Enabling Go Program Analysis in Rascal

Luke Swearngan, Mark Hills
Appalachian State University, Boone, NC, USA
swearnganjl@appstate.edu, hillsma@appstate.edu

***Abstract*—In this paper, we present early work on the foundation of a program analysis framework for the Go programming language. This framework, named Go AiR (Analysis in Rascal), currently provides support for extracting abstract syntax trees from Go source code, working with multiple Go projects and multiple versions of a single Go project (based on Git version history), and code exploration through the use of Rascal features for working with representations of source code. We discuss the structure of the framework, describe the mapping from Go to Rascal and how this was tested, present sample code showing how the framework can be used to explore Go source code, and discuss future extensions to the framework to support program analysis and verification.**

***Index Terms*—Go, static analysis, analysis frameworks**

I. Introduction

The Go programming language was first released in late 2009. The language was designed by researchers at Google with the goal of creating a statically typed, fast, and understandable language that could support the kinds of networked and multiprocessing applications that were becoming increasingly common. Since its release, it has been used as the primary language in a number of well-known applications, including Docker (container management), Kubernetes (cluster management), Hugo (static site generation), and Terraform (cloud infrastructure provisioning). As of July 2023, it is the 13th most popular language on the Tiobe Index [1], which measures interest in programming languages, and the 14th most popular in the RedMonk programming language rankings [2], which uses a ranking formula that aggregates GitHub and Stack Overflow data. In GitHub's Octoverse 2022 report [3], it is in 5th place in the list of languages with the fastest growth.

Go is well known for it's message-passing model of concurrency. This model, based on earlier work on language calculii such as Hoare's Communicating Sequential Processes (CSP) [4] and Milner's Calculus of Communicating Systems (CCS) [5] and the π-calculus [6], [7], uses channels to allow for message exchange between different threads, known as *goroutines* in Go. Channels can be either synchronous or asynchronous—synchronous channels allow for one message to be sent at a time, with the sender waiting for a receiver to read the message from the channel, while asynchronous channels include buffers that do not require the sender to wait (if the buffer still has room for another message) or the receiver to wait (if messages are present in the buffer to read).

In this paper, we present early work on the foundation of a program analysis framework for the Go programming language. This framework, written in the Rascal meta-programming language [8], [9], is named Go AiR, i.e., Go **A**nalysis **i**n **R**ascal. Rascal provides a number of features that make it well-suited for program analysis, including a rich collection of built-in types such as tuples, sets, maps, and relations; powerful pattern-matching capabilities over Rascal terms; a language-level notion of fixpoint computation; and extensibility through libraries written in Java. Go AiR currently provides support for extracting and serializing abstract syntax trees from Go source code, working with multiple Go projects and multiple versions of a single Go project (based on Git version history), and code exploration through the use of Rascal features for working with representations of source code. Our goal is that this framework can provide a foundation for future work on static analysis and verification of Go code, either through analysis algorithms written directly in Rascal or by using existing links [10] to formal systems, such as rewriting logic [11]–[13], a logic of concurrency, and the K framework [14], [15], a framework for the formal definition of programming languages and programming language tools.

The rest of this paper is organized as follows. In Section II, we describe how the ASTs provided by the Go language are mapped to AST constructors and datatypes in Rascal. Section III then discusses the actual AST extraction process as part of an overview of the Go AiR framework. Following this, Section IV focuses on how we have tested the AST extraction process to ensure it is generating ASTs which can be loaded into Rascal. After this, Section V presents an extended example of using the framework to identify specific concurrency features (in this case, channel creation) in Go source code, comparing with results found in earlier work [16]. Section VI then discusses earlier work related to the work presented in this paper. Finally, Section VII concludes and discusses our future goals for this framework. Both Go AiR [17] and the AST extraction tool, go2rascal [18], are available on GitHub.

