RHODES UNIVERSITY

Where leaders learn

HYDRA: A PYTHON EXTENSION FOR PARALLELISM

Submitted in partial fulfilment
of the requirements of the degree of

BACHELOR OF SCIENCE (HONOURS)

of Rhodes University

Waide Barrington Tristram

Grahamstown, South Africa

November 2008

Abstract

Parallel and concurrent programming is a very broad and well researched field. There are numerous models and frameworks for parallel programming, however these frameworks vary in their scope and ease of use. This research investigates the feasibility of developing a CSP to Python translator using a concurrent framework for Python. The objective of this translation framework, developed under the name of Hydra, is to produce a tool that helps programmers implement concurrent software easily using CSP algorithms. This objective was achieved using the ANTLR compiler generator tool, Python Remote Objects and PyCSP. The resulting Hydra framework is able to take an algorithm defined in CSP, parse and convert it to Python and then execute the program using multiple instances of the Python interpreter. Testing revealed that the Hydra framework does indeed function correctly, allowing simultaneous process execution, while introducing negligible overhead. Therefore, it can be concluded that converting CSP to Python using a concurrent framework such as Hydra is both possible and beneficial to the advancement of concurrent software.

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System (1998 version, valid through 2009) [16]:

D.1.3 [Concurrent Programming]: Parallel programming

D.3.2 [Language Classifications]: Concurrent, distributed, and parallel languages

D.3.3 [Language Constructs and Features]: Concurrent programming structures

D.3.4 [Processors]: Code generation, Compilers

General-Terms: Languages, Performance

Acknowledgements

The completion of this thesis is the result of many months of hard work and it would be an injustice to imply that I was not helped along the way. I would therefore like to thank all those who contributed to this project's completion, whether it be in the form of technical assistance, guidance or by providing support throughout the year. There are a number of people that I would to give special thanks to.

Firstly, I would like to acknowledge the financial and technical support of Telkom SA, Business Connexion, Comverse SA, Verso Technologies, Stortech, Tellabs, Amatole, Mars Technologies, Bright Ideas Projects 39 and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

Secondly, I would also like to acknowledge the assistance of the Rhodes University Postgraduate Funding division for the Rhodes University Postgraduate Scholarship extended to me, and the National Research Foundation for the NRF Prestigious Honours Scholarship awarded to me. It must be noted that this research was initiated and performed under my own volition and was not influenced by the above parties.

Thirdly, I would like to thank my parents, Tony Tristram and Deborah Moore, for their continued support and encouragement throughout my academic career. This would not have been possible without their love and guidance. I would also like to thank my step-father, Alan Moore, for accepting me into his family openly and providing his support throughout the year.

I would also like to thank my colleagues in the Department of Computer Science at Rhodes University. Without them, this year would not have been as fun and enjoyable as it was and their encouragement and friendship was much appreciated. In particular, I would like to thank Blake Friedman the input and ideas he offered during our discussions on Hydra.

Finally, I would like to thank my supervisor, Dr. Karen Bradshaw. Without her continued input, support and guidance, this thesis would not have been possible. Her understanding and supportive nature has helped me overcome many obstacles over the course of the year.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel computing is by no means a new topic in the field of computer science. Parallel architectures started making an appearance from as early as the mid-1960s and continue to be the primary design for high performance computing systems. This is particularly evident in modern supercomputers, such as IBM's Roadrunner and Blue Gene/L, which make use of thousands of processors to achieve their astonishing computational power. However, these systems are only available to a select few scientists and researchers and it wasn't until a few years ago that multi-processor computers started becoming readily available to consumers.

The recent availability of dual and quad core CPUs targeted at the consumer and enthusiast market has caused an interesting situation in the software field. Multi-core computers have the power and potential to greatly outperform their single-core counterparts, but this potential can only be realised if the software is able to make use of multiple processors. Both consumers and researchers stand to gain from the performance increases afforded by multi-core CPUs and parallel software. Consumers benefit from faster, more responsive computers that are able to handle computationally intensive tasks, such as decoding and playing high resolution video. Researchers benefit from being able to construct small high performance computing systems for their data processing needs by combining a number of relatively cheap multi-core CPU systems.

While some progress has been made towards developing better concurrent software, there needs to be a shift in software development practices to harness the power of parallel computers on a greater scale. With such a shift in development practices comes the need for tools that enable and assist developers in their task of creating concurrent software. The Hydra project aims to provide such a tool to Python developers.

1.1 Problem Statement and Research Goals

This research project investigates the feasibility of developing a framework for Python that exposes parallel processing features based on Communicating Sequential Processes (CSP). Therefore, a concurrent framework for Python, named Hydra, will be developed to assess the feasibility of converting directly from CSP to Python, without requiring that the application programmer manually convert their algorithm as is the case for existing CSP implementations such as JCSP.

This framework should be able to generate Python code capable of executing in a parallel manner over an arbitrary number processors in a single computer. Furthermore, this framework should make it easy specify the process and communication architecture of the program, with readability and ease of programming being the focus. The details of how the concurrency is implemented should be mostly hidden from the program developer, while maintaining a clear mapping between the programmer's specified architecture and the generated concurrent code. Most importantly, this framework must improve upon the existing concurrent framework by automating the algorithm conversion process.

The secondary aims of this project include making the system flexible and extensible, thereby allowing for additional features to be added and the ability to generate code for alternative architectures, such as Grid computing systems. The performance of the Hydra system should also be maximised without affecting the primary goals.

1.2 Thesis Organisation

The relevant chapters of this thesis are organised as described below:

Chapter 2 introduces and discusses some of the relevant work in the areas of multi-processor computers, parallel processing, Communicating Sequential Processes and Python.

Chapter 3 will describe the approach and methodology adopted for the development of the Hydra system.

Chapter 4 will provide an introduction to language parsing and present an in-depth description of the prototype and final parsers for the Hydra system.

Chapter 5 describes the code generation process and presents the implementations of both the prototype and final code generator modules.

Chapter 6 then describes the testing and performance analysis methodology along with test examples, and goes on to present and discuss the results.

Chapter 7 summarises this thesis and provides conclusions drawn from the research, development and testing of the Hydra project. Future work and extensions to the Hydra project are then discussed, bringing the thesis to an end.

Chapter 2

Background and Related Work

2.1 Introduction

This chapter highlights and discusses some of the work in the area of parallel computing, which is fairly broad, with a vast amount of literature available. As such, this literature review will not attempt to cover all of the concepts and work, but will focus instead on work relevant to the creation of a concurrent framework for Python based on CSP.

The first section defines some of the relevant terminology to aid understanding and clarify how the terms are used in this review. The second section introduces some of the concepts related to parallel programming and multi-processor systems, the current trends and motivation for pursuing development in this field. The third section describes CSP, its strengths and weaknesses and how it can be used in the development of concurrent programs. Some existing CSP implementations are also introduced briefly. The fourth section introduces the Python programming language, discusses the motivations for choosing Python as the host language and some of the issues regarding its use. The sixth section discusses some of the existing concurrent frameworks such as PyCSP. A basic summary of the work is then provided in the final section.

2.2 Key Terminology and Concepts

A number of terms and concepts relating to parallel computing are defined below. These definitions serve to aid in the understanding of the forthcoming work for those who are unfamiliar with common terms used. They also serve to clarify how the author defines and uses these terms if other definitions exist.

Central Processing Unit (CPU). The Central Processing Unit is the primary instruction execution engine of a computer system. It is usually made up of a single integrated circuit, which contains the arithmetic logic unit (ALU), control unit and various caches, registers and memory controllers. It operates on a fetch, decode and execute cycle for instructions stored in memory [29].

Concurrency. Concurrency is defined as any set of tasks or processes that are executing and have the potential to execute simultaneously, but are not necessarily doing so [13].

Multi-core CPU. A multi-core CPU is a CPU that is made up of multiple, separate processor cores placed on the same CPU die. These processor cores are usually able to communicate with each other over either the system bus, crossbar switches or shared memory, such as the on-die cache memory [20].

Multiple Instruction set Multiple Data set (MIMD). A MIMD computer consists of a number of independent CPUs asynchronously executing their own instructions streams on their own data streams [13].

Parallelism. Parallelism is defined as any set of tasks or processes that are actually executing simultaneously [13].

Process. A process is an instance of a task that is actively executing, however this term can be used interchangeably with task [13].

Process Algebra. Process Algebra is an algebraic approach to the study of concurrent processes [45]. Algebraical languages are used for the definition of processes and statements about them [45]. These statements can then be verified through the use of the appropriate process calculi [45].

Processor. The processor is the piece of hardware on which processes execute [13]. A more detailed definition can be seen above in the definition of a CPU.

Task. A task consists of a single operation or multiple operations that are to be executed [13].

2.3 Multi-Processor Systems and Parallel Programming

Many people have access to parallel computers, but only a small percentage of programmers develop software that runs on parallel capable computers. There are very few people with the knowledge, tools and experience to leverage the processing power provided by these parallel processors [22, 42]. The current trend towards developing multi-core and multi-CPU systems instead of increased clock speed is likely to make this distinction even more important in the future [20, 42].

In the past, semiconductor firms such as Intel and AMD were able to increase performance by increasing clock speeds for single-core CPUs. This was as a result of being able to shrink the manufacturing process for the transistors and CPUs. In 1974, Robert H. Dennard, and his colleagues at IBM, observed that as the size of transistors decreased, so did the voltage and current requirements [20]. This scaling law means that as the transistors decrease in size, more of them can be packed closer together on the chip, while maintaining the same power density. This leads to the trend of doubling the CPU transistor counts and speed every 18 months, which is widely known as Moore's Law [20, 42]. However, these scaling laws do not provide a perfectly linear decrease in power requirements, and as such, process shrinks are becoming less effective at yielding increased speed [20].

This has led CPU manufacturers to the new strategy of placing more than one CPU core on a processor chip instead of constantly increasing clock speeds [42]. This is evident from the abundance of dual-core CPUs, the introduction of quad-core CPUs and Intel and AMD's plans for eight-core CPUs in late 2009, while firms such as Sun Microsystems already have 16 core processors [11, 20]. The only issue now is whether or not these new processors can be used to their full potential by current software development techniques.

The most significant obstacle to developing concurrent software and making effective use of multi-core CPUs is that the designing, writing and debugging of concurrent code is fairly difficult [43]. Truly parallel programs are rarely written, despite the existence of trivially parallel tasks and the publishing of numerous parallel algorithms. Multi-core CPUs fit in the *MIMD (Multiple Instruction set Multiple Data set)* category of parallel computers and are thus suited to the message-passing model of parallel computing [19, 20, 22]. Implementations such as CSP and *MPI (Message Passing Interface)* follow the message-passing model [22]. It is therefore necessary to provide programmers with the tools and knowledge, based on the above models, so that they can use the multi-processor systems available today effectively [43].

2.4 Communicating Sequential Processes

Communicating Sequential Processes was first introduced in 1978 by Hoare. In his paper [23], Hoare identified a number of operations and constructs as the primary methods for structuring computer programs. He identified *input* and *output* operations as being important but noted that these were not well understood. He also noted that the *repetitive*, *alternative* and *sequential* constructs were well understood, whereas there was less agreement on other constructs such as *subroutines*, *monitors*, *procedures*, *processes* and *classes* [23].

Processor development at the time was such that multiprocessor systems and increased paral-

lelism was required to improve computation-speed. However, Hoare noted that this parallelism was being hidden from the programmer as a deterministic, sequential machine, effectively trying to make a multiprocessor machine appear as a mono-processor machine. He saw that a more effective way of making use of these multiple processors, would be to introduce this parallelism at the programming level by defining *communication* and *synchronization* methods [23]. As a result, Hoare developed the CSP programming notation.

2.4.1 The CSP Programming Notation

The programming language or notation specified by Hoare is based on a number of fundamental proposals. The first of these is the use of the *alternative* command in conjunction with *guarded commands* as a sequential control structure and a means to control non-determinism. The guarded command will execute its command list sequentially only when its *guard* succeeds and the alternative command will select only one ready guard command at a time and terminate when all of its guards fail. Associated with the guarded and alternative commands is the *repetitive* command, which loops until all its guards terminate. Secondly, the *parallel* command specifies a means to start parallel execution of a number of processes or commands, by starting them simultaneously, and synchronizing on termination of each of the parallel processes. Parallel processes may not communicate directly, except through the use of *message passing* [23].

```
assign ::
  x :  integer;
  x := 5;
```

Figure 2.1: A CSP Process with simple declaration and assignment.

To support the message passing concept, *input* and *output* commands are specified. These commands enable communication between processes. Essentially, a *channel* is created and used for communication when a source process names a destination process for output and the destination process names the source process for input. Communication only occurs once both the source and destination are ready and results in the value being copied from the source to the destination process. If either of the two processes is not ready for communication, the command will wait until such a time as both are ready. This effectively introduces the *rendezvous* as the primary method of synchronization [23].

Input commands may be used as guards and result in the command only being executed when the other process is ready to execute its output command. If multiple input guards are ready, an arbitrary choice is made and only the selected command will execute with no effect on the

```
-- Process src runs in parallel with process dest
[ src ::  dest !  5;  -- output the value 5 to dest
||
  dest ::
    x :  integer;
    src ?  x;  -- read input from src into x
]
```

Figure 2.2: The Parallel, Input and Output commands with comments.

other input guards. The final proposal includes the screening of input messages by way of *input pattern matching* to ensure that the input message follows the correct pattern [23].

```
[ buf ::
    buffer :  [0 ..  9] integer;
    in, out :  integer;
    in := 0; out := 0;
    *[
       in < out+10; producer ?  buffer[in%10] -> in := in + 1
     [] out < in; consumer !  buffer[out%10] -> out := out + 1
    ]
||
  producer ::
    x :  integer; x := 1;
    *[ x <= 100; -> buf !  x; x := x + 1 ]
||
  consumer ::
    item :  integer;
    *[ true -> buf ?  item ]
]
```

Figure 2.3: The Repetitive and Guarded commands with the use of the Input command as a Guard.

While Hoare indicated that programs expressed in this notation should be implementable, he also made it clear that the notation was not suitable for use as a programming language. This was in light of the fact that there were serious issues that had been overlooked. These included the fairly static nature of CSP programs, which could only have a fixed number of concurrent processes, and the lack of recursion. The issue of performance had also been overlooked. But the most serious issue was the lack of a proof method to verify the correctness of programs [23].

2.4.2 The CSP Meta-Language

After the publication of the initial CSP paper in 1978, Hoare continued to refine CSP and in 1985, he released a book on the CSP notation [24]. The CSP described in the book has evolved substantially from the notation described in his earlier paper. CSP has moved from being a programming notation to being a *process algebra* that allows for the formal description and verification of interactions in a concurrent system. The new notation consists of two primitives, namely the *process* and the *event*, and a number of algebraic operators. Concurrent and sequential systems can then be defined through a combination of these operators and primitives.

An important addition to CSP is the introduction of *traces*, which allows for the description of each possible behaviour in a system as a sequence of actions. The combination of the formal description of the system and the traces allows for the analysis and verification of a system's possible behaviours. In the book, Hoare describes methods for expressing and verifying a number of important concepts, namely, processes, concurrency, non-determinism, sequential processes and communication [24].

Use of these description and verification techniques makes it possible for one to check for the absence of undesirable conditions such as *deadlock, live-lock* and *starvation*. There are a number of tools that have been developed to aid in the verification of systems based on CSP and its proof methods. CSP has also seen use in the verification of large systems such as the Certification Authority for the Multos smart card scheme, developed by Praxis Critical Systems [18].

2.4.3 CSP Implementations

There are a number of programming language implementations based on or around CSP. The most notable of these implementations is *Occam*, which was developed closely around CSP by David May in collaboration with Tony Hoare. Occam is a minimalist language developed at INMOS for use in their transputer devices [26]. There are also a number of implementations for modern programming languages, such as *JCSP* and *CTJ* for Java, *CCSP* for *C*, *CSP.NET* for *Microsoft .NET 2.0* and *PyCSP* for Python [2, 8, 39]. Each of these implementations has its own strengths and their weaknesses, which are often influenced by the target language.

2.5 The Python Programming Language

Python is a powerful, very high level programming language. It is a multi-paradigm programming language supporting the functional, object orientated and procedural programming

paradigms. Python has a strong, dynamic typing system, which features *"duck typing"*. It also has a robust automatic memory management system. It is very well suited to use both as a scripting language, much like *Perl*, and as a general purpose programming language. Python places a great deal of emphasis on programmer productivity and supports this via its expansive standard library and its minimalist syntax, which enhances code readability. The major implementations of Python are *CPython*, *Jython*, *IronPython* and *PyPy* [28, 37].

2.5.1 Features and Benefits

There are numerous benefits and features that make Python a very attractive language for both beginner programmers and advanced scientific programming. The most notable of these are mentioned below. It has very high-level built-in data types, such as the dictionary, list and tuple. The syntax is very clear with a focus on readability and it supports the natural expression of procedural code. Python has strong introspection capabilities and provides easy to use object orientation features. Other benefits include fast, exception-based error handling, extensive standard libraries and third-party modules that provide support for most programming tasks. There is also plenty of support and readily available documentation. Free access to the source code makes it possible to modify Python if required [4, 31, 37].

Python's extensibility provides even greater power over its existing functionality via its support for full modularity and hierarchical packages. Modules and extensions can be written in Python or alternatively, they can easily be written in C, C++, Java (for Jython) or .NET languages (for IronPython). Python can also be embedded within applications as a scripting interface. This makes it very useful for linking together previously unrelated modules. Python can therefore be used for quickly prototyping of algorithms, with any performance critical modules being rewritten in C and added as extensions. All of the above factors have aided in the acceptance of Python in the computational science community [4, 25, 31, 37].

2.5.2 Limitations

As an interpreted language, Python's performance is not as good as compiled languages such as C++, but the performance is more than sufficient for most applications. If improved performance is required, the *Psyco* and PyPy projects, which provide optimised *Just-in-Time (JIT)* compilers for Python, can produce typical speedups of around 4x [35, 40]. However, Python's greatest limitation is its global interpreter lock. "A global interpreter lock (GIL) is used internally to ensure that only one thread runs in the Python VM at a time." [38]. So, while Python

supports multi-threading, these threads are time-sliced instead of executing in a truly parallel fashion.

In one of the earlier versions of Python, an attempt was made to remove the GIL and replace it with fine-grained locking. This was achieved through Greg Stein's *"free threading"* patches. However, there were serious performance issues associated with the new locking mechanism, even with efficient system level locks. This resulted in a halving of the performance or worse, which was not acceptable, especially for applications that did not make use of threads. The patches were later abandoned and no further significant attempts were made to address the issue [38].

There are two ways in which the GIL can be circumvented to achieve multiple CPU usage. The first of the suggested methods is to make use of C extensions, whereby a C extension is used to perform the required task. The extension can then release the GIL and maintain the executing thread within the C code. The second of the suggested methods is to divide the tasks between multiple Python processes as opposed to threads within a single Python process. This entails spawning multiple Python interpreter processes and maintaining efficient communication and synchronization between the processes [38].

Several projects already cater for the use of multiple interpreter processes. *IPython* is an enhanced interactive Python shell that provides the underlying connection architecture between interpreters for parallel computing [36]. The *River* framework for distributed computing is another relevant project that provides the fundamental abstractions for flexible communication management between multiple Python VMs (Virtual Machines) and execution of code on these VMs [6, 7]. *Trickle* is implemented on top of the River framework and provides a simple implementation of the MIMD model of parallel computing. Essentially, Trickle provides methods for injecting code and data into remote VMs, accessing remote objects and asynchronous method invocation. It also provides simple mechanisms for dynamic scheduling and balancing of work between the VMs [6].

