TU WIEN Informatics

Typprüfung einer neuartigen Sprache für rekonfigurierbare Multiagentensysteme

Originaltitel: Type checking a novel language for reconfigurable multi-agent systems

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Benjamin Stolz
Matrikelnummer 12025952

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Ezio Bartocci, PhD
Mitwirkung: Univ. Ass. Luca Di Stefano, PhD

Wien, 1. Juli 2025

Benjamin Stolz Ezio Bartocci

TU WIEN Informatics

Type checking a novel language for reconfigurable multi-agent systems

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Medical Informatics

by

Benjamin Stolz
Registration Number 12025952

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Ezio Bartocci, PhD
Assistance: Univ. Ass. Luca Di Stefano, PhD

Vienna, July 1, 2025

Benjamin Stolz	Ezio Bartocci

Erklärung zur Verfassung der Arbeit

Benjamin Stolz

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Juli 2025

Benjamin Stolz

Kurzfassung

Rekonfigurierbare Multiagentensysteme (MAS) erfordern maßgeschneiderte Modellierungswerkzeuge, die dynamische Agenteninteraktionen und Systemrekonfiguration erfassen können. Der ReCiPe-Formalismus bietet domänenspezifische Abstraktionen für die Spezifikation solcher Systeme, während das R-CHECK-Framework eine domänenspezifische Sprache (DSL) auf der Grundlage von ReCiPe mit Werkzeugen für den Modellentwurf, die grundlegende Simulation und die Verifizierung durch Modellprüfung implementiert. Die begrenzten statischen semantischen Überprüfungen, die derzeit in R-CHECK implementiert sind, lassen jedoch ein erhebliches Potential für Fehler zur Laufzeit offen. In diesem Beitrag wird der Entwurf und die Implementierung eines statischen Typsystems für R-CHECK vorgestellt, das auf einer strengen Formalisierung der Typisierungsregeln für primitive Typen, Operatoren, Prozesse und Spezifikationen in LTOL basiert. Das Typsystem deckt domänenspezifische Konstrukte wie agentenspezifische Definitionen, Beobachtungen und quantifizierte Formeln ab. Die Implementierung, die auf den Langium- und Typir-Frameworks basiert, erweitert den bestehenden R-CHECK-Workflow und die Integration in Visual Studio Code (VS Code), um eine frühzeitige Erkennung von Typfehlern und ein klares, domänenspezifisches Feedback an die Benutzer zu ermöglichen. Eine Evaluierung zeigt die Fähigkeit des Typprüfers, Fehler zu erkennen und potenziell unerwünschte Modellierungspraktiken hervorzuheben, was das Vertrauen in die Korrektheit des Modells erhöht. Dieser Beitrag hilft Domänenexperten bei der Entwicklung rekonfigurierbarer MAS-Modelle mit größerem Vertrauen in deren Sicherheit und Konsistenz.

Abstract

Reconfigurable multi-agent systems (MASs) require tailored modelling tools that can capture dynamic agent interactions and system reconfiguration. The ReCiPe formalism provides domain-specific abstractions for specifying such systems, while the R-CHECK framework implements a domain-specific language (DSL) based on ReCiPe with tooling for model design, basic simulation, and verification through model checking. However, the limited static semantic checks currently implemented in R-CHECK leave significant potential for errors to manifest at runtime. This paper presents the design and implementation of a static type system for R-CHECK, grounded in a rigorous formalization of typing rules for primitive types, operators, processes, and specifications in LTOL. The type system covers domain-specific constructs such as agent-specific definitions, observations, and quantified formulas. The implementation, based on the Langium and Typir frameworks, extends the existing R-CHECK workflow and Visual Studio Code (VS Code) integration to provide early detection of type errors and clear, domain-focused feedback to users. An evaluation demonstrates the ability of the type checker to catch errors and highlight potentially undesirable modelling practices, improving confidence in model correctness. This contribution helps domain experts develop reconfigurable MASs models with greater assurance in their safety and consistency.

Contents

Kurzfassung vii

Abstract ix

Contents xi

1 Introduction 1

2 Background 3
2.1 Type Checking 3
2.2 The ReCiPe Formalism 5
2.3 The R-CHECK Framework 7
2.4 Langium 8
2.5 Typir and Typir-Langium 9

3 Formalization of the ReCiPe Type System 11
3.1 Typing Rules 12
3.2 Typing Context 18
3.3 Model Rule 23

4 Implementation in R-CHECK 25
4.1 Typir-Based Type Checker 25
4.2 Langium Integration 27

5 Evaluation 29
5.1 Demonstrative Examples 29
5.2 Limitations 34

6 Conclusion 37
6.1 Summary of Contributions 37
6.2 Future Work 37

List of Figures 39

Listings 41

Glossary **43**

Acronyms **45**

Bibliography **47**

Source Code **49**
R-CHECK Type System 49
Utility Functions 62

Grammar Changes **69**

CHAPTER 1

Introduction

Domain-specific languages (DSLs) are languages that are specifically designed to offer a high level of expressiveness and ease of use when working and solving problems within a specific domain (set of similar problems). This is achieved by trading off the general expressiveness and broad feature set typical of general-purpose languages (GPLs) such as TypeScript.