II. Mapping Go to Rascal

Go includes standard libraries for parsing Go source code and representing the resulting abstract syntax trees (ASTs). These include the `go/token` library, which defines the lexical tokens in the language (e.g., literals, keywords); the `go/ast` library, which defines Go structures (*structs*) to represent AST nodes; and the `go/parser` library, which includes functions that parse Go code and return the matching ASTs.

When a Go source file is parsed, the parser returns a value of type `File`. This is the top-level node in the generated AST, and represents the contents of the entire file. Figure 1 shows a simple Hello, World program in Go. In this case, the `File` value would include the package name (`main`) and two

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, World!")
}
```

Fig. 1. Hello World, in Go.

declarations: an import declaration for the `fmt` package, and a function declaration for the `main` function. AST values also include position information, which allows each node to be mapped back to the underlying construct in the source code.

As a simple example of mapping from Go to Rascal, the AST structs for a unary expression, a selector expression (which selects a field from a value), and an identifier expression are shown in Figure 2. These are defined in the `go/ast` library in file `ast.go`. In Go, the name of the structure comes before `struct` (e.g., `UnaryExpr`), then the fields are listed between the braces, with the field name (e.g., `X`) followed by the field type (e.g., `Expr`). A `UnaryExp` includes the operand (`X`), the operator being used (`Op`), and the location of this operator in the source file (`OpPos`). A `SelectorExp` includes the expression (`X`) that yields a value which should have a field named `Sel`. `Sel` is of type `*Ident` (i.e., pointer to `Ident`). `Ident` includes the identifier name (`Name`) as a string.

An example that uses these AST types is `&opts.verbose`, which is a use of the address of operator `&` on a selector expression, `opts.verbose`, meaning the expression should return the address of the field `verbose` in value `opts`. `opts` itself is an identifier expression, as is `verbose`. The Rascal AST representation of this is:

```
unaryExpr(selectorExpr(ident("opts"),
  "verbose"),and())
```

This uses the `Expr` (expression) datatype and the `ident`, `unaryExpr`, and `selectorExpr` constructors, shown in Figure 3. The `ident` constructor includes a field `name` of type `str` (string), while both `unaryExpr` and `selectorExpr` have fields of type `Expr` to hold the operand and selector target, respectively. `unaryExpr` includes a field `operator` of type `Op`, which includes constructors for all the Go operators (including `and()`), while `selectorExpr` also includes a `str` field to hold the field name. Note that the `Expr` datatype also defines a field named `at`, available on all values of this type, that holds a source location pointing back to the location of the construct in the original source file. Since these can be quite long, we omit them here for conciseness.

```
UnaryExpr struct {
  OpPos  token.Pos
  Op     token.Token
  X      Expr
}
SelectorExpr struct {
  X    Expr
  Sel  *Ident
}
Ident struct {
  NamePos  token.Pos
  Name     string
  Obj      *Object
}
```

Fig. 2. Structs for Unary and Selector Expressions and Identifiers, in Go.

```
data Expr(loc at=|unknown:///|)
  = ident(str name)
  | unaryExpr(Expr expr, Op operator)
  | selectorExpr(Expr expr, str selector)
  ...
```

Fig. 3. AST Constructors for Unary and Selector Expressions, in Rascal.

Figure 4 shows a more complex example. In Go, a range statement is used to iterate over an array, a slice (a data structure similar to an array, but without a fixed length, and with convenience functions for operations like append), a map, or a channel (used to communicate between goroutines). The `Key` and `Value` expressions indicate where to store the key and value (for a map), the index and value (for an array or slice), or just the value (for a channel) with each iteration, while `X` is the expression that yields the value being iterated over and `Body` is the body of the loop. `Tok` provides the assignment operator used in the statement. The Rascal version of the AST node is shown in Figure 5. Since both the key and value in the range expression can be `nil`, each are represented using an option type `OptionExpr` with two constructors, `someExpr`, when an expression is present, and `noExpr`, when the expression is `nil`. Also included are the assignment operator used in the statement (field `assignOp` of type `AssignOp`), the expression that yields the values being iterated over (field `rangeExpr`), and the statement representing the body of the loop (field `body`). Similarly to `Expr`, each `Stmt` also has a field named `at` that contains a source location.

In total, the abstract syntax in Rascal currently contains 21 different data types and 128 constructors, including several constructors used to help determine situations where there is no mapping on the Rascal side for a Go AST. This is to help