2.6 Existing Concurrent Frameworks

While there are many projects that add CSP features to existing programming languages, there are very few attempts to convert directly from CSP to executable code [39]. JCSP and CTJ provide CSP features to Java [5, 21]. CCSP and C++CSP provide similar CSP features for C and C++, respectively [2, 12]. PyCSP is of great interest to this project as it introduces CSP features to Python [8]. From the list of modern language CSP implementations mentioned above, it would appear that no further work is required to expose CSP to programmers. However, these

implementations require the programmer to convert their CSP code into the appropriate form for the implementation they desire to use. For small programs, this task is relatively easy. But once the programs start to get bigger and more complex, the process becomes more difficult and is prone to error, particularly with regards to the correct naming and use of channels [39]. The time taken to develop and verify the CSP algorithm for a complex system can often be rivaled by the time taken to convert and debug the program written for one of the above mentioned CSP implementations [39]. Clearly this is not ideal and a means for translating the original CSP directly to executable code is more desirable.

2.6.1 Translation to an Intermediate CSP Implementation

The most notable work in the area of translating CSP to executable code is a set of tools developed for converting CSP_M to CTJ, JCSP and CCSP [39]. While this may not be a direct translation to executable code, this method of using an intermediary CSP implementation to produce programs based on a CSP algorithm is quite effective at eliminating the probability of errors and speeding up development [39]. The translation process is best visualised through the use of *T-diagrams*. The conversion from CSP to JCSP can be seen in Fig. 2.4, the conversion to CTJ is shown in Fig. 2.5 and the conversion to CCSP can be seen in Fig. 2.6.

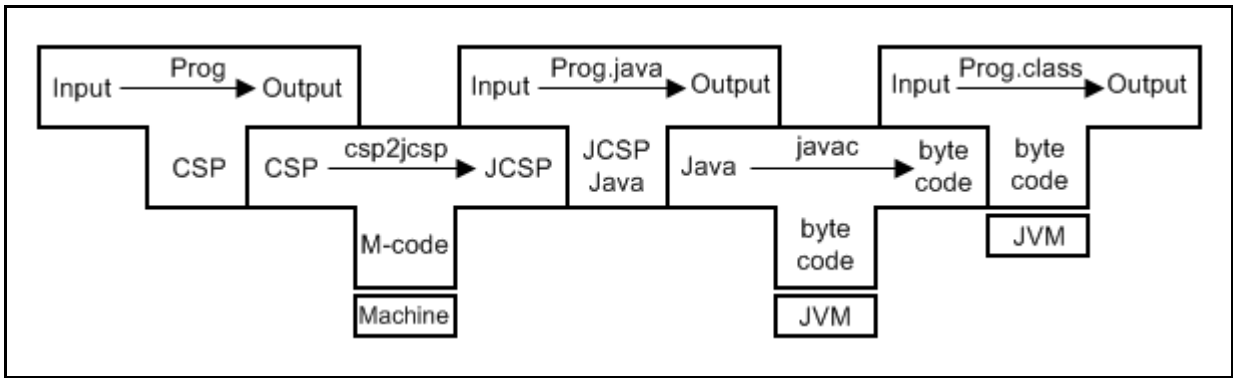


Figure 2.4: T-diagram showing the translation of CSP to Java byte code using JCSP [39].

The developers of the above mentioned conversion tools chose to implement the translators in C++. Their reasoning for this is the amount of string processing required and the need for powerful, dynamic lists. These requirements are satisfied by the *String* and *Vector* classes available in the *Standard Template Library (STL)* [39]. However, as discussed earlier, Python's powerful standard libraries, good string handling capabilities and its built-in list data type make it very suitable for the task of translating CSP.

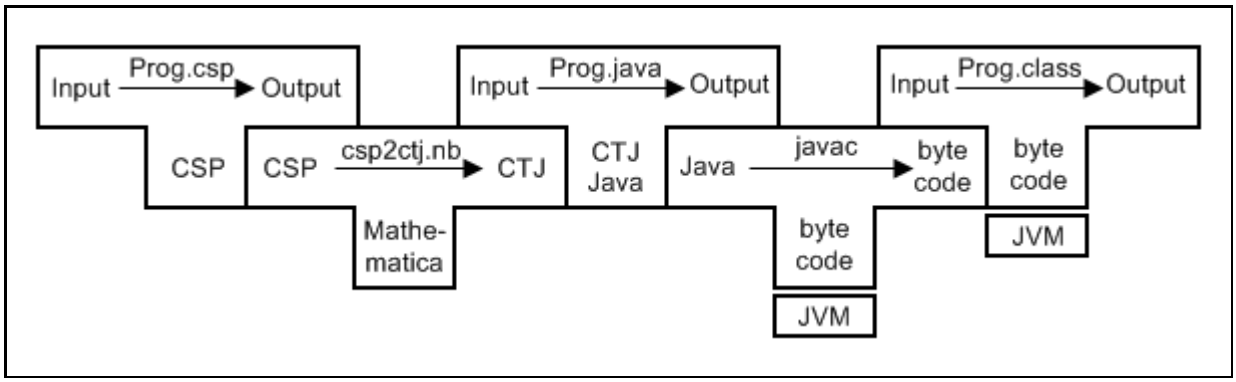


Figure 2.5: T-diagram showing the translation of CSP to Java byte code using CTJ [39].

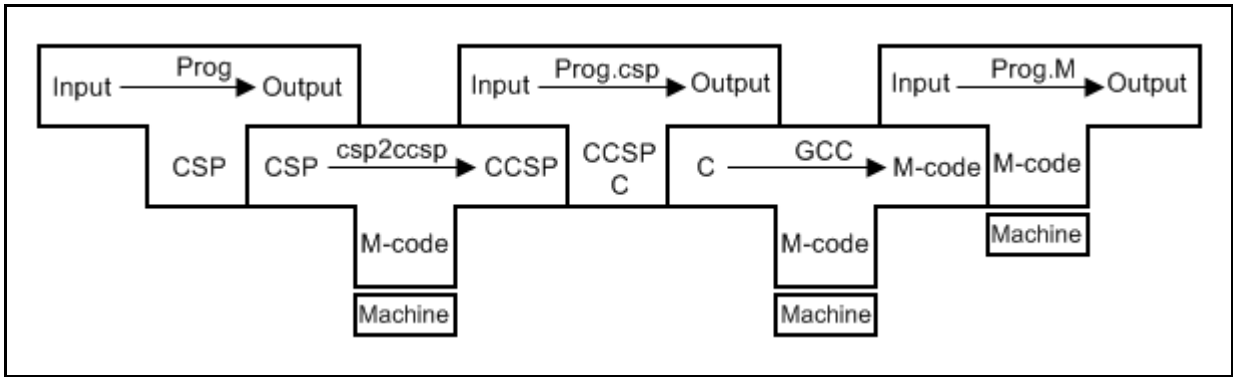


Figure 2.6: T-diagram showing the translation of CSP to executable machine code using CCSP [39].

2.6.2 Use of a Python-based CSP Implementation

PyCSP is another relevant module to investigate as it provides Python with a number of CSP constructs such as channels, channel poisoning, basic guards, skip guards, input guards, processes, and the alternative, parallel and sequential constructs [8]. However, as with the previously mentioned CSP implementations, PyCSP leaves the programmer with the task of translating the CSP algorithm to Python code. While the PyCSP library is promising in its own right, it is not without its downsides. A translation tool, similar to the one mentioned above, that takes CSP and converts it to PyCSP based Python code, or native Python code, would be more productive and less error-prone [8, 39]. This would free the programmer from having to deal with complex channel naming situations and having to correctly handle the undesirable aspect of channel poisoning.

A further step from translating to an intermediary CSP implementation, such as PyCSP, would be to translate directly to the parallel code. This could be done using the underlying implementation of a CSP library like PyCSP as a guide, which in turn uses the JCSP implementation as its guide [8]. The biggest drawback of PyCSP is that its current implementation makes use

of Python's threading library, which limits the parallel execution of code because of the GIL [8, 38]. The suggested solution to this problem is to make use of network channels for communication between multiple local or remote operating system processes [8, 38].

PyCSP's implementation of synchronization is achieved through the use of Python's *decorator* construct [8]. Decorators are used to provide wrappers around methods that require synchronization, with the wrapper function handling the acquisition and release of *locks* using the *with* keyword [8]. The PyCSP *Process* construct is implemented by simply instantiating an object of the `Process` class, which extends from Python's `Thread` class [8]. The constructor for the `Process` class takes the function to be executed by the Process and the list of arguments for the function [8]. This method of implementing Processes would need to be modified to use network channels to allow for truly parallel execution. The instantiated Process object does not begin execution until it is used in either the *Parallel* or *Sequential* constructs [8].

The Parallel and Sequential constructs have a very simple implementation in PyCSP. The Parallel construct is implemented by a class that takes a list of Processes to be run in parallel, and calls `start()` for each Process to begin execution and then calls the `join()` method for each Process to synchronise and terminate the parallel execution [8]. The Sequential construct is implemented as a class that takes a list of Processes to be run in sequence and calls the `run()` method for each Process, thus executing them in the specified sequence without synchronization [8].

Communication via *Channels* is handled by simply passing the read and write methods of a Channel object directly as arguments in a Process's constructor [8]. Python allows for this kind of functionality which is very useful because it helps avoid using the incorrect end of a Channel [8]. PyCSP Channels allow for any object to be sent across them, including Processes [8]. This is a useful feature that allows for easy distribution of work and instructions, as well as the removal of type limitations present in other CSP implementations [2, 5, 8, 21, 39].

Finally, the PyCSP *Alternative* construct is implemented as a class that takes a list of *Guards* in its constructor. The next active Guard is then selected in a JCSP-like fashion by using the `priSelect()` method, which returns each active Guard in turn [8]. Python's ability to return a reference to an object allows PyCSP to improve upon JCSP by returning a reference to the active Guard as opposed to an index, which then has to be analysed to determine the correct code segment to execute [5, 8]. Guards can be simple objects that extend from the `Guard` base-class, Channel inputs or the special *Skip Guard* [8]. The lack of an explicit *Repetitive* construct hints at an implementation that simply uses a natural Python loop over an Alternative construct.

2.7 Summary

Research into the area of parallel computing has shown that parallel computers are no longer confined to the scientific research community. It has been seen that certain factors relating to the design and manufacturing of modern CPUs have led to a new trend of increasing the number of processor cores on the CPU instead of simply increasing clockspeed to improve performance. This has resulted in affordable and readily available parallel computers for the desktop.

It has also been seen that while parallel computers are becoming increasingly abundant, the software and development tools are lacking when it comes to harnessing the performance of parallel computers. The author has noted that there needs to be a shift in software development practices towards parallel programming models, as well as an increase in the number of tools designed to aid in the development of parallel software.

Based on this insight, an appropriate parallel programming model and programming language were investigated for the development of such a tool. It has been seen that the CSP model, coupled with the Python programming language, provides a good base for the creation of a message passing based concurrent software development framework. It is also evident that modules such as PyCSP can be improved upon and used as a guide for the creation of such a framework.

Chapter 3

Methodology

3.1 Introduction

As stated in Chapter 1, the aim of the project is to investigate the feasibility and development of a concurrent framework for Python based on CSP's message-passing model. However, due to the scope of such a development project and the time available, the initial objectives of this project have been restrained such that the goal is the creation of a demonstration prototype framework as opposed to a complete framework suitable for public distribution. This scope refinement has some implications on the approach and methodology used for the project.

3.2 Approach

Development of the Hydra framework is performed in two phases. The first phase involves the development of a hand-coded recursive-descent parser for a cut-down version of the CSP grammar. The resulting abstract syntax tree (AST) generated by the parser is used to generate and run JCSP Java code. The objective of this phase is to gain familiarity with the issues involved in parsing CSP and generating executable code. Since the aim is centered around experimentation and prototyping, semantic checks are excluded and error reporting is minimal if present. Once the code generation is complete, the JCSP code output is visually inspected and then executed to determine if the translation was successful. Once the JCSP prototype is deemed to produce acceptable output code, the project advances to phase two.

The second phase centers around developing a flexible and extend-able parser and code generator that can be used, firstly, as a proof of concept, and secondly as a base for further development of the Hydra project. The parsing aspect of this phase involves selecting an appropriate compiler

generator, refining the CSP grammar for easier parsing and then implementing the grammar for the chosen compiler generator. Even though this phase has a greater focus on usability and functionality, it is still a demonstration-only prototype. As such, semantic checks and error reporting, while better than the JCSP prototype, are minimal and incomplete.

The code generation aspect requires identifying suitable libraries and frameworks on which to build the parallel constructs and then designing the necessary code segments to represent the CSP constructs, using the selected underlying libraries. Once the concurrent framework is complete, the actual code generation takes place. This involves multiple passes over the AST to generate the concurrent code using the above-mentioned code segments. Again, the output code is visually inspected and executed to assess its correctness. Performance analysis and further testing is only carried out once the code generator is able to produce consistently correct concurrent code.

3.3 Summary

The objectives of this project focus on investigating the feasibility of developing a concurrent framework for Python. Therefore an approach based on prototyping and experimentation has been chosen to explore the topic with greater freedom. This has resulted in a number of compromises in the development of the framework, such as the exclusion of certain features and reduced error checking. A two-phased approach has been adopted involving the development of a hand-coded JCSP prototype and then a more functional prototype based on a custom-built concurrent framework.

Chapter 4

Parsing CSP

4.1 Introduction

To generate the desired concurrent code, it is first necessary to parse and interpret the CSP source code supplied by the application programmer. The development of a parser for the CSP grammar is thus required. The construction of the Hydra parser was achieved in two phases. The first phase involved hand-crafting a prototype scanner and parser for a limited version of the CSP grammar. The purpose of this first phase was assessing the viability of parsing and converting CSP, as well as to gain familiarity with the semantics of the CSP language. The second phase involved the construction of a complete, maintainable and powerful parser for the 1978 version of the CSP notation [23] using the ANTLR compiler generator [33].

As noted in Chapter 2, the grammar for CSP described in 1978 [23] is not suitable as a programming language by itself. For this reason, a number of compromises and changes have been made to the grammar to allow for greater integration with Python. CSP is essentially used to define the architecture and communication channels of the processes used by the program. These grammar modifications are described in Section 4.6 along with the actual construction of the ANTLR-based parser.

4.2 Basic Parser Construction

A number of techniques exist for parsing code from a source language and translating it to a target language. The typical translation process involves a number of stages, which are usually divided into a *front-end* parser, and a *back-end* compiler or interpreter, as seen in Fig. 4.1. The parser consists of a *lexical analyser*, which takes the *source code* string as input and outputs

the identified tokens or symbols. The second stage in the parser is the *syntax analyser*, which takes the tokens from the lexical analyser and evaluates them against the grammar of the source language to identify the statements and expressions [44].

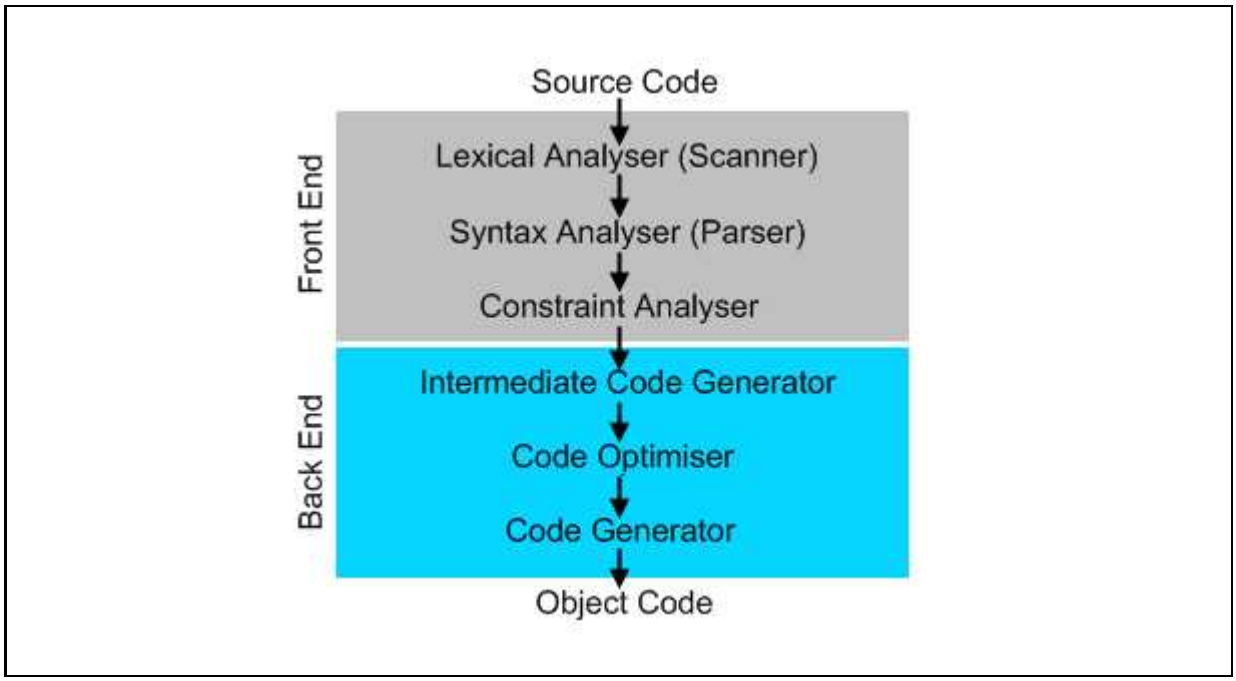


Figure 4.1: Structure and Phases of a Compiler [44].

The output from the syntax analyser is often passed through a *constraint analyser*, which checks that the syntactic components adhere to the scope and type rules applicable to the current context. The results from the constraint analysis are then passed into the back-end where they are converted into intermediate code by the *intermediate code generator*. This code is often sent through a *code optimiser* before being converted to *object code* by the *code generator* [44].

4.3 Types of Parsers

As stated above, there are a number of parsing techniques, each of which is suited to certain types of grammars. The *top-down* parsing technique makes use of *LL (Left to right, Leftmost derivation)* parsing, with *recursive-descent parsers* being a fairly common form of *LL* parser [33]. The top-down technique involves starting from a specific goal production and working its way down, identifying appropriate lower level productions as it descends, until it has identified all the productions associated with the input tokens [44]. Recursive-decent parsers can usually

be constructed by hand, but are typically limited to a subset of context free grammars, specifically those that are *LL(1)* compliant. There is a special form of *LL(k)* parsing, known as *LL(1)* parsing, which uses only one look-ahead token and has the benefit of certain optimisations, but has the downside of being more restrictive than other *LL(k)* parsing techniques [44].

