2CPP(block}_s, \ldots, \text{block}_n) \\
  \text{MyNode->Statement();} \\
\} \\
```
<table>
<thead>
<tr>
<th>Block</th>
<th>Description</th>
</tr>
</thead>
</table>
| block$_{i_1,...,i_j}$ abort block$_{i_1,...,i_j}$ | Ret = MyNode->Abort(this,  
  IF (INPUTPARAM(block$_{i_1,...,i_j}$, i, NOEXPR),  
  IF (RETPARAM(block$_{i_1,...,i_j}$, i, NORET),  
  IF (LOCALVAR(block$_{i_1,...,i_j}$, i, NOVAR),  
  (ProcessBlock)&processClass_::method_Block::s 1 2) ; |
| block$_{i_1,...,i_j}$ interrupt block$_{i_1,...,i_j}$ | Ret = MyNode->Interrupt(this,  
  IF (INPUTPARAM(block$_{i_1,...,i_j}$, i, NOEXPR),  
  IF (RETPARAM(block$_{i_1,...,i_j}$, i, NORET),  
  IF (LOCALVAR(block$_{i_1,...,i_j}$, i, NOVAR),  
  (ProcessBlock)&processClass_::method_Block::s 1 2) ; |
| delay E | MyNode->Delay(P2CPP(E)) ; |
| skip | MyNode->Statement(); |

**P2CPP (P)**

<table>
<thead>
<tr>
<th>P</th>
<th>P2CPP (P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>instVar</td>
<td>self(&quot;instVar&quot;)</td>
</tr>
<tr>
<td>inputParam</td>
<td>i(&quot;inputParam&quot;)</td>
</tr>
<tr>
<td>retParam</td>
<td>r(&quot;retParam&quot;)</td>
</tr>
<tr>
<td>localVar</td>
<td>l(&quot;localVar&quot;)</td>
</tr>
<tr>
<td>instParam</td>
<td>self(&quot;instParam&quot;)</td>
</tr>
</tbody>
</table>
E  Smalltalk Classes

The Smalltalk source code of all classes used by the POOSL-compiler (including the modified T-gen classes) can be found in the directory POOSL-compiler within the SHE-directory. The instance variables of the classes are stated between round brackets behind the class name.

E.1 Parse Tree Node Classes

Object ()
TreeNode ()
ParseTreeNode ()

POOSLParseTreeNode ('interval' 'parent' 'level')

SystemSpecNode ('behaviourSpec' 'definitions' 'instances' 'hiddenConnectionDict')
BehaviourSpecNode ('behaviourSpecs' 'channelManipulations')
BehaviourSpecInstNode ('className' 'parms' 'channelManipulations')
BehaviourHidingListNode ('channels')
BehaviourRelabelListNode ('relabells')
ClassDefinitionsNode ('classDefinitions')
ClusterClassNode ('className' 'parms' 'channels' 'messageInterface' 'behaviourSpec'
'systemName' 'filename' 'instances' 'connectionDict' 'hiddenConnectionDict')
ProcessClassNode ('className' 'parms' 'instVars' 'channels' 'messageInterface'
'initMethodName' 'initMethodParms' 'instMethods' 'systemName' 'filename')
DataClassNode ('className' 'instVars' 'instMethods' 'filename')
DeclarationNode ('varNames' 'dataClassName')
MessageProtoNode ('channelName' 'comptype' 'messageName' 'parmPrototypes')

POOSLExpressionNode ()

DataExpNode ('dataExp1' 'operator' 'dataExp2' 'varRef' 'blockNr')
DataExpressionNode ('expression' 'dataClasses')
DataFactorNode ('sign' 'value' 'dataMethodCalls' 'varRef' 'blockNr')
DataMethodCallNode ('name' 'parms')
DataPrimitiveNilNode ()
DataPrimitiveBooleanNode ()
DataPrimitiveCharNode ()
DataPrimitiveIntegerNode ()
DataPrimitiveRealNode ()
DataPrimitiveStringNode ()
DataVariableNode ('name' 'type' 'scope')

POOSSMethodNode ('number')

POOSSDataMethodNode ()

DataAssignNode ('var' 'dataExp')
DataIfNode ('condition' 'thenStatements' 'elseStatements')
DataMethodCallsOnValNode ('value' 'dataMethodCalls')