```
RangeStmt struct {
  For         token.Pos
  Key, Value  Expr
  TokPos      token.Pos
  Tok         token.Token
  Range       token.Pos
  X           Expr
  Body        *BlockStmt
}
```

Fig. 4. Struct for a Range Statement, in Go.

```
data Stmt(loc at=|unknown:///|)
= rangeStmt(OptionExpr keyOpt,
    OptionExpr valOpt, AssignOp assignOp,
    Expr rangeExpr, Stmt body)
```

Fig. 5. AST Constructor for Range Statement, in Rascal.

detect situations in the future where the Go language adds a new language construct and the Rascal AST definition needs to be extended, but has also been useful to help debugging during the construction of the existing mapping. Although the correctness of the mapping is currently being checked by hand, we have started to explore testing techniques that would provide a way to check this in an automated fashion.

III. Extracting Go ASTs

An overview of Go AiR, focused on AST extraction, code querying, and analysis, is shown in Figure 6. To start, a user of Go AiR can load either a single Go file or a directory containing a system made up of multiple files. In either case, Go AiR invokes the Go Parser/AST Printer (go2rascal) on each Go source file. go2rascal parses a Go source file and returns a string containing the Rascal AST term for the source file. The AST is structured based on the Rascal AST definition for Go included in Go AiR. By default, source locations are included with the AST nodes, linking the nodes back to the associated constructs in the parsed Go file. If Go AiR loaded a single Go file, the resulting type is `File`, which is the Rascal equivalent of the `File` struct in the Go AST. If Go AiR instead loaded a system, the resulting type is `System`, which is a collection of multiple `File`s, indexed by each file's location.

If multiple systems are being analyzed, Go AiR can also load all the systems provided in a directory. Each system in parsed, loaded, and serialized to disk for later analysis. Similarly, the different tagged versions of a single system, based on Git tags, can be individually loaded and serialized. These options allow exploration across different systems, as well as exploration across different versions of the same system. Serialization allows the parsing cost to only be incurred once, since loading the serialized ASTs is much faster.

Once loaded, Rascal code can be used to interactively explore the ASTs or extract empirical information from Go programs. A common use of this, based on our experience using similar tools such as PHP AiR [19], it to iteratively narrow queries to find specific uses of different features in the code—for instance, to find function calls, then calls of specific functions, then calls of specific functions with specific parameters. An example of this is shown in Section V. This helps with the other planned use, which is program analysis and verification. The expectation is that program analysis tasks are scripted for reproducibility, and to allow them to be used as part of other analysis tasks or code explorations. Although we plan to provide some core analyses (e.g., def/use information, control flow graph construction), these are still work in progress. The results of these program analysis tasks and explorations can be printed to the console, saved to file, or used to generate visualizations (e.g., by emitting a dot file for GraphViz).

IV. Testing AST Extraction