LL(k) parsers are slightly more flexible than hand-coded recursive-descent *LL(1)* parsers. The *k*-value represents the number of look-ahead tokens, allowing the parser to access tokens further down the input stream to make decisions regarding how to match a production rule [33, 44]. The greater the look-ahead, the more flexible the parser is, thus allowing more complex grammars to be parsed [33]. However, constructing *LL(k)* parsers with a look-ahead greater than 1 is usually harder and not easily achieved by hand [33]. As such, a tool known as a compiler generator or compiler compiler is used to produce these parsers [44]. Compiler generators usually accept the target grammar, which is specified using an appropriate syntax, and generate the scanner and parser components of the compiler [44]. Another important form of *LL(k)* parser is the *LL(*)* parser. The *LL(*)* parser has an arbitrary look-ahead and often makes use of backtracking to help it in the identification of production rules [33]. This arbitrary look-ahead allows the parser to evaluate a number of alternatives further down the input stream, making it an extremely powerful parsing technique capable of handling an even greater range of grammars [33]. Some examples of currently available compiler tools using these techniques are *Coco/R*, *ANTLR*, *Yapps* and *Parsec*.

The *bottom-up* class of parsing techniques attempt to work from the tokens up and identify the appropriate top-level productions. These parsers are usually known as *LR (Left to right, Rightmost derivation)* parsers. They can be constructed using a recursive-ascent parsing technique, with a set of mutually-recursive functions. Two forms of this parser are the *SLR (Simple LR)* parser, which has no look-ahead and is thus limited to simpler grammars, and the *LALR (Look-ahead LR)*, which allows for look-ahead and can thus parse more complex context free grammars [1]. The most notable tools for generating *LR* parsers are *lex/yacc*, *Parsing*, *Wisent*, *PLY* and *Bison*.

4.4 Prototype Parser Implementation

Previous attempts have been made to generate parsers for CSP. Firstly, a parser for a dialect of CSP, known as CSP_M for CSP machine-readable, was written using *Flex* and *Bison*. However, it was necessary for some productions to be rewritten to remove ambiguity [41]. Flex and Bison grammars for CSP_M are listed in [41]. Bison is an *LR* parser and as such, it would appear that it is possible to parse CSP, or at least a specific dialect of CSP, using an *LR* parser. However,

CSP_M follows the later version of CSP described in [24], which is more of a process algebra than a programming notation. A parser for *CSP-CASL*, which is the process algebra of CSP integrated with the algebraic specification language *CASL*, was developed using Parsec, which is a monadic recursive-descent parser written in *Haskell*. As with CSP_M, certain productions in the grammar for CSP-CASL had to be rewritten slightly to remove left recursion. This shows that it is possible to use the recursive-descent parsing technique [17].

With the above findings in mind, it was decided that a recursive-descent compiler would be developed, based on the notation described in [23]. The aim of this exercise was to gain familiarity with the CSP notation and its semantics, as well as evaluate any potential pitfalls and considerations that may need to be taken into account when converting to the target concurrent code.

4.4.1 Prototype Design

The design of the prototype system makes use of Python's object-orientated features to ensure ease of modification and maintenance. The prototype Hydra implementation consists of a scanner module (`Hydra.scanner`), a parser module (`Hydra.csp`), an abstract syntax tree module (`Hydra.ast`) and a code generator module (`Hydra.codegenerator`). The csp execution method in `Hydra.csp` is defined in such a way that it allows for easy switching between different parsers, scanners and code generators. The parser module takes the CSP code as input and passes it to the scanner. The parser module then instantiates the appropriate parser class, passing it the desired scanner and code generator objects constructor arguments. The parser class begins by calling the goal method and follows a recursive-descent parsing technique, requesting tokens from the scanner and generating an abstract syntax tree (AST) as it parses the input. Once parsing is complete, the AST is serialised using Python's `pickle` module. The pickled AST is then sent to the code generator module, which produces the appropriate output code based on the AST. Figure 4.2 provides an overview of the prototype system design.

Before describing the scanner and parser implementations, it is necessary to describe the manner in which CSP is used within a Python program. The mechanism chosen is fairly simple, but effective and straightforward. First the Hydra csp module must be imported. Then, the CSP algorithm is defined within a triple-quoted string. This string is then passed to the `cspexec` method of the Hydra csp module, along with any helper functions and external variables. A simple example can be seen in Fig. 4.3 below.

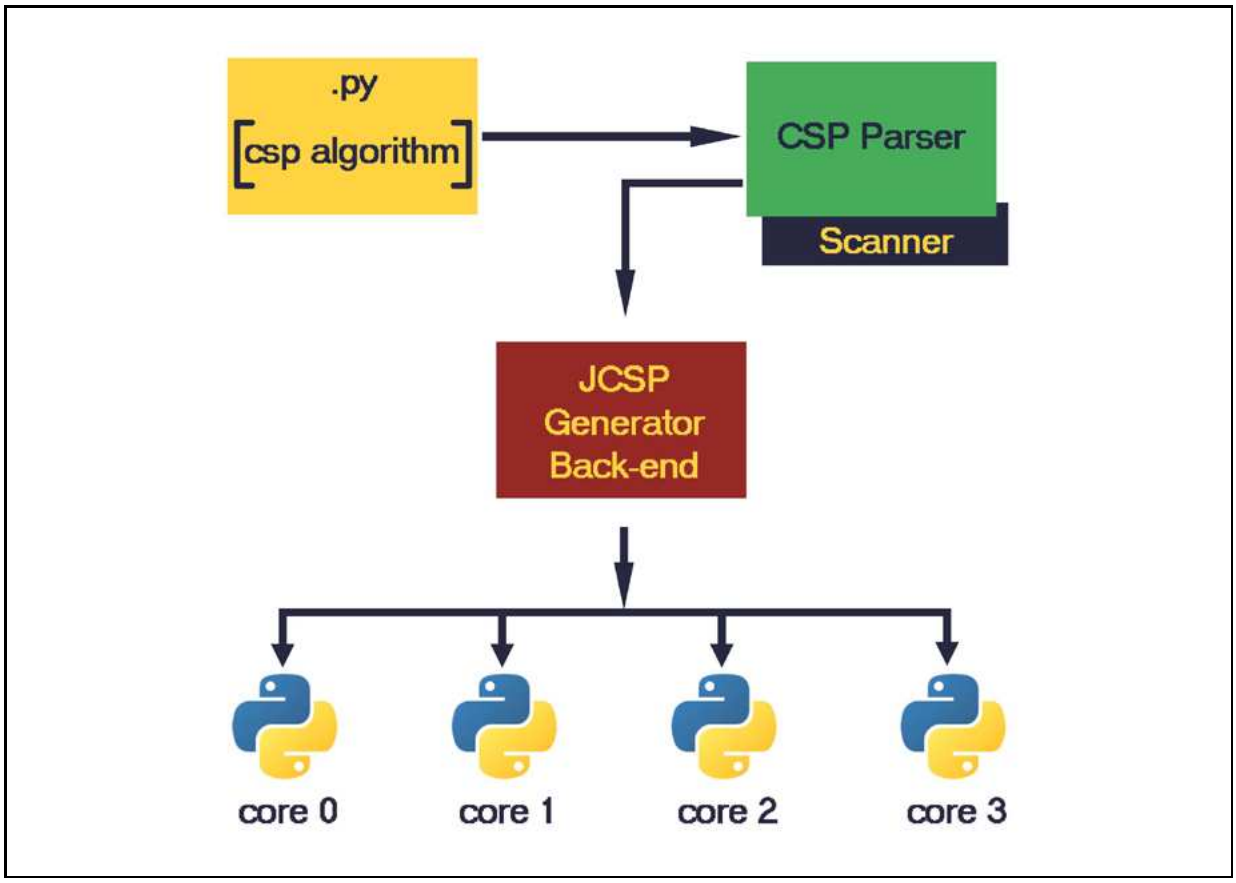


Figure 4.2: Prototype system design.

4.4.2 Hand-crafted CSP Scanner

The scanner or lexical analyser is responsible for reading the input stream one character at a time and identifying the tokens specified by the grammar [44]. The prototype scanner was implemented in pure Python and is fairly simple in its design. The Scanner class consists of a list of accepted symbols or tokens, a `getChar` method and a `getSym` method. The tokens represent the lexical elements of the grammar, such as identifiers, constants and the various punctuation marks used for structure. The `getChar` method simply advances the position in the input string by one character and sets the current character to the character from this new position. The `getSym` method is the bulk of the scanner and is responsible for identifying the different symbols based on the characters in the input stream, which are retrieved using the `getChar` method.

The `getChar` method, seen in Fig. 4.4, simply checks if the end of the input has been reached and sets the current character to an end-of-file token if that is the case, otherwise it sets the current character to the next character in the input and increments the index.

The `getSym` method is somewhat more involved as it has to make decisions based on the input

```
from Hydra.csp import cspexec

code = """[
  prod ::
    data :  integer;
    data := 4;
]
"""
cspexec(code)
```

Figure 4.3: Specifying and executing CSP within a Python program.

```
def getChar(self):
    if (self.index >= self.codelen):
        self.ch = EOF
    else:
        self.ch = self.input[self.index]
        self.index += 1
```

Figure 4.4: `getChar` method of the prototype scanner.

characters as to what symbol has just been read. This is usually achieved by making use of a large `switch` statement, but Python does not have a `switch` statement like other languages such as C++ and Java. However, Python's `if` and `elif` statements are capable of achieving the same result.

The `getSym` method starts by removing whitespace and comments. It then attempts to identify constants and identifiers. If the character does not correspond to either of these symbols, it makes use of a large if-elif decision structure to identify the appropriate symbol. Once the symbol has been found, it is returned as a tuple consisting of the symbol type and the string that matched the symbol. Some relevant extracts of this method are shown in Fig. 4.5. If at any stage the scanner encounters incorrect input, it returns a special no symbol result, indicating that it was unable to identify the input tokens.

4.4.3 Recursive-Descent CSP Parser

A recursive-descent parser works by starting at some goal production and trying to match lower productions and symbols according to the grammar of the source language [44]. In a recursive-descent parser, productions are represented by methods, which either match input symbols or call further production methods, thus delegating the matching of production rules to the appropriate methods until all input symbols have been matched [33, 44]. This technique is only suitable for *LL(1)* grammars that are free of left recursive rules [33]. Therefore, for use in this

```
def getSym(self):
    while (self.ch > EOF and self.ch <= ' '):
        self.getChar()
    symLex = []
    symKind = noSym
    if self.ch.isdigit():
        symLex.append(self.ch)
        self.getChar()
        while self.ch.isdigit():
            symLex.append(self.ch)
            self.getChar()
        symKind = numSym
    elif self.ch.isalpha():
        ...
    else:
        symLex.append(self.ch)
        if self.ch == EOF:
            symLex = list('EOF')
            symKind = EOFSym
        elif self.ch == ';':
            symKind = semicolSym
            self.getChar()
    ...
    self.sym = (symKind, ''.join(symLex))
```

Figure 4.5: Extracts from the `getSym` method of the prototype scanner.

prototype, the CSP grammar was extensively refactored and cut down to make it easily parsable by a recursive-descent parser. It must be noted that this grammar is not meant for use as a working component of Hydra and is purely for the purposes of investigating the code generation stage of compilation. The modified EBNF grammar for CSP is shown in Fig.4.6.

As per the above grammar, the prototype parser implements the following methods: `program`, `parallel`, `process`, `command_list`, `command`, `expression`, `input_command`, `output_command`, `assignment`, `repetitive`, `alternative`, `guarded`, `guard` and `guardlist`. To aid the parsing process, a number of helper methods have been created. The `symkind` method extracts and returns the type of symbol from the symbol tuple returned by the scanner. The `symlex` method returns the string matched by the scanner for the returned symbol. The `reportError` method prints the supplied error message. The `abort` method prints the supplied error message and halts parsing and is used in cases where the parser cannot recover from bad input. And finally, the `accept` method takes the symbol to be matched and an error message for when the symbol is not found. It then attempts to match the symbol and update the current symbol with the next symbol in the input

```
program = process | parallel .
parallel = '[' process {'||' process} ']' .
process = IDENT '::'  command_list .
command_list = {command ';'} .
command = IDENT (assignment | input | output) .
            | alternative | repetitive .
expression = IDENT | NUMBER .
input = '?'  IDENT .
output = '!'  expression .
assignment = ':=' expression .
repetitive = '*' alternative .
alternative = '[' guarded { '[]' guarded } ']' .
guarded = guard '->' command_list .
guard = guardlist | input .
guardlist = bool { ';' bool} [ ';' input ] .
```

Figure 4.6: Modified EBNF grammar for the Hydra prototype.

stream. If, however, the symbol is not found, it calls the `abort` method with the supplied error message.

The `program` method represents the goal production and is called when the parser is started. The parsing process starts with an empty AST which is represented by `basetree` in the next example. As the parser goes about identifying productions, it passes the relevant portions of the tree to the production methods it calls. These production methods then build up the tree by adding the appropriate tree nodes to the AST in the correct positions and filling in the appropriate details for these nodes such as identifier names. An example of some of the production methods and AST construction can be seen in Fig. 4.7.

The process of building this parser by hand provided some important insights. These insights include the importance of identifying channels and processes and ensuring that information is available in the top-level nodes of the AST as opposed to just adding them to the tree at the level they were discovered. It was also noted that parsing the full CSP grammar by hand would not be feasible as even with the simplified grammar, the construction of the AST was tedious and extensions to the grammar would require significant changes to many of the production methods. For this reason, a more solid approach to parser construction was needed, which ultimately meant making use of a compiler generator, specifically ANTLR using the Python target language runtime.

```
def program(self):
    # program = process | parallel .
    if self.symkind() == self.syms.lBrackSym:
        self.parallel(self.basetree)
    else:
        self.process(self.basetree)
    s = pickle.dumps(self.basetree)
    self.codegen.programEndFound(s)

def parallel(self, progtree):
    # parallel = '[' process {'||' process} ']' .
    partree = Hydra.ast.Parallel()
    progtree.node.append(partree)
    self.scan.getSym()
    self.process(partree)
    while self.symkind() == self.syms.parallSym:
        self.scan.getSym()
        self.process(partree)
    self.accept(self.syms.rBrackSym, '] expected')
```

Figure 4.7: Production methods for the Hydra prototype parser.

4.5 Parser Generators

While handcrafted scanners and parsers are certainly viable and straightforward for many language translation tasks, the complexities of certain grammars can make the construction of such hand-coded parsers problematic [44]. Even with careful planning and a modular design, these parsers can become hard to understand and maintain, especially when semantic checking and code generation are incorporated [44]. For this reason, scanner and parser generators are typically used instead of manual parser construction. These parser generators take a set of productions for the intended grammar and automatically generate the corresponding scanner and parser modules [44]. A number of parser generators and parsing frameworks for Python were investigated. The strengths and weaknesses of each parser generator were assessed and the parser generator most suited to the task of translating CSP was selected for use in Hydra. The main criteria used in the selection process were ease of use, ability to parse CSP, availability of documentation, development activity and support for all stages of translator construction, from scanner to code generator.

4.5.1 ANTLR

ANTLR (ANother Tool for Language Recognition) is a parser generator that automates the construction of lexers and parsers [33]. Given a formal description of the source language, ANTLR (version 3.1.1) is able to generate the appropriate lexer and parser modules [33]. ANTLR also supports the addition of code segments to the parser, allowing for language translation and code generation [33]. It is also very flexible and automates many common parser construction tasks [33]. ANTLR generates language recognisers that use a fast and powerful *LL(*)* parsing technique, which is an extension to *LL(k)* that uses arbitrary lookahead to make decisions [33]. This parsing strategy makes ANTLR suitable for all parsing and translation problems, from the simplest to the most complicated language translation tasks [33]. For grammars that do pose a problem, ANTLR's backtracking functionality allows the parser to work out the correct course of action during runtime, and partial memoization of results means that this can be achieved with linear time complexity [33].

ANTLR generates human-readable code that is easily incorporated into other software projects [33]. ANTLR v3 features improved error reporting and recovery over its predecessors for the generated parsers [33]. Dynamically scoped attributes allow rules to communicate, thus facilitating semantic checking. The code generation features of ANTLR are also quite advanced, with formal abstract syntax tree construction rules allowing ASTs to be constructed easily instead of developing actions to manually construct the AST. Another feature in ANTLR's favour is its tight integration with StringTemplate, which is a template engine for generating structured text such as source code [33]. This makes the code generator easily retargetable as only the template needs to be changed to generate code for a new target language.

ANTLRWorks is a grammar development IDE for ANTLR grammars that allows for the visualisation and debugging of parsers generated in any of ANTLR's supported target languages [33]. An example screenshot of ANTLRWorks can be seen in Fig. 4.8. ANTLR supports multiple target languages such as Java, C#, Python, Ruby, Objective C, C and C++, with Python support being of the greatest relevance to the Hydra project [33]. ANTLR is also actively supported with mailing lists, an informative and frequently updated project website and active project development. Overall, ANTLR is easier to understand and use than many of the other compiler generators that are discussed hereafter. Finally, ANTLR has a wealth of documentation available, from a project wiki, to examples, mailing list archives and most importantly, the book for ANTLR version 3 written by ANTLR's creator, Terence Parr [33].

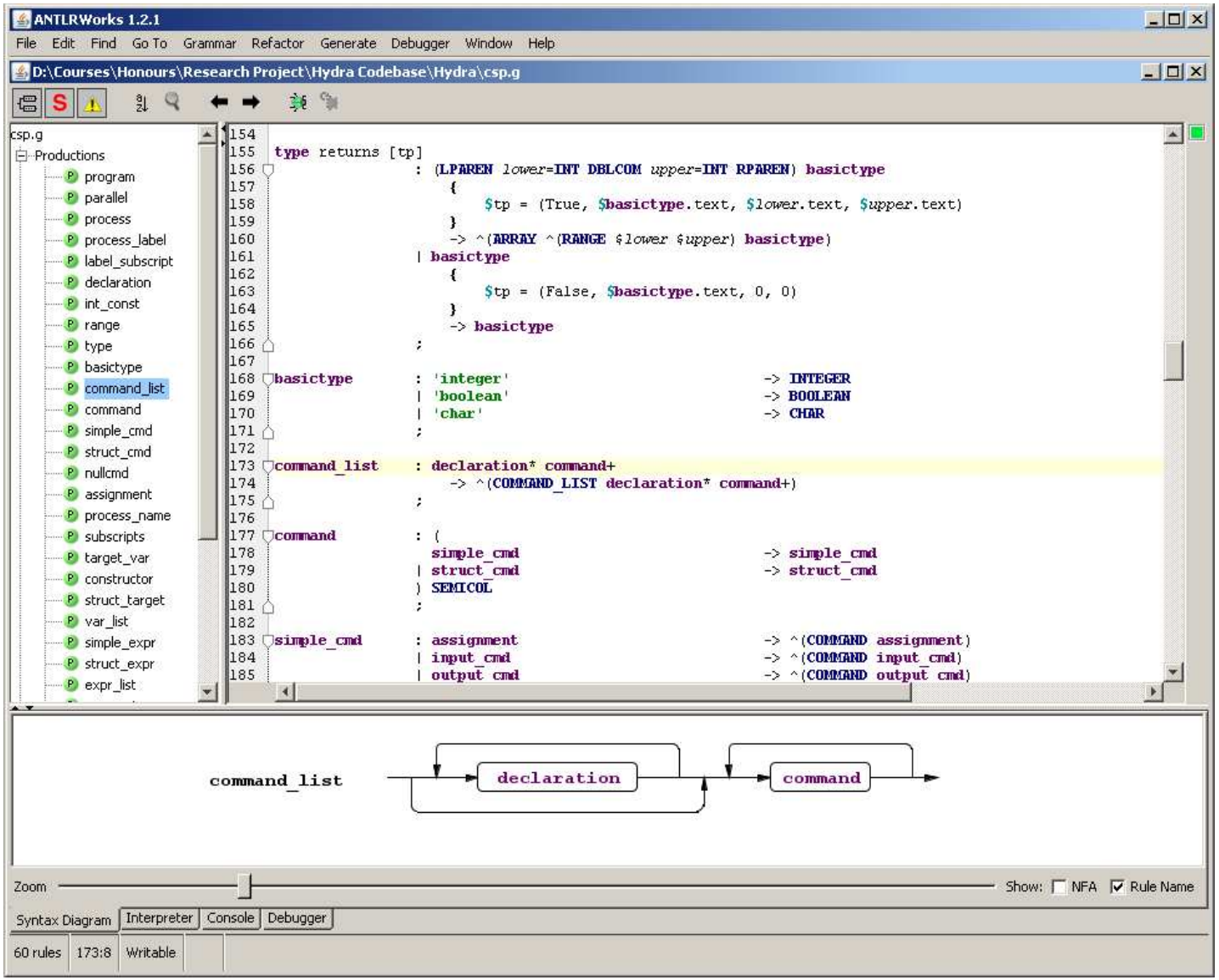


Figure 4.8: Screenshot of the ANTLRWorks graphical development environment.