DataMethodNode ('name' 'parms' 'returnType' 'localVars' 'behaviour'
'dataClasses')
DataPrimitiveNode ('string')
DataReturnNode ('value')
DataStatementsNode ('statements')
DataWhileNode ('condition' 'statements')
POOSL}\text{ProcessMethodNode}'(\text{varRef}'\text{blockNr}')\n\text{ProcessAbortNode}'(\text{statements}'\text{abortStatements}')\n\text{ProcessAssignNode}'(\text{var}'\text{dataExp}')\n\text{ProcessBroadcastNode}'(\text{channelName}'\text{eventName}'\text{sendParms}')\n\text{ProcessDataMethodCallsOnVarNode}'(\text{var}'\text{dataMethodCalls}')\n\text{ProcessDelayNode}'(\text{delayExp}')\n\text{ProcessGuardedCmdNode}'(\text{guard}'\text{statements}')\n\text{ProcessIfNode}'(\text{condition}'\text{thenStatements}'\text{elseStatements}')\n\text{ProcessInterruptNode}'(\text{statements}'\text{interruptStatements}')\n\text{ProcessMethodCallNode}'(\text{name}'\text{parms}'\text{returnParms}'\text{tailRecursive}')\n\text{ProcessMethodNode}'(\text{name}'\text{parms}'\text{returnParms}'\text{localVars}'\text{behaviour}'\text{channels}'\text{messageInterface}'\text{dataClasses}'\text{blocks}')\n\text{ProcessReceiveNode}'(\text{channelName}'\text{eventName}'\text{receiveArgs}'\text{condition}')\n\text{ProcessSelectNode}'(\text{alternatives}')\n\text{ProcessSendNode}'(\text{channelName}'\text{eventName}'\text{sendParms}')\n\text{ProcessSkipNode}()\n\text{ProcessStatementsNode}'(\text{statements}')\n\text{ProcessWhileNode}'(\text{condition}'\text{statements}')
E.2 Other Classes

Object ()

DecorateParm {'parentNode' 'level' 'varRef'}
DecorateClassParm {'instParmDict' 'instVarDict' 'inputParmDict' 'localVarDict'
 'retParmDict' 'clusterParmDict' 'dataClasses' 'blockNr'}
DecorateClusterClassParm {'instanceCollection' 'connectionDict' 'hiddenConnectionDict'
 'relabelDict'}

DecorateMethodParm {'nodeNumber'}
DecoratedDataMethodParm ()
DecorateProcessMethodParm {'channels' 'messageInterface' 'tailRecursion' 'blocks'}

EmitParm ()
EmitCPPParm {'systemName' 'currentClass' 'currentMethod' 'headerStream' 'sourceStream'
 'headerExt' 'sourceExt' 'path' 'varDebug'}

EmitSTParm {'debugFlag'}

VariableReference {'refToInputVar' 'refToReturnVar' 'refToLocalVar'}

Connector {'instance' 'port'}

Collection ()
Set {'tally'}
Dictionary ()
  ConnectionDictionary ()
  HiddenConnectionDictionary ()

Stream ()
  PeekableStream ()
    PositionableStream {'collection' 'position' 'readLimit' 'writeLimit'}
    ExternalStream ()
      BufferedExternalStream {'lineEndCharacter' 'binary' 'lineEndConvention'
       'bufferType' 'ioBuffer' 'ioConnection'}
      ExternalWriteStream ()
      FormattedExternalWriteStream {'indent'}

AbstractScanner {'source' 'nextChar' 'token' 'tokenType' 'buffer' 'interval' 'offset'}
FSABasedScanner {'fsa'}
OptimizedScanner {'finalStateTable'}
OptimizedLookaheadScanner {'savePosition'}
OptimizedScannerWithOneTokenLookahead ()
POOSLParse ()

AbstractParser {'scanner' 'prevToken' 'requestor' 'failBlock'}
TableDrivenParser {'parseTable' 'transcript' 'treeBuilder'}
LALIParse {'finalState' 'intervalStack' 'nrOfIntervals'}
OptimizedLALIParse {'tokenTypeTable'}
POOSLParse ()