To test the process of mapping Go ASTs into Rascal, we used a corpus of 854 Go systems, based on a corpus used in earlier work on the use of channel-based concurrency constructs in Go [16]. The systems chosen for the corpus were, at the time (August 2018), the 900 most popular Go projects on GitHub (measured by how many people had starred the project), minus projects that the authors determined were not human-made, and minus several projects which are no longer available. Included are such well-known projects as Docker, Kubernetes, Hugo, Terraform (all mentioned in Section I), CockroachDB (a distributed SQL database), and ngrok (a reverse proxy). We opted to use this corpus because of the diversity of systems included and because it had been used in earlier published research. We are using the version of each system current as of October 20, 2022, which is when the underlying repositories were cloned from GitHub. Overall, the corpus includes 482,039 Go source files with a total of 126,175,477 non-blank, non-comment lines of Go source code, as counted by cloc [20]. Since some of the projects include library code in their repositories, some code files may be duplicated between projects.

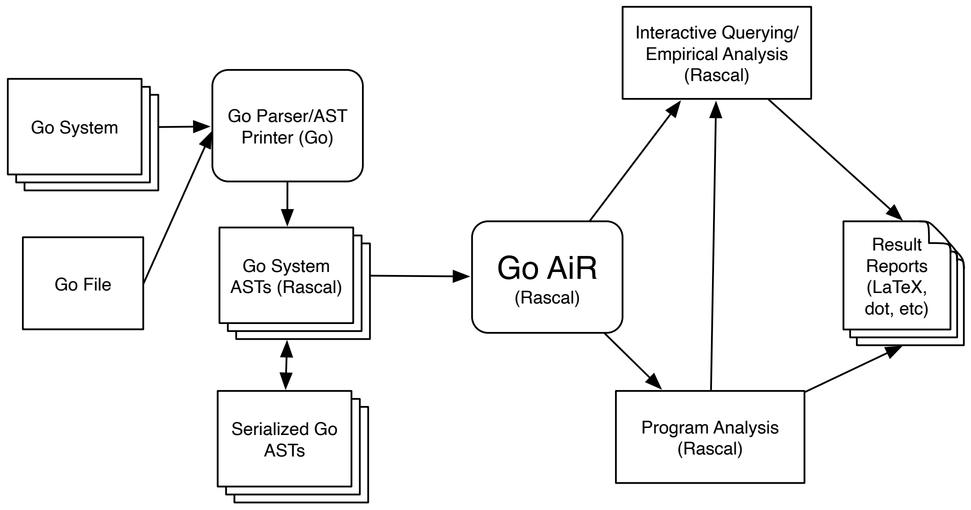
Each system in the corpus was parsed and loaded into Go AiR, with the resulting `System` serialized to disk. We then searched through all systems to identify any files that did not parse, which are represented by an alternate `errorFile` constructor for the `File` type. A total of 193 files in the corpus, across 15 different systems, were not able to be parsed and loaded into Rascal. After investigating, the following causes were identified:

- In 2017-talks and zygomys, a number of files are missing a starting package specification, and (in zygomys) some files have code replaced with an ellipses (assumedly as a placeholder), making them invalid;
- gotraining seems to include files meant to be used as templates but that are not valid Go code;
- amazon-ecs-agent and dep includes files with only a comment but with no code, which cannot be parsed as Go program files;
- buffalo and gotests include completely empty files which also cannot be parsed as Go program files;
- cockroach, ginkgo, gitkube, golang-go, gotests (also above), pulumi, subnet, tools, and vgo include test files that are supposed to cause parse errors.

In summary, all files that could not be loaded contain some sort of either intentional or inadvertent parse error in the file.

V. An Extended Example

To explore how channel-based concurrency features are used in Go, you can script an analysis, using Rascal and Go AiR, to first identify where, and how often, these features occur. The Gocurrency tool [16] does this directly in Go, using visitors over Go ASTs (similar to what we described in Section II).



Go System
Go File
Go Parser/AST Printer (Go)
Go System ASTs (Rascal)
Serialized Go ASTs
Go AiR (Rascal)
Interactive Querying/ Empirical Analysis (Rascal)
Program Analysis (Rascal)
Result Reports (LaTeX, dot, etc)

Fig. 6. Overview: Go Analysis in Rascal (AiR).

Here, we show how to use Go AiR to script one part of that analysis. Using Go AiR allows us to built such code queries in an iterative, exploratory fashion while taking advantage of Rascal features such as functions, relations, and advanced pattern matching operations.

A *channel* in Go is used to allow *goroutines*—functions that run in lightweight threads—to communicate. A channel is created by calling the `make` function, passing in the type of channel to create and an optional channel size. If a size is given, this denotes the size of the send/receive buffer: channels can send as long as the buffer is not full, and can receive as long as the buffer is not empty, but otherwise must wait. If no size is given, no buffer is present, making all calls synchronous. For instance, `make(chan bool, 10)` would create a buffered channel for sending and receiving values of type `bool`, with a buffer size of 10, while `make(chan error)` would create an unbuffered channel for sending and receiving `error` values.