4.5.2 CocoPy

CocoPy is a Python implementation of the Coco/R compiler generator (Coco/R stands for compiler compiler generating recursive descent parsers) [27]. CocoPy takes an attributed grammar of a source language, described in Cocol notation, and generates a scanner and a parser for this language [27]. The scanner is constructed as a deterministic finite automaton and the parser makes use of recursive descent and allows for symbol lookahead and semantic checks to be added to the parser [27]. This means that CocoPy is able to accept the *LL(k)* class of grammars as input and is thus suitable for parsing CSP with some slight modifications to the grammar. The use of Cocol as the grammar specification language makes CocoPy relatively easy to use as an EBNF grammar for CSP can be converted to Cocol without much hassle.

The greatest advantage of CocoPy is the availability of documentation. Coco/R is very well documented with numerous textbooks and online resources, while CocoPy itself has adequate documentation and examples. CocoPy supports attributed grammars, which allow actions to be incorporated into the parsing process for the purpose of semantic checking and code generation. However, the code generation aspect is more suited to generating output for stack-based architectures [44]. Another downside, as evidenced by the release notes, is that the CocoPy implementation of Coco/R is not complete and is not equivalent to the latest Java and C# implementations. The last update was released in late 2007, casting doubt on the project's development activity and likelihood of further updates.

4.5.3 Parsing and PyParsing

The Parsing module is a pure-Python module that implements a *LR(1)* parser generator, with Characteristic Finite State Machince (CFSM) and Generalised Left-to-right Rightmost derivation (GLR) parser drivers [15]. The Parsing module makes use of a very powerful and scalable algorithm for *LR(1)* parser generation, instead of the somewhat limited *LALR(1)* or *SLR(1)* algorithms seen in most *LR* parser generators [15]. The Parsing module also provides robust conflict resolution mechanisms and extensive error checking [15]. The source language grammar is specified in Python, which is fairly straightforward, but not as straightforward as using an EBNF style notation. Furthermore, the resulting rules for Parsing are not as clear and easy to read as an attributed EBNF style notation. The powerful *LR(1)* algorithm used by Parsing is likely to be sufficient for implementing a CSP parser and the conflict resolution mechanisms will help address any issues that arise.

However, the Parsing module has some significant drawbacks. Firstly, there is very little documentation and the documentation that does exist is in the form of docstrings within the Parsing

module code and a single, fairly simple example parser. It is also unclear how suitable the Parsing module is for code generation. And finally, the last update to the project was in August of 2007, meaning that any bugs or issues are unlikely to be addressed quickly.

The PyParsing module is a parser framework written in Python and is suited to creating and executing simple grammars [30]. This module provides a library of classes that can be used to construct the parser directly within a Python program in a fairly straightforward manner [30]. The PyParsing module makes use of various Python features to allow the production rules to be implemented directly in Python using an EBNF-like notation [30]. This makes PyParsing fairly easy to use, however, it is unclear how semantic checks and code generation are implemented as the production methods simply return a list of parsed tokens. It is also unclear which parsing algorithm is used, and even though examples show that PyParsing is able to parse the Python language, it is not immediately apparent whether it is able to parse and translate CSP successfully. Documentation for PyParsing is available in a variety of forms, from numerous detailed code examples to mailing lists, wiki documentation and published articles. Another positive remark for PyParsing is that it appears to be updated fairly frequently, with the last update occurring during October 2008.

4.5.4 PLY

PLY is a straightforward implementation of the lex and yacc tools and is implemented entirely in Python [3]. As with lex and yacc, PLY uses an *LR* (*LALR* specifically) parsing technique and is reasonably efficient and suited to large grammars [3]. PLY supports the majority of lex and yacc's features, such as empty productions, precedence rules, error recovery and mechanisms for dealing with ambiguous grammars [3]. PLY also provides extensive error checking and grammar validation to aid in the development of the parser [3]. Lexer and Parser rules are specified in separate files and are written as fairly straightforward Python code. PLY's *LR* parsing technique makes it a viable choice for implementing a CSP parser, although the resulting parser generated by PLY does not provide any additional features to aid in the code generation phase. The available documentation is detailed with numerous example programs, which help to clarify aspects of PLY's usage. Updates to PLY, while originally fairly frequent, are now few and far between, with only one update in 2007 and another in May 2008.

4.5.5 Wisent

Wisent is a Python based parser generator that converts a description of a context free grammar into Python parser code, which is able to parse source code according to the supplied grammar

[46]. Wisent provides helpful error messages, both for errors in the input grammar and when the parser encounters invalid input [46]. The parser will attempt to continue parsing and produce a list of errors when parsing is complete, instead of stopping for each error [46]. The context free grammar is supplied to Wisent in a separate grammar file, which has an EBNF-like syntax [46]. Wisent currently generates *LR(1)* parsers, with support for *LALR(1)* parsers in development, making it a viable choice for parsing CSP. Another useful feature of Wisent is that the generated parser has no dependencies on Wisent itself and can easily be incorporated into other projects [46]. Once the parser has finished parsing the input, it returns a parse tree [46]. This can be used for code generation, but an abstract syntax tree would be more appropriate. The parser also lacks features to make code generation easier. Documentation, while available, is fairly sparse and examples are rather basic. The last update for Wisent was in March 2008 and it is not apparent what the update schedule was before this date.

4.5.6 Yapps, Yappy and Yeanpypa

These three tools are not discussed in as much detail as the previous tools as it was immediately apparent that they were not likely to be suitable candidates, and besides, similar implementations have already been discussed for some.

Yapps (Yet Another Python Parser System) is another easy to use parser generator, written in Python [34]. Yapps is simple, easy to use, and produces human-readable *LL(1)* recursive descent parsers. Grammars are specified in Python following a similar format to PyParsing, but with the ability to add attributes to the grammar, much like Cocol. Unfortunately, Yapps is not particularly flexible and is more suited to simple parsing tasks such as parsing logs and config files.

Yappy is another tool for generating lexical analysers and *LR* parsers for Python applications [32]. Yappy is able to construct *SLR*, *LR(1)* and *LALR(1)* parsing tables and handle ambiguity provided the appropriate priority is given to the ambiguous elements. Yappy is useful for teaching *LR* parsing techniques, but does not provide much in the way of special functionality for use in a complex language translation project [32]. The lack of updates since 2006 is another reason why Yappy was not investigated further.

Yeanpypa is a parser framework written in Python and is very similar to PyParsing. It is used to contruct recursive-descent parsers in Python code in much the same way as PyParsing allows [10]. While fairly simple to use, the documentation is limited to a basic introduction and API documentation and it is unlikely that Yeanpypa is able to handle CSP successfully.

4.5.7 Parser generator selection

Based on the above comparisons, a summary of which can be seen in Table 4.1, ANTLR v3 (version 3.1.1) was chosen over the other compiler generators. Its powerful *LL(*)* parsing method, which supports backtracking and memoization, makes it an ideal choice for parsing CSP code. It is able to generate the lexer, parser and tree walker in pure Python, allowing for easy integration into the Hydra project. The tree rewrite mechanism allows for tailoring the abstract syntax tree to meet the needs of the code generation process. The ANTLRWorks grammar IDE makes developing and debugging grammars far easier, with its built-in rule visualiser, interpreter and debugger. Finally, ANTLR's support for tree walker grammars and StringTemplate makes code generation far simpler and allows for the parsing process to be kept separate from the code generation process, thus increasing maintainability. The extensive documentation and frequent update schedule were also important factors in ANTLR's selection as they ensure that any issues arising during development are likely to be easily and quickly resolved.

4.6 ANTLR Grammar for CSP

In order for ANTLR to generate the parser and lexer, the grammar for the source language must be formally described in an ANTLR grammar file [33]. The ANTLR tool is then given this grammar file as input, and as output, it produces the parser and lexer modules written in the specified target language [33]. The grammar file is separated into two sections, the first for lexer rules and the other for parser rules. The lexer rules specify the symbol or symbols that must be matched for each lexical token in the source language, while the parser rules specify the syntactical structure of the source language. Each production rule, starting from the goal production, describes the ordering of tokens and sub-rules required to match the given rule.

4.6.1 The Lexer

The lexer for CSP is relatively simple as the CSP grammar is minimalist and mostly unadorned. However, this simplicity leads to numerous ambiguities and as such, a few minor changes have been made to the original CSP grammar to make parsing easier. These changes, along with any custom tokens required for additions to the language, are described below. The full set of lexer rules can be seen in Appendix A.1.

The lexer for Hydra was defined such that all whitespace and comments are ignored. Single-line comments are supported, starting with '–' and ending in a newline. The first and most obvious change to the original CSP grammar is the use of semi-colons at the end of all statements.

Compiler generator	Parsing strategy	Ease of use and flexibility	Documentation	Code generation support	Development activity
ANTLR	LL(*) with backtracking	Very good	Very good	Very good - custom AST construction and StringTemplate	Actively developed
CocoPy	LL(1), LL(k)	Very good	Very good	Good - attributed grammars allow for code generation actions	Stalled
Parsing	LR(1)	Average	Poor	Poor	Infrequent
PLY	LALR	Good	Good	Poor	Infrequent
PyParsing	Unknown	Good	Very good	Poor - list of recognised tokens returned by parser	Actively developed
Wisent	LR(1)	Good	Average	Average - returns a parse tree	Infrequent
Yapps	LL(1)	Average	Average	Poor	Infrequent
Yappy	LR(1), SLR, LALR	Average	Average	Poor	Stalled
Yeanpypa	Unknown	Poor	Poor	Poor	Unknown

Table 4.1: Comparisons between Python based compiler generators.

The original grammar only called for semi-colons to be used between multiple statements on a single line, otherwise a newline indicated the end of a statement. While it would be possible to maintain the original notation, it would mean that whitespace could no longer be ignored, thus making the parser more complex. This minor compromise was therefore deemed acceptable for the sake of reducing parser complexity.

Since there is no symbol for '→' and '□' on common keyboards, '->' and '[]' were used in their place for the *guarded* statement. The Hydra lexer supports four basic expression types, namely identifiers, characters, integers and booleans. Identifiers start with a lowercase letter of the alphabet, and can be followed by any combination of uppercase and lowercase letters, digits and the underscore character. Characters can be any valid ASCII character, denoted between single-quotes. Integers are simply defined as a series of digits. And finally, Boolean expressions are denoted by either 'True' or 'False' and are case-sensitive. A summary of these changes can be seen in Table 4.2.

Production	Original Lexer Tokens	Altered Lexer Tokens
Alternative	'[' and ']'	-
Parallel	'[' and ']'	'[[' and ']]'
Guarded	'→' and '□'	'->' and '[]'
All statements	End in ';' or newline	End in ';'

Table 4.2: Summary of token changes to the CSP lexer.

Another important change that warrants discussion is the removal of expression operators such as the arithmetic and Boolean operators. In their place, the ability to use Python expressions was added, allowing for much greater flexibility when it comes to expressions. The Python code is enclosed in braces and can be any valid Python expression. To support functionality from Python's vast module collection, the ability to add Python import statements to the beginning of the program was added. These import statements are preceded by 'include' and are enclosed in braces. The rationale behind this rather significant change centers around Python's ability to evaluate expressions specified in string format during runtime. This removes the burden of parsing and evaluating expressions and essentially gets the Python interpreter to do this on behalf of the parser. This feature also allows for the use of all of Python's data types, bypassing the limited data type support natively provided by the parser.

4.6.2 The Parser

The parser section of the ANTLR grammar consists of a number of productions based on the constructs described in the CSP programming notation. Where possible, an attempt has been

made to develop a parser capable of recognising all the constructs present in the original grammar. One important aspect to note regarding AST construction is that by default, ANTLR will return a flat AST structure that simply represents the matched tokens. It is therefore necessary to use ANTLR rewrite rules to specify the structure of the AST. Only the relevant aspects of parsing CSP will be described here, however, the full set of parser rules can be seen in Appendix A.1.

The ANTLR grammar starts off with a number of options that modify the way that ANTLR generates the parser code. While there are numerous options available, only the following were actually required for Hydra. Firstly, Python was specified as the target language and the resulting output from the parser is in the form of an AST using the CommonTree label type provided by ANTLR. Other options include enabling backtracking and memoization. After the necessary options are defined, a list of tokens is supplied. These tokens are not the same as those used in the lexer and do not have any corresponding symbols associated with them. Instead, these tokens define custom labels that can be used by the AST rewrite rules for constructing the AST nodes. Thereafter, the language production rules are defined, starting with the goal production.

The goal production for the Hydra version of CSP is the `program` rule. This production starts with an optional *imports* section, which is followed by a *command list*. Additionally, the program rule defines and initialises a variable list, which is used for scope checks, using ANTLR's dynamic scoping features. This rule produces an AST node with the `PROGRAM` token as its root and the *imports* and *command list* as its branches. The remaining rules follow a similar pattern when constructing AST nodes. However, complications may arise during AST construction when productions containing optional elements are encountered. Fortunately, ANTLR's syntactic predicates allow for these situations to be handled explicitly, thus ensuring consistent AST construction. An example of an AST for a very simple CSP program can be seen in Fig. 4.9.

The *command list* then allows zero or more *declarations*, followed by one or more *commands*. The `declaration` production allows for the declaration of one or more variable identifiers of a given type. It also makes sure that these variables are added to the program's variable list with the appropriate scope, and makes sure that variables can only be declared once. The `command` production separates the different *commands* into two types. The first type is the *simple command*, which includes *assignment*, *input* and *output*, while the second type is the *structured command*, which includes the *alternative*, *repetitive* and *parallel* commands. The *alternative* command allows for a number of *guarded* commands, each with a list of *guards* and a *command list*. This is similar to the `if-elif` construct in Python. The *repetitive* command starts with an asterisk and is followed by an *alternative* command. The *repetitive* construct is

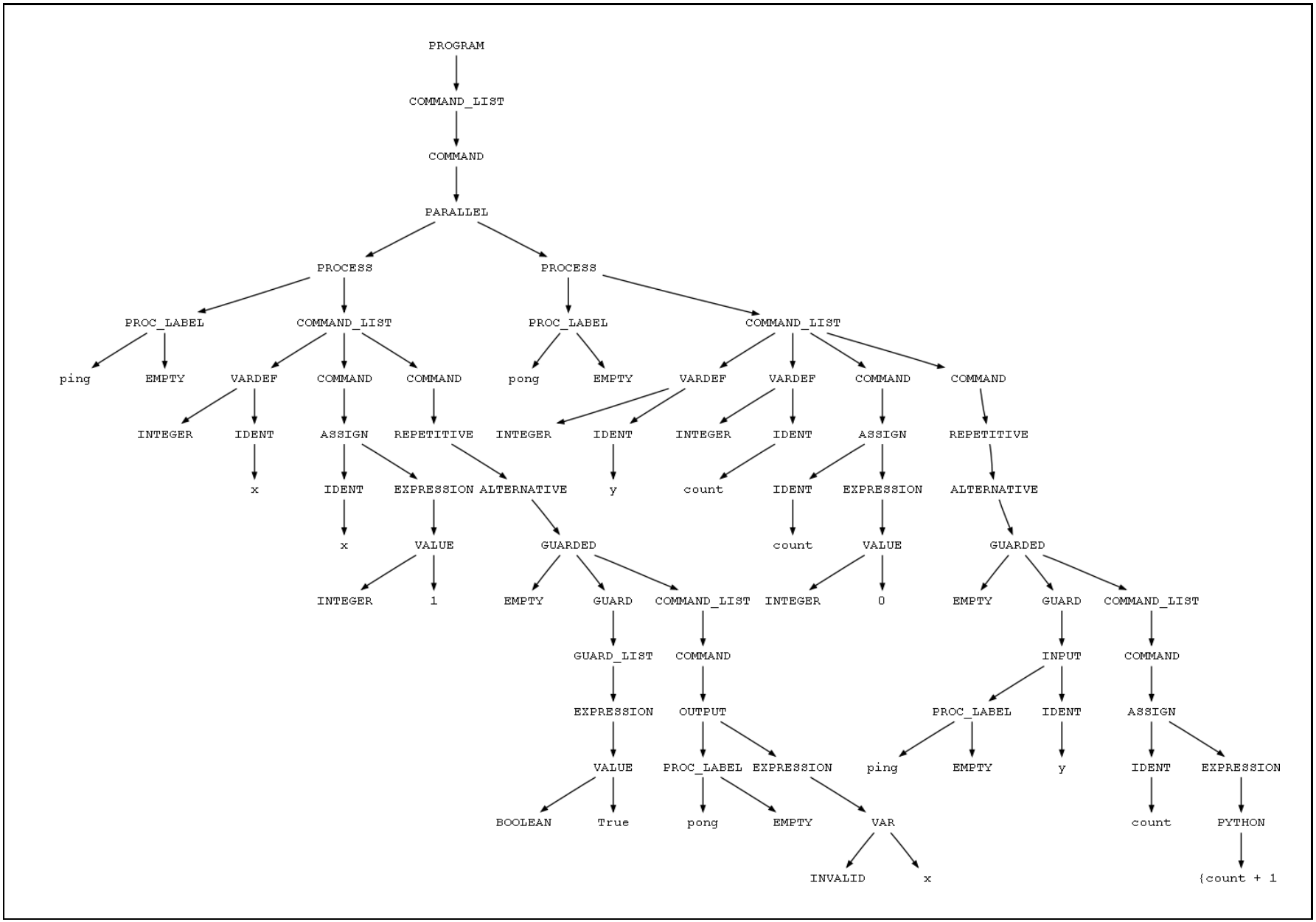


Figure 4.9: Example AST generated for a simple CSP program.

much like a while loop that continues to loop while any of the *guards* in the *alternative* are still active.

The `parallel` production, although very simple in its appearance, is of paramount importance as it defines the concurrent architecture of the program. It takes a list of one or more *processes* to be executed in parallel. During execution, these *processes* are spawned asynchronously and may execute in parallel, thus achieving one of the project goals; execution of code over multiple processors. Related to this is the equally important *process* construct, which is represented by the `process` production. This production defines a *process* as a *process label* followed by a *command list*. The *process label* allows for both named *processes*, with the option of *label subscripts*, and anonymous *processes*. A *process* is essentially a block of statements that can be executed either in sequence or in parallel with other such *processes*.

Another important set of CSP constructs is the *input* and *output* commands. These essentially define the *channels* of communication between *processes* and provide a synchronisation mechanism in the form of a rendezvous. The `input` production takes a *process name*, specifying the *source process*, and a *target variable*, which specifies where the result will be stored. The `output` production takes a *process name*, specifying the *destination process*, and an *expression*, which specifies the value to be sent. *Expressions* and *target variables* are either simple or structured, with the simple versions referring to a single value or target and the structured versions allowing for multiple values or targets to be specified in a tuple-like fashion.

An *expression* can take a number of forms and is represented by the `expression` production, which allows for both *simple expressions* and *structured expressions*. A *simple expression* can take the form of either an identifier, an integer, Boolean, character value, or a Python expression. As described in Section 4.5, the parser allows for *expressions* written in Python code. Unlike the other kinds of expressions, it is not possible to determine the type of the value that is returned by the Python expression during parsing. As such, the `expression` production returns 'python' as the type for these expressions, and the actual type for the other expressions. ANTLR's support of production attributes allows for the passing of type information between productions, as well as other relevant information. This type information is used for semantic checks in many of the productions that involve assigning values to identifiers. These semantic checks are made possible by using ANTLR's grammar actions functionality.

This concludes the discussion on Hydra's parser construction. While much can still be said about the finer details of the parser, such a discussion would likely be of little value to the overall objectives of the project. The full ANTLR grammar for the CSP parser can be seen in Appendix A.1.

4.7 Summary

It has been seen that language translation is a multifaceted task consisting of two major phases, namely the front-end construction dealing with language recognition, and the back-end construction dealing with code generation. This chapter focused on the front-end construction, which involved developing a parser for CSP. It was also showed the many different techniques for parsing, each with its own strengths and weaknesses.

The development of Hydra's parser was carried out in two phases. First, a basic prototype parser was crafted by hand for a cut-down version of the CSP grammar. This phase was for experimentation purposes and was not intended for incorporation into the final Hydra framework. The second phase involved the construction of a parser for inclusion in the Hydra framework. Instead of hand-coding this parser, a compiler generator tool was used as such an approach would be more flexible and less prone to error. A number of compiler generator tools were assessed to find the best tool for generating a parser for CSP and assisting in code generation. ANTLR was found to be the best compiler generator due to its plethora of features, good documentation and ease of use. The ANTLR based lexer and parser grammars for CSP were then described, leading to the conclusion that it is certainly possible to use CSP as a source language, at least as regards parsing.

Chapter 5

Code Generation

5.1 Introduction

This chapter introduces and discusses the main concepts and issues pertaining to the code generation phase of the Hydra framework. As discussed in Chapter 4, the CSP parser for Hydra takes a CSP algorithm as input, parses this input and returns an AST that represents the semantics of the CSP algorithm. The resulting AST is then used as the input for the code generation module, which has the task of translating the AST into the final executable code. As with the parser described in Chapter 4, an initial, hand-crafted prototype of the code generator was developed for translating CSP to JCSP. This JCSP prototype is only compatible with the hand-crafted CSP parser described in Section 4.4 and, like its parser counterpart, is purely for experimentation and testing. After experimentation with the JCSP prototype had been completed, the actual Hydra code generation module was then developed.

The final code generation module was developed using the ANTLR compiler generator tool, and follows on from the ANTLR parser for CSP described in Section 4.6. This code generation module is responsible for outputting Python code that is semantically equivalent to the original CSP algorithm, using an underlying concurrent framework to implement the CSP constructs. Therefore, before getting involved in the actual code generation process, an evaluation of a number of existing concurrent frameworks was undertaken. Once an appropriate concurrent framework had been chosen, it was then possible to implement the code generation process with the chosen framework used to implement parallelism.

5.2 JCSP Prototype Code Generator

5.2.1 Background and Framework

The JCSP prototype has two primary tasks. The first task involves translating the AST into suitable JCSP Java code. The second task then compiles and executes the JCSP code at runtime, thus making the translation process appear as a single action, as opposed to the multiple stages that are actually involved in such a language translation task. The AST construction occurs during the parsing process and is described in Section 4.4.3, however it may be beneficial to include a bit more detail of the AST structure here.

Each node type in the AST is represented by a Python class inheriting from a base `ASTNode` class. The `ASTNode` class provides functionality for specifying static code segments, which can be used as templates. There are AST nodes for most of the fundamental CSP constructs and commands, with the node classes appropriately named: `Program`, `Process`, `Channel`, `Input`, `Output`, `Expr`, `Assign`, `Parallel`, `Alternative`, `Repetitive`, `Guarded` and `Guard`. The structure of these nodes is fairly generic, with each node consisting of a list of child nodes, and other relevant information such as identifier labels, type information or expression strings. The `Program` and `Process` nodes are special in that they also store a dictionary of all the *channels* that are used in the program and the relevant process. The `Channel` class is another special case in that it is not added to the AST as a direct result of a parser rule. Instead, *channels* are implied by the presence of *input* and *output* commands, therefore, when *input* and *output* commands are encountered, a `Channel` node is instantiated and added to the channel dictionary of the `Program` node and the relevant `Process` node. Once the AST has been constructed, it can be traversed by simply iterating through the node list of each tree node.

As stated in Section 4.4.1, once the AST has been constructed, it is serialised using Python's `pickle` module and is passed to the `programEndFound` method of the code generator module. The call to the `programEndFound` method signals the end of the parsing phase and indicates that the code generation process can begin. The `programEndFound` method is very simple as it only has to unpickle the AST and call the `finalise` method, supplying the AST object as an argument. The `finalise` method then calls the `generatecode` method to perform the actual code generation. Once code generation is complete, it compiles and runs the JCSP `PCMain` class that was generated. Compilation and execution of the JCSP program is achieved using Python's `os.system` method, which executes a system command supplied in string format. Using this method, the `javac` command is executed to compile the `PCMain.java` file and then the `java` command is executed to run the compiled `PCMain`

class. A drawback of using JCSP Java programs for the target output code is that it introduces a dependency on non-Python libraries and programs. Specifically, the JCSP prototype requires that the correct versions of both the Java runtime environment and the JCSP libraries are installed and correctly configured. This problem can be mitigated when using Python libraries exclusively as there are tools for generating setup scripts capable of automatically fetching and installing module dependencies.

5.2.2 JCSP Code Generation

The `generatecode` method is the core of the code generator module. As stated before, the `generatecode` method is responsible for translating and outputting the desired concurrent code. This is achieved by traversing and analysing the AST, constructing the appropriate code segments to represent each of the AST nodes, and then writing these code segments to Java files. To make the file output process simpler, code segments are appended to the appropriate dictionary element representing one of the output files. After all the code segments have been generated and added to the dictionary, the code generator iterates through all the key-value pairs in the dictionary. For each key-value pair, a file with name specified by *key* is opened for writing, and the contents of *value* are then written to this file.

Actual code generation starts with generating the `PCMain` class. `PCMain` is commonly used as the starting class for JCSP programs [5]. All the *channels* and *processes* are instantiated within the `PCMain` class and the actual process execution order is defined within the class's `main` method. First, the beginning code segment for `PCMain` is static and is simply appended to the output as is. Then, the *channels* need to be defined and instantiated. This requires iterating through the dictionary of `Channel` nodes stored in the root `Program` node and outputting the appropriate *channel* definitions. *Channels* are named by appending the source *process* name to the destination *process* name.

The `Program` node's child nodes are then searched until a `Parallel` node is found. The `Parallel` node then lists the `Process` nodes that need to be translated. These `Process` nodes specify the names of the *processes* that need to be instantiated within JCSP's `Parallel` class. During *process* instantiation, each `Process` node's channel dictionary is analysed to determine which *channels* need to be passed to the *process*'s constructor. The *process* instantiation code is then output, along with the instruction to run the instantiated `Parallel` object. Finally, the remaining static code for the `PCMain` class is output.

Another pass is made over the `Parallel` node's child nodes and for each `Process` node found, a new Java class, named after the value of the `Process` node's label attribute, is generated. The `Process` node's channel dictionary is analysed and the appropriate *channel* defini-

tions are output as class variables. Using this information, a class constructor is generated that accepts the appropriate *channels* as parameters and initialises the class variables to the value of these parameters. The `run` method is then generated by iterating through the `Process` node's child nodes and generating the appropriate code for any *commands* that are found. Since no declaration construct was supplied in the prototype grammar, the code for generating the `run` method needs to analyse the process's *commands* at the start of the method and generate the appropriate variable declarations based on *commands* that involve assignment.

`Input` nodes generate code that simply uses the `read` method to read from the appropriate *channel* and assign the result to the specified target variable. `Output` nodes generate code that uses the `write` method to write the value of an *expression* to the specified channel, where an *expression* is represented by an `Expr` node. `Assign` nodes generate Java assignment statements that assign the result of an `Expr` node to a target variable. The `Alternative` and `Repetitive` nodes require a fairly complex translation process to generate the semantically equivalent code, especially with JCSP's convoluted `Alternative` class. Only the *alternative* construct will be discussed as the *repetitive* construct is simply represented in the output code as an *alternative* placed in a Java `while` loop with the appropriate loop control code added to the *alternative*.

When generating code for an `Alternative` node, it is necessary to pre-process all the associated `Guarded` nodes. This is as a result of JCSP's method for specifying *guards* in an *alternative.* Firstly, an array of the appropriate *input guards* must be generated. Secondly, an array of the appropriate *Boolean guards* must be generated. The size of these two arrays needs to be the same, therefore, the first iteration over the list of `Guarded` nodes is used to determine the size of the array, with the second iteration performing the task of actually generating the array structure. For *Boolean guards* with multiple *guard* elements, it is necessary to generate a single Boolean expression, which is achieved by combining *guard* elements using the Boolean 'and' operation.

However, some `Guarded` nodes contain either a *Boolean guard* or an *input guard*, but not both. This poses a problem as the array subscripts for the related *Boolean guards* and *input guards* need to match. This problem is easily resolved by supplying `'true'` for a missing *Boolean guard* or a `Skip()` guard for missing *input guards*. Java code is then generated to define and instantiate a JCSP `Alternative` object, which takes the *input guard* array as its constructor argument. A `switch` statement is then generated that takes the result of the `Alternative` object's `priSelect` method, which is passed the *Boolean guard* array as an argument. Then, `case` statements are generated corresponding to each array subscript in the input guard array. The statement blocks corresponding to each `case` statement is generated by processing the *command* nodes for the corresponding `Guarded` node.

The above description of the code generation phase provides a general overview of the translation process involved in converting CSP to executable code for the JCSP prototype. The actual method used to traverse the AST is rather mundane and is unlikely to provide any value to this discussion. However, a number of lessons were learned from this manual translation process. The main lesson being the need for multiple traversals over parts of the AST to retrieve necessary information from nodes further down the tree. Other lessons included the techniques necessary for implementing *alternative* statements and *guards* and for defining *channels* and *processes*.

5.3 Concurrent Frameworks

The translation from one language or notation to another is by no means a simple task. A code generator needs to be developed that is able to interpret the semantics of the source language and produce a semantically equivalent version in a target language [44]. The complexity of the code generator is often dependent on the complexity of the target language or architecture. Simple stack based architectures, such as Assembler, are usually fairly straightforward to generate code for, especially if a recursive decent parser is used [44]. However, other architectures, such as parallel systems, often require a more sophisticated approach to code generation [4, 8]. One approach involves developing all the necessary constructs and underlying framework from scratch. Needless to say, this can be very time consuming and complicated. A more practical approach is to find and use existing frameworks for the target architecture, adding custom code only for the functionality that is missing or incomplete [6]. The back-end concurrent framework for Hydra is built on top of a number of existing Python frameworks. These frameworks provide the constructs and architectural elements necessary to implement CSP programs in Python. This section briefly introduces each framework and highlights the role it plays in Hydra.

5.3.1 Python Remote Objects

Python Remote Objects (PYRO) is a simple yet powerful framework for working with distributed objects written in Python [14]. PYRO essentially handles all the network communication between objects, allowing remote objects to appear as local ones [14]. Additionally, PYRO provides remote method invocation functionality, which allows for methods from remote objects to be called locally [14]. PYRO can be used over a network, allowing processes to be distributed between a number of separate computers, or it can be used purely on the local machine to provide a convenient inter-process communication mechanism [14]. PYRO consists

of a special nameserver component that provides functionality for registering and retrieving remote objects. Client code is then able to register named objects with the PYRO nameserver and retrieve these objects using the specified name [14]. This remote object framework provides all the necessary functionality to implement CSP channels. Each communication *channel* between *processes* can be implemented as a remote `Channel` object with `read` and `write` methods. As such, PYRO plays a critical role in the implementation of the concurrent Hydra back-end.

5.3.2 PyCSP

While PyCSP has already been discussed in Section 2.6.2, recent updates to PyCSP framework have yielded some important functionality. As of writing, the latest version of PyCSP is version 0.3.0, which was released in May 2008 [9]. Since version 0.3.0, PyCSP provides network channel functionality using PYRO [9]. With the appropriate custom framework code, this new functionality can be leveraged to overcome PyCSP's greatest weakness, namely its reliance on Python's threading library. As described in Section 2.6.2, PyCSP has already implemented Python versions of most of the CSP constructs, such as the *process*, *channel*, *guard* and *alternative* commands [9]. PyCSP's `Parallel` class can still be useful for implementing parallel sub-processes, even though it is limited by Python's GIL. Therefore, PyCSP forms another critical component of the concurrent Hydra back-end as it removes the need to develop these CSP constructs from scratch.

5.3.3 River and Trickle

The initial investigation into suitable back-end frameworks yielded the River framework as a possible candidate for process distribution and remote method invocation. River is a Python framework for distributed and parallel programming [7]. It is a useful framework for writing parallel Python programs and prototyping parallel systems [7]. It has a number of features that make it very useful for a project such as Hydra. These features include: dynamic River VM discovery, process naming and creation, message passing and state management [7]. As with PYRO, River supports communication between remote objects or processes and remote method invocation [7]. River also has a number of extensions with the 'trickle' extension being of relevance to the Hydra project. Trickle provides a number of abstractions for the River functionality, such as process and data distribution with workload balancing, easy VM discovery and asynchronous process execution [6].

While River and Trickle appear to be the perfect platform on which to build the Hydra framework, a number of issues were encountered during prototype development. River and Trickle

require that code be developed for execution within the River VM, which is essentially a modified Python interpreter, since it is not possible to simply import the necessary modules. While this does not sound like much of a problem, it does have implications for the Hydra code generator and process distribution methods. Since processes would be distributed using the Trickle functionality, it would be necessary to run the Hydra framework completely within a River VM instead of simply importing the necessary functionality. This would then require users of Hydra to also use the River interpreter when developing their applications otherwise the Hydra code generator will not function correctly. These restrictions and other added complications, such as determining the path information for River and Trickle, were seen as definite drawbacks for both the user and the development of Hydra. However, River and Trickle could still be useful if the project were to be extended to multiple computers.

5.4 Python Code Generation

Now that an appropriate underlying framework has been found, it is possible to begin the process of generating Python code. One of the many strengths mentioned for ANTLR in Section 4.5.1 is its ability to recognise AST grammars and generate structured output using StringTemplate. These features are critical to generating the concurrent Hydra-based code easily and successfully.

5.4.1 ANTLR Tree Walker

ANTLR provides the ability to develop AST parsers that can be used to traverse the AST that was generated by the ANTLR parser [33]. This removes the burden of having to write tree traversal routines manually for the code generation process and significantly speeds up development. Not only can these ANTLR tree grammars parse abstract syntax trees, they can also incorporate all the usual ANTLR features such as grammar actions, syntactic predicates and rewrite rules [33]. These rewrite rules can also take the form of StringTemplate style template calls, thereby allowing the tree parser to generate code from predefined templates [33]. The Hydra code generator was developed using this tree walker and templates approach.

5.4.2 StringTemplate

As mentioned previously, StringTemplate is a library that provides functionality for defining static code templates with embedded structure information [33]. These template rules are stored

in a template group file that can be accessed by the ANTLR tree walker and used to generate structured output code [33]. This approach has the benefit of keeping the tree walker implementation clean and concise as well as improving code generator maintainability. It also allows for the code generator to be re-targeted without too many changes to tree walker itself [33]. The template rules take the form of a rule name with optional parameters and the corresponding template code, with any of the supplied parameters embedded therein. A simple example of two such rules and the corresponding tree walker rules can be seen in Figs. 5.1 and 5.2.

```
assignment :  ^(ASSIGN ^(IDENT ID) expr=expression)
-> assignment(ident={$ID.text}, value={expr});
simple_expr :  ^(EXPRESSION ^(PYTHON PYEXPR))
-> python_expr(expr={($PYEXPR.text)[1:-1]})
```

Figure 5.1: ANTLR tree walker rules for CSP.

Figure 5.1 shows two tree walker rules, the first being a rule that identifies assignment statements in the AST and the second being a rule for one of the many alternatives of a simple expression. The form of the templates used by these two rules is depicted in Fig. 5.2.

```
assignment(ident, value) ::= "<ident> = <value>"
python_expr(expr) ::= "eval('<expr>')"
```

Figure 5.2: StringTemplate rules for Python code generation.

With careful planning and the correct structuring of these rules, it is possible to generate concurrent Hydra programs without having to add too many custom code generation routines.

5.4.3 Implementation

As with the ANTLR grammar parser, the ANTLR tree grammar starts with a section for the options. The options for the Hydra tree walker include setting the target language to Python, setting the output format to use templates and instructing the tree walker to use the tokens defined in the CSP parser. The options section is then followed by a `@members` section that defines a number of custom variables and methods that are used during the translation process. The methods will be discussed as they become relevant. The tree grammar productions start with the `program` rule. This represents the goal production for the tree walker and is called when the tree walker is executed. The AST is parsed by starting with the `program` rule and recursively matching AST node tokens and sub-rules. When the whole AST has been parsed, the `program` rule compiles a list of all the main *processes* and *channels* identified in the AST and returns these as tree walker result attributes. When the `program` rule has been fully matched,

it calls the `program` template rule, which then generates the appropriate program skeleton using the supplied Python import statements, *command* list and list of program arguments. The program arguments are constructed using a `@member` method called `buildprocargs` that takes a list of *processes*. It essentially constructs code to handle command-line arguments and execute the relevant *process*. Of the remaining rules, only those that warrant discussion are described here, however, the listings for the ANTLR tree walker and StringTemplate group file can be found in Appendix A.2 an Appendix A.3 respectively.

The `parallel` rule is important in that it has two distinct behaviours. If it is parsing the top-level *parallel* construct, it will generate the appropriate code for executing over multiple Python interpreters. However, any `PARALLEL` nodes found further down the AST will generated using PyCSP's `Parallel` method. The rationale behind this is that current desktop computers have at most eight processor cores, therefore, implementing every process in a new Python interpreter instance is not likely to give the desired scaling and will just increases the memory usage of the Hydra program. The `process` rule and the corresponding `process_label` rule are fundamental productions in that they generates the *process's* method definition as a PyCSP `Process`. Thankfully, Python permits nesting of methods, making this code generation process much easier. The `process` rule also takes a list of *channels* that are used in the *process* and generates code to retrieve the relevant `Channel` objects from the PYRO nameserver using the `getNamedChannel` method. Since CSP allows for the definition of anonymous *processes*, a technique for handling and defining these methods was devised. The technique is fairly simple and involves incrementing a counter and appending this count to the end of the predefined *process* name, such as `'__anonproc_'`. It is worth noting that since anonymous *processes* have no user-defined name, it is not possible to use *input* and *output* commands within these processes and any attempt to do so will lead to undefined behaviour.