Figure 7 shows several iterations of developing a query to find `make` calls that create channels (`make` can also be used to create other types of values, such as slices). In Step 1, we find all calls to `make`, regardless of the type of value being created. To do this, we find/match (the `:=` operator) all occurrences of the `callExpr` node, which represents function calls (`/c:callExpr` binds `c` to the result and matches regardless of depth in the tree). The function name is given as the literal identifier `make`. The second and third parameters of `callExpr` are ignored using _. We return a relation from the location of a call (`c.at`) to the call itself (just `c`) (the angle brackets represent a tuple, while the `|` indicates this is a comprehension, with results on the left based on the values from the right, which finds each match in turn). In this example, this search is done over the ASTs for Docker Community Edition [21], one of the projects analyzed using Gocurrency. This first query returns 9,564 potential calls.

Next, we want to narrow the results to only include channel creation. To do this, we need to include a channel type as part of the pattern. This is shown in Step 2. The channel type is given as an argument to the call to `make`, so we provide this as the first parameter in the list of parameters for the call expression. `chanType(_,_)` matches any channel type: the first parameter is the type of value transmitted over the channel, while the second is the direction (send, receive, or bidirectional). `_*` then matches 0 or more additional list parameters. The query now returns 1,139 channel creation calls. If we replace `_*` with `Expr e`, this requires a second parameter for the `make` call, which would be the capacity. This gives us 441 calls. If we instead remove `_*` completely (i.e., capacity is not included), we instead get 698 calls.

Finally, we can create a function with the proper logic that returns the information we need: the location of the calls to `make` that create channels, plus the type of channel and the capacity, if given. The code for this is shown in Step 3. We create a new data type, `Capacity`, that lets us represent two options: either the capacity is given as an expression (`capacityProvided`), in which case we want to record the expression used to indicate the capacity, or no capacity is given (`capacityNotProvided`). We then define a relation over triples: the call location, the channel type, and the capacity. We give this relation type the name `ChannelMakes` using `alias`, then use this as the return type of the function we declare. The function accepts a `System`, then identifies both calls in this `System` where a capacity is provided and calls where no capacity is provided. The function returns the union of both results. Given `docker`, this again returns the 1,139 results we saw above.

To compare directly with the results computed by Gocurrency, we can remove any results included in `test`, `tests`, or `vendor` folders, which include test code (`test` and `tests`) and library code (`vendors`). Doing this, we get a total of 482 channel creation calls. Gocurrency instead computes 424 total calls. This discrepancy seems to be caused by Gocurrency missing channel creation calls that occur nested inside object