While a number of the CSP constructs, such as the *process*, *guarded*, *input* and *output* statements, allow for label subscripts, this functionality has not been implemented in the code generator. As such, the parser rules for identifying these subscripts have been disabled, even though the parser is capable of recognising them. Any attempt to use subscripts in a Hydra program will lead to parser errors. The reason for not including subscripts is that they introduce a great deal of complexity to the code generator. These constructs may be re-enabled at a later stage when the Hydra project is more mature. Another important issue regarding user defined labels and names is the issue of keywords. Python has a number of keywords that cannot be used as method or variable names. Therefore, all user-defined identifiers are sanitised by the `@members` method, `fixkeywords`, that simply prefixes an underscore to any identifiers that clash with know keywords.

Variable declarations are now supported by the `declaration` rule, unlike the JCSP prototype,

which had to pre-process assignment statements to generate declarations. Although Python does not require variable declarations, it was decided that simply assigning the value of `None` to these variables would probably reduce the chance of scope errors. Array declarations are another reason for generating declaration code in that CSP permits the declaration of array bounds, which can lead to out-of-bounds errors if the programmer attempts to reference an uninitialised list variable. Therefore, arrays are declared as a Python `list` with the appropriate number of elements all set to `None`. An `@member` method, named `arrayinit`, is used to generate the appropriate number list elements.

Python expressions and statements embedded in the CSP are handled by generating Python `eval` and `exec` calls respectively with the given expressions and statements as input. Structured expressions and variables are also possible using Python's `tuple` type. The `input_cmd` and `output_cmd` rules generate simple `read` and `write` method calls on the appropriate `Channel` objects. This is possible due to the fact that all the *channel* configuration and initialisation is handled at the beginning of the *process*.

The `repetitive` rule generates a Python `while` loop with the appropriate loop control variable set to `True` initially. Within the loop, an `if-else` statement is included with the `if` expression set to `False`. An *alternative* construct is then inserted between the `if` and the `else` statements and generates a list of `elif` statements. If all the *alternative* statements evaluate to false, the `else` statement is executed, setting the loop control variable to `False`, thereby stopping the loop. The `alternative`, `guarded`, `guardlist` and `guard` rules all serve to generate a list of elif statements that represent the Boolean expressions supplied by programmer and execute the supplied command list if that expression evaluates to true. *Input guards* are implemented using PyCSP's `Alternative` class and the `priSelect` method that uses order of appearance as an indicator of priority.

Once the AST has been traversed and the templates have generated the appropriate Python code, the resulting StringTemplate object containing the generated code is returned to the `cspexec` method where the code is distributed and executed.

5.4.4 Process Distribution and Execution

A relatively simple approach was taken to bootstrapping and executing the relevant processes once code generation was complete. As mentioned in Section 5.4.3, the `program` method of the tree walker object returns a list of *channels* and *processes* that need to be configured and executed. One of the problems encountered with using PyCSP's network channel functionality is that all *channels* need to be registered with the PYRO nameserver before the *processes* are able to retrieve the remote `Channel` objects. There is no easy way to add this registration

process to the generated program without encountering situations where one *process* requests a *channel* that has not yet been registered. This problem was addressed by registering all the necessary *channels* beforehand in the `cspexec` method of the `Hydra.csp` module. The list of *channels* received from the tree walker allows the `cspexec` method to simply loop through all the *channels* and register the appropriate *channel* names with the PYRO nameserver. Since this happens before *process* execution, there is no chance of *channels* being unregistered or multiple registrations occurring for the same *channel* name, thus breaking inter-process communication. Fig. 5.3 shows how the *channel* registration process was implemented.

```
chans = []
for i, chan in enumerate(outpt.channels):
cn = One2OneChannel()
chans.append(cn)
registerNamedChannel(chans[i], chan)
```

Figure 5.3: PYRO channel name registration.

Once the *channels* are registered, the *processes* are asynchronously executed using a simple loop and Python threads. The implementation of the *process* spawning routine can be seen in Fig. 5.4. The `cspexec` method then waits for the *processes* to finish executing and allows the user to view the results before ending the program.

```
class runproc(Thread):
def __init__ (self, procname):
Thread.__init__(self)
self.procname = procname
def run(self):
os.system('python hydraexe.py ' + self.procname)

proclist = []
for proc in outpt.procs:
newproc = runproc(proc)
proclist.append(newproc)
newproc.start()

for proc in proclist:
proc.join()
```

Figure 5.4: Asynchronous process execution.

5.5 Summary

The two phases of the construction of the code generator resulted in two separate code generators being developed. The first phase involved hand-building a simple code generator for JCSP. This was performed as an experiment to gain familiarity with CSP and code generation techniques. The second phase involved the use of ANTLR to develop a flexible, maintainable and powerful code generator. However, before beginning the construction of the ANTLR code generator, a number of existing concurrent frameworks were investigated to reduce the amount of custom coding required. The PyCSP and PYRO libraries were found to meet the requirements of such a concurrent framework.

Using ANTLR's tree grammar parser and StringTemplate support, a code generator was developed that is able to convert CSP algorithms to concurrent Python code and distribute this code for execution over multiple processor cores. With the completion of the code generator comes the completion of the development aspect of the project. Testing is therefore required to assess whether the Hydra framework actually meets the objectives it set out to achieve.

Chapter 6

Results

6.1 Introduction

Two forms of testing were performed to determine Hydra's degree of success in meeting the project objectives. First, a qualitative analysis was performed on various aspects of the Hydra framework. This involved constructing a simple CSP example and converting it using Hydra. The resulting output code was then analysed to determine if it represents an accurate representation of the CSP algorithm. A subjective analysis of Hydra's ease of use was also performed. The code was then executed and the operating system's process monitoring tools were used to determine whether or not the program was executing over multiple process cores.

Second, a quantitative analysis of Hydra's performance was undertaken. While the aim of this project is not to develop the fastest, most efficient framework possible, it is certainly worthwhile to perform such an analysis as performance is likely to affect the usability of the framework. The first quantitative test involves comparing the number of lines of code from the original algorithm to the number of lines of code in the resulting output program. The second series of tests involve obtaining timings for conversion overhead and channel communication overhead.

All testing was performed on the system configuration specified in Table 6.1.

6.2 Testing

6.2.1 Generated Code Analysis

An example Hydra CSP algorithm can be seen in Fig. 6.1. This is a simple program with two processes. The `producer` process outputs the value of `x` to the `consumer` process 10000

Category	Type
CPU	AMD Opteron 170 (2x2.0GHz)
Motherboard	ASUS A8R32-MVP Deluxe
Memory	2x1GB G.Skill DDR400
Hard Disk	Seagate 320GB 16MB Cache
Network	Marvel Gigabit On-board Network Card
Operating System	Microsoft Windows 2003 Server SP2

Table 6.1: Testing platform configuration.

times and the `consumer` process simply inputs the value received from `producer` and stores it in `y`. The resulting Python output code can be seen in Fig. 6.2.

```
from Hydra.csp import cspexec
prodcons = """
[[
  -- ping process : sends the value of x to pong
  producer ::
    x : integer; x := 1;
    *[
       {x <= 10000} -> {print "prod: x = " + str((x*x*x) % 10)};
                       consumer ! x;
                       x := {x + 1};
    ];
||
  -- consumer process : receives a value from producer and stores it in y
  consumer ::
    y, count : integer; count := 0;
    *[
       {count < 10000} -> producer ? y;
                          count := {count + 1};
                          {print "cons: y = " + str((y*y*y) % 10)};
       [] {count == 10000} -> {print "The count is: " + str(count)};
    ];
]];
"""
cspexec(prodcons)
```

Figure 6.1: Simple producer-consumer CSP example.

Looking at the output code, it is clear that Hydra has generated a semantically equivalent version of the CSP algorithm. Both *processes* are defined correctly, with correct *channel* initialisation and variable declarations. The repetitive commands are both present in the form of `while` loops with the appropriate control variables and *alternative* code. The *guarded* commands can also be seen in the form of the `elif` statements, with expressions and statement blocks correctly represented. The use of embedded Python statements is apparent in the inclusion of `eval` and `exec` statements. *Input* and *output* commands can be seen by the respective `read`

and `write` method calls on the *channel* objects. This example, while simple, is able to show most of the CSP constructs and their respective representation in Python using Hydra.

It apparent from the nature of the resulting Python code, that it is simpler and quicker to write a CSP algorithm and have Hydra convert it to Python. The incorporation of Python expressions and statements is also bound to make using Hydra easier as the programmer is free to use Python's powerful data-types and libraries within the CSP program.

The resulting Hydra program was then run and the Windows Task Manager was used to monitor the `python.exe` interpreter processes and overall CPU usage. The results of this test can be seen in Fig. 6.3 and Fig 6.4. To demonstrate the parallel execution effectively, the *guard* conditions for the `producer` and `consumer` processes were changed to True, thus creating infinite loops. This provided enough time to effectively demonstrate multicore usage.

The processor usage information gathered from the Windows Task Manager clearly shows that the Hydra program is executing on multiple CPU cores simultaneously. The 'CPU Usage History' graph shows that both CPU cores have processor usage of between 60 and 80 percent during the period of process execution. Furthermore, the 'Processes' list shows the Python interpreters that were running during that period. The Python interpreter with 0 percent CPU usage is the PYRO nameserver process, and the other three instances of Python are for the main Hydra program, the `producer` process and the `consumer` process. Adding their respective CPU usage values together equates to a CPU usage of 64 percent for the Hydra program, where 50% usage indicates the maximum usage for single CPU core. This proves that the Hydra program is executing in parallel and more CPU intensive algorithms are likely to further reinforce this distinction.

6.2.2 Basic Quantitative Analysis

The first test being conducted is a simple comparison between the number of lines of CSP algorithm code and the number of lines of converted Python code. The aim of this test is to demonstrate the work reduction benefits of using Hydra, since less lines of code have to be written to get the same result. A number of CSP algorithms were written, incorporating a mix of CSP constructs. These CSP programs were then converted to Python code and the relative line counts were compared.

From these results in Table 6.2, it can be seen that there is an increase of between 300 and 400 percent in the number of lines of code from CSP algorithm to the Python program. This essentially means that the programmer has to write less code to produce the desired program.

To test the channel communication overhead, a simple test was devised. The producer-consumer

Example	Lines of CSP	Lines of Python
1	34	121
2	18	67
3	22	78

Table 6.2: Line-count comparison

example was modified to take a time measurement before the *repetitive* and take another measurement after the *repetitive*. This was performed 100 times and the average was taken. The results can be seen in Table 6.3. Average compilation time was also computed. From the results it is apparent that there is very little overhead associated with compilation and communication.

Run	Communication (in seconds)	Compilation (in seconds)
1	0.001719	0.047
2	0.001719	0.078
3	0.001719	0.065
4	0.001720	0.079
5	0.001560	0.072
6	0.001559	0.068
7	0.001710	0.081
8	0.001559	0.062
9	0.001570	0.049
10	0.001559	0.078
Average	0.001639	0.0679

Table 6.3: Communication overhead.

6.3 Summary

From these results, it is apparent that the Hydra framework has achieved its primary objective. That is, it has demonstrated that it is possible to take a concurrent CSP algorithm and translate it into concurrent Python code that is capable of parallel execution. This translation process is automatic and does not require the user to implement their concurrent algorithm manually using one of the many existing concurrent frameworks. This automatic algorithm conversion feature is what makes Hydra easy to use. It was also found that Hydra did not add much in the way of computing overhead.

```
import sys
from pycsp import *
from pycsp.plugNplay import *
from pycsp.net import *

def __program(_proc_):
    @process
    def producer():
        __procname = 'producer'
        print '# producer'
        __chan_consumer_out = getNamedChannel("producer->consumer")
        x = None
        x = 1
        __lctrl_1 = True
        while(__lctrl_1):
            if False:
                pass
            elif eval('x <= 10000'):
                exec 'print "prod: " + str(x)' in globals(), locals()
                __chan_consumer_out.write(x)
                x = eval('x + 1')
            else:
                __lctrl_1 = False

    @process
    def consumer():
        __procname = 'consumer'
        print '# consumer'
        __chan_producer_in = getNamedChannel("producer->consumer")
        y = None
        count = None
        count = 0
        __lctrl_2 = True
        while(__lctrl_2):
            if False:
                pass
            elif eval('count < 10000'):
                y = __chan_producer_in.read()
                count = eval('count + 1')
                exec 'print "cons: y = " + str(y)' in globals(), locals()
            elif eval('count == 10000'):
                exec 'print "The count is: " + str(count)' in globals(), locals()
                count = 10001
            else:
                __lctrl_2 = False

    # process spawning
    if False:
        pass
    elif _proc_ == "producer":
        Sequence(producer())
    elif _proc_ == "consumer":
        Sequence(consumer())
    else:
        print 'Invalid process specified.'
__program(sys.argv[1])
```

Figure 6.2: Python code for producer-consumer example.

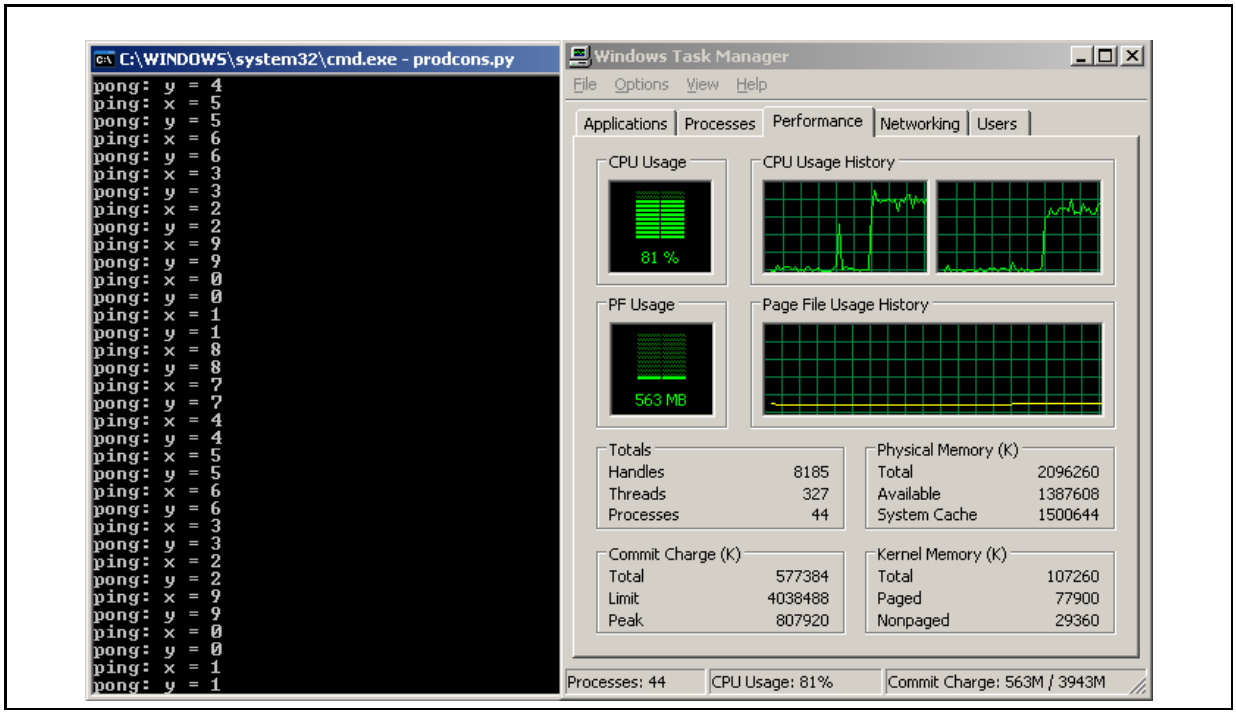


Figure 6.3: Processor activity during Hydra process execution.

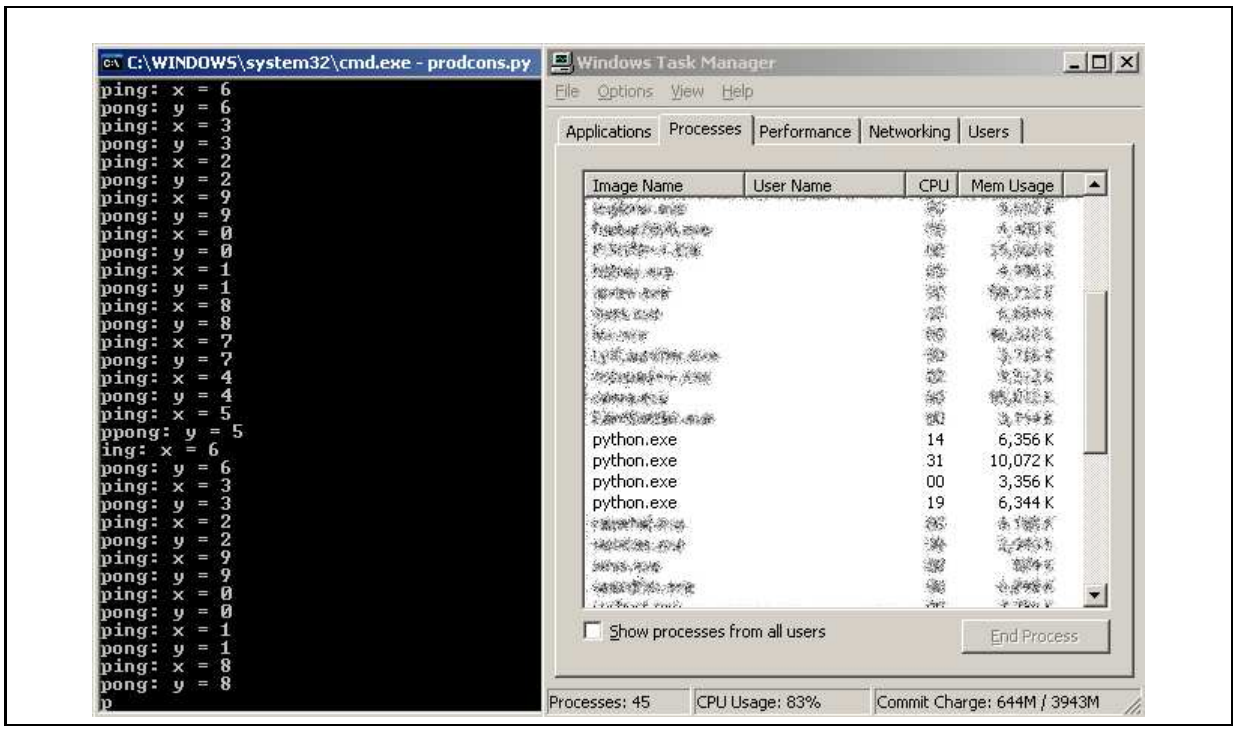


Figure 6.4: Python interpreter CPU usage during Hydra process execution.

Chapter 7

Conclusions

7.1 Summary

The goal of the Hydra project is the creation of a concurrent framework for Python. This framework is given the task of converting CSP code into concurrent Python code. This process involved the development of a parser for CSP. First, a prototype parser was coded by hand and then a fully working ANTLR parser was created after identifying ANTLR as the best compiler generator to use. It is also necessary to develop a code generator. A prototype code generator was hand-crafted to analyse the AST from the parser and generate JCSP. Thereafter, ANTLR was used to generate Python code from the AST supplied by the ANTLR parser. Finally, basic testing was performed to determine whether or not the Hydra framework was capable of meetings its objectives.