```
// Step 1: Find calls to make
calls = { < c.at, c > | /c:callExpr(ident("make"),_,_) := docker };

// Step 2: Find calls to make a channel
calls = { < c.at, c > | /c:callExpr(ident("make"),
    [chanType(_,_),_*],_) := docker };

// Step 3: A function to find all channel declarations
public data Capacity = capacityNotProvided() | capacityProvided(Expr e);
alias ChannelMakes = rel[loc callLocation, Expr channelType, Capacity cap];
public ChannelMakes findChannelMakes(System pt) {
    callsWithCapacity = { < c.at, ct, capacityProvided(cap) >
        | /c:callExpr(ident("make"),[ct:chanType(_,_),Expr cap],_) := pt };
    callsWithoutCapacity = { < c.at, ct, capacityNotProvided() >
        | /c:callExpr(ident("make"),[ct:chanType(_,_)],_) := pt };
    return callsWithCapacity + callsWithoutCapacity;
}
```

Fig. 7. Finding Channel Creation Calls, in Rascal.

creation expressions, where the new channels are assigned to fields of the object being created.

VI. Related Work

As Go has become more widely used, it has attracted attention from the research community. Most of this work has focused on techniques for analyzing and verifying properties of Go programs. Lange et al [22] presented a framework for verification of Go programs making use of message-passing concurrency with channels. This framework models the communication between goroutines as behavioral types, which are then analyzed by the framework. An extension of this work [23] covered a larger part of the Go language, providing more precise results for a larger range of programs. Chajed et al. [24] described Perennial, a system for verifying concurrent programs. Perennial extends Iris [25], a concurrent separation logic formalized in the Coq proof assistant. A subset of Go, dubbed Goose, is supported by Perennial, but, while this subset does include support for goroutines, it does not include support for channels or for the concurrency features found in the `sync` package. Gabet and Yoshida [26] also used behavioral types, in this case to analyze a mixture of both message-passing and shared-memory concurrency features. Wolf et al. [27] described Gobra, a verification tool based on separation logic [28] that allows users to add program annotations, such as pre- and post-conditions, and that can then verify properties such as memory safety for a significant subset of Go, including for programs that use concurrency and the heap. Gobra transforms annotated programs into an existing intermediate language, Viper [29], which can then be verified using existing verification tools.

Liu et al. [30] used a constraint solver to detect blocking bugs, and then proposed patches that can be used to fix these bugs. Dilley and Lange [31] used Promela to model the concurrency features of a subset of Go named *MiniGo*, then used the Spin model checker to detect concurrency errors. An extension of this, also by Dilley and Lange [32], used bounded model checking to verify behavioral types that model concurrency features in Go, including traditional features such as mutexes. Other work includes that of Stadtmüller et al. [33] on modeling Go concurrency using an extended regular expression notation and automata; the work of Ng and Yoshida [34], which models concurrency in Go using session types; the work of Midtgaard et al. [35], which focuses on analyzing Go programs where goroutines communicate using synchronous message passing; and the work of Zhang et al. [36], which provides a static analysis to detect errors in both channel and `WaitGroup` usage. Sulzmann and Stadtmüller [37], [38] instrumented code to generate a runtime trace that is analyzed to detect potential bugs in the use of message-passing concurrency. Dilley and Lange [16] explored the use of message-passing concurrency features in Go, such as channel creation, the sending and receiving of messages, and the use of the `select` construct, which includes different cases for the different concurrent operations that could occur, and which chooses a specific case non-deterministically when multiple cases could execute.

The goal of Go AiR is to provide what we hope is a more open and extensible foundation for the analysis of Go programs. This could include the creation of analyses directly in Rascal, similar to PHP AiR [19] (for PHP), Clair [39] (for C), Ada AiR [40] (for Ada), and Lua AiR [41] (for Lua), but could also use Rascal to bridge to different analysis and verification tools. This is discussed more in Section VII.