7.2 Revisiting the Objectives

The primary objective of this project was investigating the feasibility of converting a CSP algorithm into concurrent Python code. As can be seen by the results in Chapter 6, this objective has been achieved. It is therefore possible to take a CSP algorithm defined within a Python program and convert it into a concurrent Python code and have the concurrent program execute over multiple CPU cores. The objective of developing a flexible parser and translator was also achieved thanks to ANTLR's powerful parsing and code generation functionality. Results showed that it is possible to accurately convert CSP into Python using structured code templates. Results also showed that the translation process only adds a negligible amount of overhead to the program.

7.3 Future Work

The Hydra framework is still in its prototype phase. As such, there are a number of possible extensions. A short list of these extensions is listed below:

- A number of changes had to be made to the CSP grammar during parser and code generator development. Many of these compromises were made to speed up development as opposed to be actual requirements. Further studies could investigate how these changes could be reverted and how such a task would affect the parsing and code generation processes.
- Support for embedding Python in the CSP program was added, however, this support is very simplistic. It would be beneficial to research better ways of allowing CSP programs to use Python functionality.
- Semantic checking and error reporting for the parser and code generator are very weak. Further work could involve implementing stronger semantic checks and provide friendlier error reporting.
- This research only looked at implementing parallelism for a single computer system. Further studies could investigate extending the target architecture to multiple computers or Grid computing platforms.

Bibliography

[1] AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools, 2/E*, 2nd edition ed. Addison-Wesley, 2006.

[2] ARROWSMITH, B., AND MCMILLIN, B. How to program in ccsp, August 1994.

[3] BEAZLEY, D. Ply (python lex-yacc) [online]. May 2008. Available from: `http://www.dabeaz.com/ply/`, Accessed 31 October 2008.

[4] BEAZLEY, D., AND LOMDAHL, P. Feeding a large scale physics application to python. In *Proceedings of the 6 th International Python Conference* (San Jose, California, October 1997). Available from: `citeseer.ist.psu.edu/beazley97feeding.html`.

[5] BELAPURKAR, A. Csp for java programmers [online]. June 2005. Available from: `http://www-128.ibm.com/developerworks/java/library/j-csp1.html`, Accessed 31 May 2008.

[6] BENSON, G., AND FEDOSOV, A. Python-based distributed programming with trickle. In *PDPTA* (2007), H. R. Arabnia, Ed., CSREA Press, pp. 30–36.

[7] BENSON, G., FEDOSOV, A., GUTIERREZ, J., HARDIE, B., NGO, T., REYES, J., AND WU, Y. River - a python-based framework for rapid prototyping of reliable parallel runtime systems [online]. May 2008. Available from: `http://www.cs.usfca.edu/river/index.html`, Accessed 25 May 2008.

[8] BJØRNDALEN, J., VINTER, B., AND ANSHUS, O. *PyCSP - Communicating Sequential Processes for Python*. IOS Press, 2007.

[9] BJØRNDALEN, J. M. Pycsp [online]. May 2008. Available from: `http://www.cs.uit.no/~johnm/code/PyCSP/`.

[10] BRÜCKNER, M. yeanpypa - yet another python parser framework [online]. 2008. Available from: `http://www.slash-me.net/dev/snippets/yeanpypa/documentation.html`, Accessed 31 October 2008.

[11] BROWN, N. C. C. Rain: A new concurrent process-orientated programming language. In *Communicating Process Architectures 2006* (September 2006), P. Welch, J. Kerridge, and F. Barnes, Eds., IOS Press, pp. 237–251.

[12] BROWN, N. C. C. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In *Communicating Process Architectures 2007* (July 2007), A. A. McEwan, W. Ifill, and P. H. Welch, Eds., IOS Press, pp. 183–205.

[13] CLAYTON, P., AND ZHAO, D. Distributed and parallel processing. Print, Department of Computer Science, Rhodes University, April 2008.

[14] DE JONG, I. Pyro - python remote objects [online]. May 2008. Available from: `http://pyro.sourceforge.net/`, Accessed 2 June 2008.

[15] EVANS, J. Parsing – python parser generator module [online]. August 2007. Available from: `http://www.canonware.com/Parsing/`, Accessed 31 October 2008.

[16] FOR COMPUTING MACHINERY, A. The acm computing classification system [1998 version] [online]. October 2008. Available from: `http://www.acm.org/about/class/ccs98-html`, Accessed 25 September 2008.

[17] GIMBLETT, A. Parsing csp-casl with parsec. Presentation, November 2006.

[18] HALL, A., AND CHAPMAN, R. Correctness by construction: Developing a commercial secure system. *IEEE Software Jan/Feb* (January/February 2002), 18–25.

[19] HASSELBRING, W. Programming languages and systems for prototyping concurrent applications. *ACM Comput. Surv. 32*, 1 (2000), 43–79.

[20] HAYES, B. Computing in a parallel universe. *American Scientist 95* (2007), 476–480.

[21] HILDERINK, G., BROENINK, J., VERVOERT, W., AND BAKKERS, A. Communicating java threads. In *Proceedings of the 20th World Occam and Transputer User Group Technical Meeting* (The Netherlands, 1997), IOS Press, pp. 48–76.

[22] HINSEN, K. Parallel scripting with python. *Computing in Science & Engineering 9*, 6 (November/December 2007), 82–89.

[23] HOARE, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 8 (1978), 666–677.

[24] HOARE, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[25] JACKSON, K. R. pyglobus: a python interface to the globus toolkit. *Concurrency and Computation: Practice and Experience 14* (2002), 1075–1083.

[26] LIMITED, S.-T. M. *Occam 2 Reference Manual*, 2.1 ed. Prentice-Hall International Ltd, United Kingdom, May 1995.

[27] LONGO, R. Cocopy 1.1.0rc [online]. November 2007. Available from: `http://pypi.python.org/pypi/CocoPy/`, Accessed 31 October 2008.

[28] MARTELLI, A. *Python in a Nutshell*. O'Reilly & Associates, Inc., 2003.

[29] MCDONALD, I. Cpu architecture and operation [online]. July 1997. Available from: `http://www.dcs.gla.ac.uk/~ian/project3/node13.html`, Accessed 28 October 2008.

[30] MCGUIRE, P. Pyparsing wiki home [online]. October 2008. Available from: `http://pyparsing.wikispaces.com/`, Accessed 31 October 2008.

[31] MILLER, P. J. Parallel, distributed scripting with python.

[32] MOREIRA, N., AND REIS, R. Yappy - yet another lr(1) parser generator for python [online]. October 2006. Available from: `http://www.ncc.up.pt/FAdo/Yappy/Yappy.html`, Accessed 31 October 2008.

[33] PARR, T. J. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, Raleigh, North Carolina, 2007.

[34] PATEL, A. Parsing with yapps [online]. 2008. Available from: `http://theory.stanford.edu/~amitp/yapps/`, Accessed 31 October 2008.

[35] PEDRONI, S. Pypy - goals and architecture overview. Online, March 2007. Available from: `http://codespeak.net/pypy/dist/pypy/doc/architecture.html`.

[36] PÉREZ, F. Ipython: An enhanced interactive python shell [online]. March 2008. Available from: `http://ipython.scipy.org/moin/`, Accessed 25 May 2008.

[37] PYTHON.ORG. About python [online]. May 2008. Available from: `http://www.python.org/about/`, Accessed 25 May 2008.

[38] PYTHON.ORG. Python library and extension faq [online]. January 2008. Available from: `http://www.python.org/doc/faq/library/`, Accessed 29 October 2008.

[39] RAJU, V., RONG, L., AND STILES, G. *Automatic Conversion of CSP to CTJ, JCSP, and CCSP*. IOS Press, 2003.

[40] RIGO, A. Psyco introduction [online]. January 2008. Available from: `http://psyco.sourceforge.net/introduction.html`, Accessed 28 October 2008.

[41] SCATTERGOOD, B. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, University of Oxford, Trinity, 1998.

[42] SUTTER, H. A fundamental turn toward concurrency in software. *Dr. Dobb's 30* (March 2005).

[43] SUTTER, H., AND LARUS, J. Software and the concurrency revolution. *Queue 3*, 7 (2005), 54–62.

[44] TERRY, P. *Compiling with C# and Java*. Addison-Wesley, 2005.

[45] VAN GLABBEEK, R. Process algebra. Online, October 2008.

[46] VOSS, J. Wisent: a python parser generator [online]. March 2008. Available from: `http://seehuhn.de/pages/wisent`, Accessed 31 October 2008.

Appendix A

Grammar listings

This appendix presents the ANTLR grammars and StringTemplate code templates used to construct the parser and code generator for Hydra. The listings provided below are not the full listings as much of the grammar action code has been removed for readability.

A.1 ANTLR Parser Grammar for CSP

```
1  grammar csp;
2
3  options
4  {
5      language=Python;
6      output=AST;
7      ASTLabelType=CommonTree;
8      backtrack=true;
9      memoize=true;
10 }
11
12 /*------------------------------------------------------------------
13 * PARSER RULES
14 *------------------------------------------------------------------*/
15 // $<Productions
16
17 program returns [procs, chans]
18         scope
19         {
20             vars;
21         }
22         @init
23         {
24             imprts = False
25         }
26         : (pythonimport { imprts = True } )? command_list
27             -> { imprts == True }?      ^(PROGRAM pythonimport command_list)
```

```
28              ->                          ^(PROGRAM EMPTY command_list)
29          ;
30
31  pythonimport
32          : PYIMPRT+
33              -> ^(PYTHON PYIMPRT)+
34          ;
35
36  parallel
37          : LPARA process (PAR process)* RPARA
38              -> ^(PARALLEL process+)
39          ;
40
41  process
42          : process_label command_list
43              -> ^(PROCESS process_label command_list)
44          ;
45
46  process_label returns [lbl]
47          @init
48          {
49              subsup = False
50          }
51          : ID
52          |   {
53                  self.anonproc += 1
54                  $lbl = ('__anonproc_' + str(self.anonproc), False, 0, None)
55              }
56              -> ^(PROC_LABEL EMPTY EMPTY)
57          ;
58
59  label_subscript returns [subs]
60          : int_const
61              -> ^(SUBSCRIPT int_const)
62          | range
63              -> ^(SUBSCRIPT range)
64          ;
65
66  declaration
67          : ids+=ID (COMMA ids+=ID)* COLON type SEMICOL
68              -> ^(VARDEF type ^(IDENT ID))+
69          ;
70
71  int_const
72          : simple_expr
73              -> simple_expr
74          ;
75
76  range returns [rn]
77          @init
78          {
79              idsup = False
80          }
81          : (ID COLON { idsup = True })? lower=int_const DBLCOM upper=int_const
82              -> { idsup == True }?       ^(RANGE ^(VARDEF INTEGER ^(IDENT ID)) $lower $upper)
83              -> { idsup == False }?      ^(RANGE $lower $upper)
```

```
84              ->                          ^(RANGE $lower $upper)
85          ;
86
87  type returns [tp]
88          : (LPAREN lower=INT DBLCOM upper=INT RPAREN) basictype
89              -> ^(ARRAY ^(RANGE $lower $upper) basictype)
90          | basictype
91              -> basictype
92          ;
93
94  basictype
95          : 'integer'                                  -> INTEGER
96          | 'boolean'                                  -> BOOLEAN
97          | 'char'                                     -> CHAR
98          ;
99
100 command_list
101         : declaration* command+
102             -> ^(COMMAND_LIST declaration* command+)
103         ;
104
105 command
106         : (
107         simple_cmd                                  -> simple_cmd
108         | struct_cmd                                -> struct_cmd
109         ) SEMICOL
110         ;
111
112 simple_cmd
113         : assignment                                -> ^(COMMAND assignment)
114         | input_cmd                                 -> ^(COMMAND input_cmd)
115         | output_cmd                                -> ^(COMMAND output_cmd)
116         | nullcmd                                   -> ^(COMMAND nullcmd)
117         | PYEXPR                                    -> ^(COMMAND ^(PYTHON PYEXPR))
118         ;
119
120 struct_cmd
121         : alternative                               -> ^(COMMAND alternative)
122         | repetitive                                -> ^(COMMAND repetitive)
123         | parallel                                  -> ^(COMMAND parallel)
124         ;
125
126 nullcmd
127         : 'SKIP'                                    -> SKIP
128         ;
129
130 assignment
131         : target_var EQUAL expression
132             -> ^(ASSIGN target_var expression)
133         ;
134
135 process_name returns [pn]
136         : ID
137             -> ^(PROC_LABEL ID EMPTY)
138         ;
139
```

```
140 subscripts
141         : subs+=simple_expr (COMMA subs+=simple_expr)*
142             -> ^(SUBSCRIPT simple_expr+)
143         ;
144
145 target_var returns [tp, stp]
146         @init
147         {
148             arrsub = False
149         }
150         : ID
151         (LBRACK int_const RBRACK
152             {
153                 arrsub = True
154             }
155         )?
156             -> { arrsub == True }?      ^(IDENT ID int_const)
157             ->                          ^(IDENT ID)
158         | struct_target
159             -> struct_target
160         ;
161
162 constructor     returns [ident]
163         : ID
164             -> ^(IDENT ID)
165         |
166             -> EMPTY
167         ;
168
169 struct_target returns [tp, stp]
170         : constructor LPAREN var_list RPAREN
171             -> ^(STRUCT_T constructor var_list)
172         ;
173
174 var_list returns [tps]
175         : tv1=target_var (COMMA tv2=target_var)*
176             -> target_var+
177         |
178             -> EMPTY
179         ;
180
181 simple_expr returns [tp]
182         @init
183         {
184             arrsub = False
185         }
186         : ID (LBRACK int_const RBRACK)?
187
188             -> {$tp == 'integer' and arrsub}?           ^(EXPRESSION ^(VAR INTEGER ID int_const))
189             -> {$tp == 'boolean' and arrsub}?           ^(EXPRESSION ^(VAR BOOLEAN ID int_const))
190             -> {$tp == 'char' and arrsub}?              ^(EXPRESSION ^(VAR CHAR ID int_const))
191             -> {$tp == 'integer' and not arrsub}?       ^(EXPRESSION ^(VAR INTEGER ID))
192             -> {$tp == 'boolean' and not arrsub}?       ^(EXPRESSION ^(VAR BOOLEAN ID))
193             -> {$tp == 'char' and not arrsub}?          ^(EXPRESSION ^(VAR CHAR ID))
194             -> {$tp == 'array-integer' and not arrsub}? ^(EXPRESSION ^(VAR ^(ARRAY INTEGER) ID))
195             -> {$tp == 'array-boolean' and not arrsub}? ^(EXPRESSION ^(VAR ^(ARRAY BOOLEAN) ID))
```

```
196             -> {$tp == 'array-char' and not arrsub}?    ^(EXPRESSION ^(VAR ^(ARRAY CHAR) ID))
197             ->                                          ^(EXPRESSION ^(VAR INVALID ID))
198         | INT
199             -> ^(EXPRESSION ^(VALUE INTEGER INT))
200         | BOOL
201             -> ^(EXPRESSION ^(VALUE BOOLEAN BOOL))
202         | CHR
203             -> ^(EXPRESSION ^(VALUE CHAR CHR))
204         | PYEXPR
205             -> ^(EXPRESSION ^(PYTHON PYEXPR))
206         ;
207
208 struct_expr returns [tp, stp]
209         : constructor LPAREN expr_list RPAREN
210             -> ^(STRUCT_E constructor expr_list)
211         ;
212
213 expr_list returns [tps]
214         : ex1=expression (COMMA ex2=expression)*
215             -> expression+
216         |
217             -> EMPTY
218         ;
219
220 expression returns [tp, stp]
221         : simple_expr
222             -> simple_expr
223         | struct_expr
224             -> struct_expr
225         ;
226
227 input_cmd
228         : process_name QUEST target_var
229             -> ^(INPUT process_name target_var)
230         ;
231
232 output_cmd
233         : process_name EXCLAM expression
234             -> ^(OUTPUT process_name expression)
235         ;
236
237 repetitive
238         : ASTER alternative
239             -> ^(REPETITIVE alternative)
240         ;
241
242 alternative
243         : LBRACK guarded ( GBRACK guarded )* RBRACK
244             -> ^(ALTERNATIVE guarded+)
245         ;
246
247 guarded
248         @init
249         {
250             rngsup = False
251         }
```

```
252         : (LPAREN range RPAREN { rngsup = True })? guard GARROW command_list
253                 -> { rngsup == True }? ^(GUARDED range guard command_list)
254             ->                         ^(GUARDED EMPTY guard command_list)
255         ;
256
257 guard
258         : guardlist
259             -> ^(GUARD guardlist)
260         | input_cmd
261             -> ^(GUARD input_cmd)
262         | nullcmd
263             -> ^(GUARD nullcmd)
264         ;
265
266 guardlist
267         : frst=guard_elem
268         ( SEMICOL follow=guard_elem)* ( SEMICOL input_cmd )?
269             -> ^(GUARD_LIST guard_elem+ input_cmd?)
270         ;
271
272 guard_elem returns [tp]
273         : simple_expr
274             -> simple_expr
275         | declaration
276             -> declaration
277         ;
278
279 // $>
280
281 /*------------------------------------------------------------------
282 * LEXER RULES
283 *------------------------------------------------------------------*/
284
285 // $<Lexer Tokens
286
287 LBRACK  : '[';
288 RBRACK  : ']';
289 LPAREN  : '(';
290 RPAREN  : ')';
291 GBRACK  : '[]';
292 GARROW  : '->';
293 LPARA   : '[[';
294 RPARA   : ']]';
295 PAR     : '||';
296 SEMICOL : ';';
297 PROCCOL : '::';
298 ASTER   : '*';
299 EQUAL   : ':=';
300 QUEST   : '?';
301 EXCLAM  : '!';
302 COMMA   : ',';
303 COLON   : ':';
304 DBLCOM  : '..';
305 ID      : ('a'..'z') ('a'..'z'| 'A'..'Z'| '0'..'9'| '_')*;
306 CHR     : '\'' (options {greedy = false;} : . ) '\'';
307 INT     : ('0'..'9')+;
```

```
BOOL    : 'True' | 'False';
PYEXPR  : '{' (options {greedy=false;} : . )* '}';
PYIMPRT : '_include' '{'(options {greedy=false;} : . )* '}';
WS      : ( '\t' | ' ' | '\u000C' )+    { $channel = HIDDEN; } ;
COMMENT : ('--' (options {greedy = false;} : . )* CRLF) { $channel = HIDDEN; } ;
CRLF    : ('\r'? '\n') { $channel = HIDDEN; } ;

// $>
```

A.2 ANTLR Tree Walker Grammar for CSP

```
tree grammar cspWalker;

options
{
    language=Python;
    tokenVocab=csp;
    ASTLabelType=CommonTree;
    output=template;
}

@members
{
    keywords = ['and','del','from','not','while','as','elif','global','or','with','assert','else','if',
                'pass','yield','break','except','import','print','class','exec','in','raise','continue',
                'finally','is','return','def','for','lambda','try']

    def fixkeywords(self, inp):
        if inp in self.keywords:
            return "_"+inp
        return inp

    def arrayinit(self, a, b):
        return '[' + ''.join(['None, ' for x in range(max(int(a),int(b)) + 1)]) + ']'

    def buildprocargs(self, procnames):
        results = ''
        nl = False
        for p in procnames:
            if nl:
                results += '\n'
            results += 'elif _proc_ == "' + p + '":\n'
            results += '\tSequence(' + p + '())'
            nl = True
        return results

    def definechan(self, chanlist):
        results = ''
        for cn, cv in chanlist:
            results += cv + ' = getNamedChannel("' + cn + '")\n'
        return results

    def buildalts(self, inps):
        results = ''
```

```
44          for i in inps:
45              results += i + '_alt = Alternative(' + i + '.read, __sg).select()\n'
46          return results
47
48      anonproc = 0
49      repidx = 0
50  }
51
52  program returns [channels, procs]
53                  scope
54                  {
55                      chans;
56                  }
57                  @init
58                  {
59                      pnames = []
60                      $program::chans = set()
61                  }
62                  @after
63                  {
64                      $channels = list($program::chans)
65                      $procs = pnames
66                  }
67                  : ^(PROGRAM EMPTY cmdlst=command_list[True] { pnames = $command_list.pnames })
68                      -> program(commandlist={cmdlst}, procargs={self.buildprocargs(pnames)})
69                  | ^(PROGRAM (pyi+=pythonimport)+ cmdlst=command_list[True] { pnames = $command_list.pnames
70                      -> program(commandlist={cmdlst}, procargs={self.buildprocargs(pnames)}, incl={$pyi})
71                  ;
72
73  pythonimport
74                  : ^(PYTHON PYIMPRT)
75                      -> imports(imp={($PYIMPRT.text)[9:-1]})
76                  ;
77
78  parallel [toplvl] returns [procnames]
79                  @init
80                  {
81                      $procnames = []
82                      tsprocs = []
83                  }
84                  : ^(PARALLEL (procs+=process
85                      {
86                          if $toplvl:
87                              $procnames.append($process.procname)
88                          else:
89                              tsprocs.append($process.procname)
90                      }
91                     )+)
92                      -> { $toplvl }?  parallel(proc={$procs}, procnames={tsprocs}, lvl={False})
93                      ->               parallel(proc={$procs}, procnames={tsprocs}, lvl={True})
94                  ;
95
96  process returns [procname]
97                  scope
98                  {
99                      chans;
```

```
100                     prcname;
101                 }
102                 @init
103                 {
104                     $process::chans = set()
105                     $process::prcname = ''
106                 }
107                 : ^(PROCESS lbl=process_label { $process::prcname = $process_label.procname } cmdlst=commar
108                     {
109                         $procname = $process_label.procname
110                     }
111                     -> process(label={lbl}, commandlist={cmdlst}, chans={self.definechan(list($process::cha
112                 ;
113
114 process_label returns [procname]
115                 : ^(PROC_LABEL ID label_subscript)
116                     {
117                         $procname = self.fixkeywords($ID.text)
118                     }
119                     -> process_label(ident={self.fixkeywords($ID.text)})
120                 | ^(PROC_LABEL ID EMPTY)
121                     {
122                         $procname = self.fixkeywords($ID.text)
123                     }
124                     -> process_label(ident={self.fixkeywords($ID.text)})
125                 | ^(PROC_LABEL EMPTY EMPTY)
126                     {
127                         self.anonproc += 1
128                         $procname = u'__anonproc_' + str(self.anonproc)
129                     }
130                     -> process_label(ident={u'__anonproc_' + str(self.anonproc)})
131                 ;
132
133 label_subscript : ^(SUBSCRIPT int_const)
134                 | ^(SUBSCRIPT range)
135                 ;
136
137 declaration     : ^(VARDEF tp=type ^(IDENT ID))
138                     -> declaration(ident={self.fixkeywords($ID.text)}, typdef={tp})
139                 ;
140
141 int_const       : se=simple_expr
142                     -> int_const(exp={se})
143                 ;
144
145 range           : ^(RANGE ^(VARDEF INTEGER ^(IDENT ID)) int_const int_const)
146                 | ^(RANGE int_const int_const)
147                 ;
148
149 type            : ^(ARRAY ^(RANGE a=INT b=INT) basictype)
150                     -> type_def(def={self.arrayinit(a.text,b.text)})
151                 | basictype
152                     -> type_def(def={'None'})
153                 ;
154
155 basictype       : INTEGER
```

```
156                 | BOOLEAN
157                 | CHAR
158                 ;
159
160 command_list [toplvl] returns [pnames]
161                 @init
162                 {
163                     $pnames = []
164                 }
165                 : ^(COMMAND_LIST (declarations+=declaration)* (commands+=command[$toplvl] { $pnames.extend(
166                     -> command_list(vardefs={$declarations}, commands={$commands})
167                 ;
168
169 command [toplvl] returns [procnames]
170                 @init
171                 {
172                     $procnames = []
173                 }
174                 : (
175                   ^(COMMAND cmdln=assignment)
176                 | ^(COMMAND cmdln=input_cmd)
177                 | ^(COMMAND cmdln=output_cmd)
178                 | ^(COMMAND cmdln=nullcmd)
179                 | ^(COMMAND cmdln=alternative)
180                 | ^(COMMAND cmdln=repetitive)
181                 | ^(COMMAND cmdln=parallel[$toplvl] { $procnames = $parallel.procnames } )
182                 | ^(COMMAND cmdln=py_command)
183                 )       -> command(cmd={cmdln})
184                 ;
185
186 py_command      : ^(PYTHON PYEXPR)
187                     -> py_exec(code={($PYEXPR.text)[1:-1]})
188                 ;
189
190 nullcmd         : SKIP
191                     -> nullcommand()
192                 ;
193
194 assignment      : ^(ASSIGN tgt=target_var expr=expression)
195                     -> assignment(target={tgt}, value={expr})
196                 ;
197
198 process_name returns [procname]
199                 : ^(PROC_LABEL ID subs=subscripts)
200                     {
201                         $procname = self.fixkeywords($ID.text)
202                     }
203                     -> proc_name(ident={self.fixkeywords($ID.text)}, sub={subs})
204                 | ^(PROC_LABEL ID EMPTY)
205                     {
206                         $procname = self.fixkeywords($ID.text)
207                     }
208                     -> proc_name(ident={self.fixkeywords($ID.text)}, sub={[0]})
209                 ;
210
211 subscripts      : ^(SUBSCRIPT (se+=simple_expr)+)
```

```
212                     -> subscript(exp={se})
213                 ;
214
215 target_var      : ^(IDENT ID)
216                     -> target_var(ident={self.fixkeywords($ID.text)})
217                 | ^(IDENT ID ic=int_const)
218                     -> target_var(ident={self.fixkeywords($ID.text)}, sub={ic})
219                 | st=struct_target
220                     -> target_var(ident={st})
221                 ;
222
223 constructor     : ^(IDENT ID)
224                 | EMPTY
225                 ;
226
227 struct_target   : ^(STRUCT_T constructor vl=var_list)
228                     -> struct_targ(targlist={vl})
229                 ;
230
231 var_list        : (tv+=target_var)+
232                     -> targ_list(targs={$tv})
233                 | EMPTY
234                     -> targ_list(targs={'___ignored___'})
235                 ;
236
237 expression      : (
238                   ex=simple_expr
239                 | ex=struct_expr
240                 ) -> expression(expr={ex})
241                 ;
242
243 simple_expr     : ^(EXPRESSION ^(VAR INTEGER ID))
244                     -> simple_expr(val={self.fixkeywords($ID.text)})
245                 | ^(EXPRESSION ^(VAR BOOLEAN ID))
246                     -> simple_expr(val={self.fixkeywords($ID.text)})
247                 | ^(EXPRESSION ^(VAR CHAR ID))
248                     -> simple_expr(val={self.fixkeywords($ID.text)})
249                 | ^(EXPRESSION ^(VAR INVALID ID))
250                     -> simple_expr(val={self.fixkeywords($ID.text)})
251                 | ^(EXPRESSION ^(VAR ^(ARRAY INTEGER) ID))
252                     -> simple_expr(val={self.fixkeywords($ID.text)})
253                 | ^(EXPRESSION ^(VAR ^(ARRAY BOOLEAN) ID))
254                     -> simple_expr(val={self.fixkeywords($ID.text)})
255                 | ^(EXPRESSION ^(VAR ^(ARRAY CHAR) ID))
256                     -> simple_expr(val={self.fixkeywords($ID.text)})
257                 | ^(EXPRESSION ^(VAR INTEGER ID ic=int_const))
258                     -> simple_expr(val={self.fixkeywords($ID.text)}, sub={ic})
259                 | ^(EXPRESSION ^(VAR BOOLEAN ID ic=int_const))
260                     -> simple_expr(val={self.fixkeywords($ID.text)}, sub={ic})
261                 | ^(EXPRESSION ^(VAR CHAR ID ic=int_const))
262                     -> simple_expr(val={self.fixkeywords($ID.text)}, sub={ic})
263                 | ^(EXPRESSION ^(VALUE INTEGER INT))
264                     -> simple_expr(val={$INT})
265                 | ^(EXPRESSION ^(VALUE BOOLEAN BOOL))
266                     -> simple_expr(val={$BOOL})
267                 | ^(EXPRESSION ^(VALUE CHAR CHR))
```

```
268                     -> simple_expr(val={$CHR})
269                 | ^(EXPRESSION ^(PYTHON PYEXPR))
270                     -> python_expr(expr={($PYEXPR.text)[1:-1]})
271                 ;
272
273 struct_expr     : ^(STRUCT_E constructor explst=expr_list)
274                     -> struct_expr(exprlist={explst})
275                 ;
276
277 expr_list       : (exprs+=expression)+
278                     -> expr_list(exprs={$exprs})
279                 | EMPTY
280                     -> expr_list(exprs={[]})
281                 ;
282
283 input_cmd       : ^(INPUT sn=process_name tv=target_var)
284                     {
285                         channm = $process_name.procname + '->' + $process::prcname
286                         $process::chans.add((channm, '__chan_' + $process_name.procname + '_in'))
287                         $program::chans.add(channm)
288                     }
289                     -> input_cmd(cname={channm}, sname={sn}, targ={tv})
290                 ;
291
292 output_cmd      : ^(OUTPUT dn=process_name ex=expression)
293                     {
294                         channm = $process::prcname + '->' + $process_name.procname
295                         $process::chans.add((channm, '__chan_' + $process_name.procname + '_out'))
296                         $program::chans.add(channm)
297                     }
298                     -> output_cmd(cname={channm}, dname={dn}, exp={ex})
299                 ;
300
301 repetitive      : ^(REPETITIVE alt=alternative)
302                     {
303                         self.repidx += 1
304                     }
305                     -> repetitive(altern={alt}, idx={self.repidx})
306                 ;
307
308 alternative
309                 @init
310                 {
311                     inpgrdlst = []
312                 }
313                 : ^(ALTERNATIVE
314                     (grded+=guarded
315                         {
316                             if not $guarded.inpgrd is None:
317                                 inpgrdlst.append($guarded.inpgrd)
318                         }
319                     )+)
320                     -> alternative(inpguards={self.buildalts(inpgrdlst)}, guarded={$grded})
321                 ;
322
323 guarded returns [inpgrd]
```

```
324                 : ^(GUARDED range grd=guard cmdlst=command_list[False])
325                     {
326                         $inpgrd = $guard.inpgrd
327                     }
328                     -> guarded(guard={grd}, commandlist={cmdlst})
329                 | ^(GUARDED EMPTY grd=guard cmdlst=command_list[False])
330                     {
331                         $inpgrd = $guard.inpgrd
332                         inppres = (not $guard.targ is None)
333                     }
334                     -> {inppres}?       guarded(guard={grd}, commandlist={cmdlst}, target={$guard.targ}, ir
335                     ->                          guarded(guard={grd}, commandlist={cmdlst})
336                 ;
337
338 guard returns [inpgrd, targ]
339                 @init
340                 {
341                     $inpgrd = None
342                     $targ = None
343                 }
344                 : ^(GUARD grdlst=guardlist)
345                     {
346                         $inpgrd = $guardlist.inpgrd
347                         $targ = $guardlist.targ
348                     }
349                     -> guard(guardlist={grdlst})
350                 | ^(GUARD grdlst=skip_grd)
351                     -> guard(guardlist={grdlst})
352                 | ^(GUARD grdlst=input_grd)
353                     {
354                         $inpgrd = $input_grd.cn
355                         $targ = $input_grd.targ
356                     }
357                     -> guard(guardlist={grdlst})
358                 ;
359
360 guardlist returns [inpgrd, targ]
361                 @init
362                 {
363                     $inpgrd = None
364                     $targ = None
365                 }
366                 : ^(GUARD_LIST (grdelems+=guard_elem)+
367                    (input_grd
368                        {
369                            $inpgrd = $input_grd.cn
370                            $targ = $input_grd.targ
371                        }
372                    )?)
373                     -> guardlist(elems={$grdelems})
374                 ;
375
376 guard_elem              : ex=simple_expr
377                     -> expression(expr={ex})
378                 | declaration
379                     -> simple_expr(val={True})
```

```
380                 ;
381
382 input_grd returns [cn, targ]
383                 : ^(INPUT sn=process_name tv=target_var)
384                     {
385                         channm = $process_name.procname + '->' + $process::prcname
386                         $process::chans.add((channm, '__chan_' + $process_name.procname + '_in'))
387                         $program::chans.add(channm)
388                         $cn = '__chan_' + $process_name.procname + '_in'
389                         cnalt = $cn + '_alt'
390                         $targ = tv
391                     }
392                     -> inputguard(chan={cnalt})
393                 ;
394
395 skip_grd        : SKIP
396                     -> skipguard()
397                 ;alternative     : ^(ALTERNATIVE (grded+=guarded)+)
398                     -> alternative(guarded={$grded})
399                 ;
```

A.3 Extracts from the StringTemplate Group File

```
1  group hydra;
2
3  program(incl, commandlist, procargs) ::=
4  <<
5  import sys
6  import time
7  from pycsp import *
8  from pycsp.plugNplay import *
9  from pycsp.net import *
10 <if(incl)>
11 <incl; separator="\n">
12 <endif>
13
14 def __program(_proc_):
15         <commandlist>
16
17         # process spawning
18         if False:
19                 pass
20         <procargs>
21         else:
22                 print 'Invalid process specified.'
23
24 __program(sys.argv[1])
25 >>
26 imports(imp) ::= "<imp>"
27 parallel(proc, procnames, lvl) ::=
28 <<
29 <proc; separator="\n">
30 <if (lvl)>
31 Parallel(<procnames; separator="(), ">())
```

```
<endif>

>>

process(label, commandlist, chans) ::=
<<
<label>
        <chans>
        <commandlist>

>>

process_label(ident) ::=
<<

@process
def <ident>():
        __procname = '<ident>'
        print '# <ident>'

>>

command_list(vardefs, commands) ::=
<<
<vardefs; separator="\n">
<commands; separator="\n">
>>

repetitive(altern, idx) ::=
<<

__lctrl_<idx> = True
while(__lctrl_<idx>):
        <altern>
        else:
                __lctrl_<idx> = False
>>

alternative(inpguards, guarded) ::=
<<
__sg = Skip()
<inpguards>
if False:
        pass
<guarded>
>>

guarded(guard, commandlist, target, inpgrd) ::=
<<
elif <guard>:
<if(target)>
        <target> = <inpgrd>_alt()
<endif>

        <commandlist>

```

```
>>

guard(guardlist) ::= "<guardlist>"
guardlist(elems) ::= "<elems; separator=\" and \">"
guard_elem(elem) ::= "<elem>"
skipguard() ::= "True"
inputguard(chan) ::= "not <chan> == __sg"
declaration(ident, typdef) ::= "<ident> = <typdef>"
type_def(def) ::= "<def>"
command(cmd) ::= "<cmd>"
assignment(target, value) ::= "<target> = <value>"
nullcommand()   ::= "pass"
int_const(exp) ::= "<exp>"
expression(expr) ::= "<expr>"
simple_expr(val,sub) ::= "<val><if(sub)>[<sub>]<endif>"
struct_expr(exprlist) ::= "(<exprlist>,)"
target_var(ident, sub) ::= "<ident><sub>"
struct_targ(targlist) ::= "<targlist>"
python_expr(expr) ::= "eval('<expr>')"
expr_list(exprs) ::= "<exprs; separator=\",\">"
targ_list(targs) ::= "<targs; separator=\",\">"
proc_name(ident, sub)::= "<ident>"
subscript(exp) ::= "<exp>"
py_exec(code) ::= "exec '<code>' in globals(), locals()"

input_cmd(cname, sname, targ, grd) ::=
<<
<if(grd)>
__chan_<sname>_in.read
<else>
<targ> = __chan_<sname>_in.read()
<endif>
>>

output_cmd(dname, exp) ::=
<<
__chan_<dname>_out.write(<exp>)
>>
```

Appendix B

Project Poster

This appendix presents the project poster, which was submitted as one of the required project deliverables.

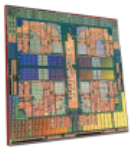
Hydra: A Python Extension for Parallelism

Waide Tristram
Supervisor: Dr. Karen Bradshaw
Email: g05t1067@campus.ru.ac.za
Website: http://www.cs.ru.ac.za/research/g05t1067/

RHODES UNIVERSITY
Where leaders learn

Problem

There has been a significant increase in the availability of multi-core CPUs over the past few years, with AMD and Intel offering dual-core and quad-core processors at affordable prices.

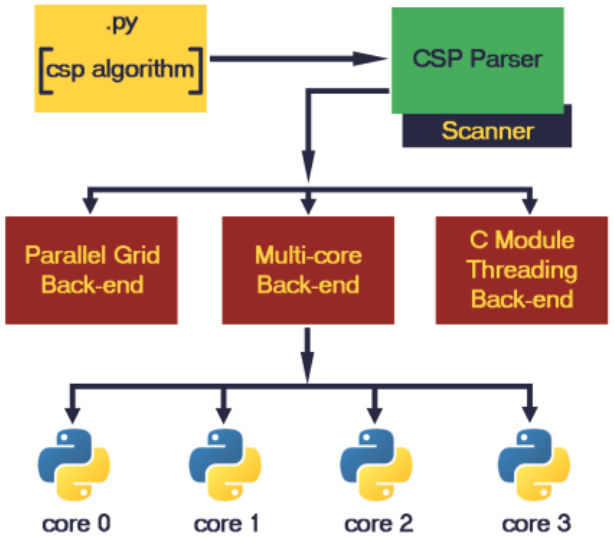


The vast majority of software doesn't make use of this parallel power so there is a need for tools that make parallel software development easier.

However, many of the libraries for creating concurrent software are lacking when it comes to ease-of-use for the average programmer. And languages like Python have other limitations that prevent parallel execution.

Solution

Hydra provides an easy-to-use framework that takes an algorithm defined in a simple concurrent notation and generates the appropriate parallel code.



sponsored by

Telkom
COMVERSE
THRIP
TECHNOLOGY AND HUMAN RESOURCES FOR INDUSTRY PROGRAMME
STORTECH
For Your Information
amatole
telecommunication services
tellabs
Bright Ideas Projects 39
openVOICE

Hydra - *n* heads are better than one

Appendix C

CD Contents

The accompanying CD contains all the necessary resources to reproduce this research and assess the findings.

Code:

This folder contains all the code that makes up the Hydra project.

Documents:

This folder contains all the documents produced during the course of the project.

Resources:

This folder contains all the offline versions of references that were accessed online.