VII. Conclusions and Future Work

In this paper, we have introduced Go AiR, a new analysis framework for Go written using the Rascal meta-programming language. Although still in early stages, we believe it is possible to already use Go AiR as a basis for code exploration and analysis. We have tested the process of mapping Go ASTs into Rascal on approximately 125 million lines of Go source code across 854 popular projects from GitHub, and provide functionality to easily work with multiple Go projects as well as multiple tagged versions of the same project, based on Git commit history. Support for representing errors in translating

ASTs from Go to Rascal, built directly into Go AiR, should make it easier to find and fix incompatibilities caused by evolution of Go and the Go AST libraries.

Going forward, we plan to continue adding new features to Go AiR. This includes the creation of core analyses, including control flow graph construction, that are needed by other analyses. Beyond this, the main focus will be on concurrency analysis, both because of Go's novel concurrency features and because Go is often used for developing concurrent, networked software systems. As part of this work, we are developing a formal definition of a core version of Go, with channel-based concurrency, in Maude [42], which will allow us to take advantage of existing support (including model checking and state space exploration) for analyzing and verifying concurrent systems using rewriting logic. This will incorporate earlier developed support for working with Maude definitions from Rascal [10], which includes the ability to extend language definitions with Rascal source location information, allowing results from Maude to be linked back to the original Go source code. This core will be built to focus first on the most-used aspects of concurrency in Go. Longer-term, this will be used as part of modeling the behaviors of, and interactions between, IoT and cloud-based systems written using Go.

References

[1] TIOBE, "TIOBE Index." [Online]. Available: https://www.tiobe.com/tiobe-index/

[2] RedMonk, "The RedMonk Programming Language Rankings: January 2023." [Online]. Available: https://redmonk.com/sogrady/2023/05/16/language-rankings-1-23/

[3] GitHub, "The state of open source software," 2022. [Online]. Available: https://octoverse.github.com/2022/top-programming-languages

[4] C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[5] R. Milner, *Communication and Concurrency*, ser. PHI Series in Computer Science. Prentice Hall, 1989.

[6] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes, I," *Inf. Comput.*, vol. 100, no. 1, pp. 1–40, 1992.

[7] ——, "A Calculus of Mobile Processes, II," *Inf. Comput.*, vol. 100, no. 1, pp. 41–77, 1992.

[8] P. Klint, T. van der Storm, and J. Vinju, "EASY Meta-programming with Rascal," in *Post-Proceedings of GTTSE'09*, ser. LNCS. Springer, 2011, vol. 6491, pp. 222–289.

[9] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proceedings of SCAM 2009*. IEEE, 2009, pp. 168–177.

[10] M. Hills, P. Klint, and J. J. Vinju, "RLSRunner: Linking Rascal with K for Program Analysis," in *Proceedings of SLE'11*, ser. LNCS, vol. 6940. Springer, 2011, pp. 344–353.

[11] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical Computer Science*, vol. 96, no. 1, pp. 73–155, 1992.

[12] J. Meseguer and G. Roşu, "The rewriting logic semantics project," *Theoretical Computer Science*, vol. 373, no. 3, pp. 213–237, 2007.

[13] J. Meseguer and G. Rosu, "The rewriting logic semantics project: A progress report," *Inf. Comput.*, vol. 231, pp. 38–69, 2013.

[14] M. Hills, T. F. Şerbănuţă, and G. Roşu, "A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters," in *Proceedings of WRLA'06*, ser. ENTCS, vol. 176, no. 4. Elsevier, 2007, pp. 215–231.

[15] G. Rosu and T. Serbanuta, "An overview of the K semantic framework," *J. Log. Algebraic Methods Program.*, vol. 79, no. 6, pp. 397–434, 2010. [Online]. Available: https://doi.org/10.1016/j.jlap.2010.03.012

[16] N. Dilley and J. Lange, "An Empirical Study of Messaging Passing Concurrency in Go Projects," in *Proceedings of SANER 2019*, 2019, pp. 377–387.

[17] "Go AiR (Analysis in Rascal)." [Online]. Available: https://github.com/PLSE-Lab/go-analysis

[18] "Go2Rascal." [Online]. Available: https://github.com/PLSE-Lab/go2rascal

[19] M. Hills, P. Klint, and J. J. Vinu, "Enabling PHP Software Engineering Research in Rascal," *Science of Computer Programming*, vol. 134, pp. 37–46, 2017.

[20] "Count Lines of Code Tool." [Online]. Available: https://github.com/AlDanial/cloc

[21] "Docker Community Edition." [Online]. Available: https://github.com/docker/docker-ce/tree/44a430f4c43e61c95d4e9e9fd6a0573fa113a119

[22] J. Lange, N. Ng, B. Toninho, and N. Yoshida, "Fencing off Go: Liveness and Safety for Channel-Based Programming," in *Proceedings of POPL 2017*. ACM, 2017, p. 748–761.

[23] ——, "A Static Verification Framework for Message Passing in Go Using Behavioural Types," in *Proceedings of ICSE 2018*. ACM, 2018, p. 1137–1148.

[24] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich, "Verifying Concurrent, Crash-Safe Systems with Perennial," in *Proceedings of SOSP 2019*. ACM, 2019, p. 243–258.

[25] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal, "The Essence of Higher-Order Concurrent Separation Logic," in *Proceedings of ESOP 2017*. Springer, 2017, pp. 696–723.

[26] J. Gabet and N. Yoshida, "Static Race Detection and Mutex Safety and Liveness for Go Programs," in *Proceedings of ECOOP 2020*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 166. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 4:1–4:30.

[27] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular Specification and Verification of Go Programs," in *Proceedings of CAV 2021*. Springer International Publishing, 2021, pp. 367–379.

[28] J. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *Proceedings of LICS 2002*, 2002, pp. 55–74.

[29] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A Verification Infrastructure for Permission-Based Reasoning," in *Proceedings of VMCAI 2016*. Springer Berlin Heidelberg, 2016, pp. 41–62.

[30] Z. Liu, S. Zhu, B. Qin, H. Chen, and L. Song, "Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems," in *Proceedings of ASPLOS 2021*. ACM, 2021, p. 616–629.

[31] N. Dilley and J. Lange, "Bounded verification of message-passing concurrency in Go using Promela and Spin," *Electronic Proceedings in Theoretical Computer Science*, vol. 314, pp. 34–45, apr 2020.

[32] ——, "Automated Verification of Go Programs via Bounded Model Checking," in *Proceedings of ASE 2021*, 2021, pp. 1016–1027.

[33] K. Stadtmüller, M. Sulzmann, and P. Thiemann, "Static Trace-Based Deadlock Analysis for Synchronous Mini-Go," in *Proceedings of ASPLOS 2016*. Springer International Publishing, 2016, pp. 116–136.

[34] N. Ng and N. Yoshida, "Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis," in *Proceedings of CC 2016*. ACM, 2016, p. 174–184.

[35] J. Midtgaard, F. Nielson, and H. R. Nielson, "Process-Local Static Analysis of Synchronous Processes," in *Proceedings of SAS 2018*. Springer International Publishing, 2018, pp. 284–305.

[36] D. Zhang, P. Qi, and Y. Zhang, "GoDetector: Detecting Concurrent Bug in Go," *IEEE Access*, vol. 9, pp. 136 302–136 312, 2021.

[37] M. Sulzmann and K. Stadtmüller, "Trace-Based Run-Time Analysis of Message-Passing Go Programs," in *Proceedings of HVC 2017*. Springer International Publishing, 2017, pp. 83–98.

[38] M. Sulzmann and K. Stadtmüller, "Two-Phase Dynamic Analysis of Message-Passing Go Programs Based on Vector Clocks," in *Proceedings of PPDP 2018*. ACM, 2018.

[39] M. T. W. Schuts, R. T. A. Aarssen, P. M. Tielemans, and J. J. Vinju, "Large-scale semi-automated migration of legacy C/C++ test code," *Softw. Pract. Exp.*, vol. 52, no. 7, pp. 1543–1580, 2022.

[40] "Ada AiR (Analysis in Rascal)." [Online]. Available: https://github.com/cwi-swat/ada-air

[41] P. Klint, L. Roosendaal, and R. van Rozen, "Game Developers Need Lua AiR - Static Analysis of Lua Using Interface Models," in *Proceedings of the 11th International Conference on Entertainment Computing (ICEC 2012)*, ser. LNCS, vol. 7522. Springer, 2012, pp. 530–535.

[42] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, Eds., *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, ser. LNCS, vol. 4350. Springer, 2007.