In this work, the domain of interest is the modelling of reconfigurable multi-agent systems (MASs), which are systems of multiple interacting agents that autonomously perceive and act upon their environment to achieve individual and collective goals. Reconfigurable MASs are capable of altering their structure, such as agent roles and communication pathways, in response to changes in the environment or internal system requirements, leading to better flexibility and adaptability [1]. Because this is a highly specialized domain, using a GPL would burden the modeller with low-level details during the formalization of MASs or the reasoning about them.

ReCiPe is a formal specification language used to serve precisely this domain of reconfigurable MASs. It provides domain-specific abstractions for defining agents and their behaviour [1]. Due to the inherent property of DSLs such as ReCiPe to operate on a confined domain, it is likely that the user base of such a language is small when compared to that of a well-spread GPL. Additionally, it is common that the users of DSLs have their professional background in the language's application domain rather than informatics or language theory. As a result, these users typically have a reduced general experience with formal languages or GPLs which can lead to difficulties when working with the DSL.

To address this issue, it is critical to provide the user with effective tools that streamline the workflow when writing specifications in a DSL. In the case of ReCiPe, these tools are provided in the form of R-CHECK. The R-CHECK framework supports a high-level programming language based on ReCiPe, which includes features to help design, simulate, and verify ReCiPe models. At the time of writing, R-CHECK mainly verifies the syntax of the provided program and only includes a few basic semantic checks for verifying program correctness [1]. As a result, working with R-CHECK remains prone to a variety of errors, particularly type errors.

In this paper, the design and implementation of a static type system for ReCiPe is presented. The implementation is seamlessly integrated into R-CHECK. By catching type errors early and providing precise feedback, this extension significantly enhances the modelling workflow within R-CHECK, enabling domain experts to more effectively construct reconfigurable MAS specifications.

What follows is a brief introduction to the topics and tools that are relevant to this work in Chapter 2. Chapter 3 provides a formal definition of the proposed type system. In Chapter 4 follows a description on how the type system is implemented in the existing codebase. In Chapter 5 a simple evaluation of the type checker is presented. Lastly, Chapter 6 provides concluding remarks and outlines directions for future work.

CHAPTER 2

Background

This chapter introduces five key topics. It starts with an overview of type checking, followed by a brief introduction to the ReCiPe formalism and the R-CHECK framework. The chapter concludes with a mention of the tools used to integrate the type system into R-CHECK. The aim is not to give a complete overview of these aspects, but rather to provide the necessary context for the chapters to follow.

2.1 Type Checking

The *grammar* of a language defines its basic constructs and the rules for how these constructs can interact with each other. During the process of *parsing*, a piece of text written in the language is compared to the grammar. The text is split into basic elements, and a new data structure, the abstract syntax tree (AST), is created. The AST is a hierarchical tree structure that represents the logical structure of the source text, making it easier to analyse or manipulate during further processing.

In the broader context of *program analysis*, which focuses on examining a program to verify its correctness and enhance its reliability before execution, the parsing process is just one part of the compilation workflow. Program analysis can be further divided into three distinct phases. The first phase, *lexical analysis*, reads the raw source text and groups characters into tokens such as identifiers, literals, and operators. The second phase, *syntax analysis*, takes the token stream and constructs an AST according to the language grammar while reporting any syntax errors encountered. The last phase, *semantic analysis*, performs various validations to ensure the meaning of the program is consistent with the rules of the language [2].

type checking refers to the phase of the semantic analysis in which a program is examined against a set of rules known as the type system. In this context, types are defined as collections of values that share certain properties. A type may be defined by membership

```
// The type of x is inferred from the expression 'true',
// which types to Boolean.
const x = true;

// The type of sum is inferred from the return type of + operator
// which is number. The types of the operands 3 and 5 are both number.
const sum = 3 + 5;
```

Listing 2.1: Type inference examples in TypeScript.

in a specific set, for example a valid range of integer values or the literals `true` and `false` belonging to the Boolean type of TypeScript. Types can also be defined by their structure such as in class definitions with their members. These types may be predefined by the programming language or introduced by the programmer. A type system also defines rules that determine valid interactions among types and specifies the warnings or errors presented to the programmer for violations of those rules [3].

In principle, all the checks of semantic analysis may run dynamically (at program execution), but identifying errors such as type errors at compile time rather than runtime significantly lowers the risk for unexpected crashes of the compiled program. The early detection of type errors during compile time is known as *static typing*. Additionally, a language with a type system that can be used to assign unambiguous type information to each expression is referred to as *strongly typed*. The purpose of all validation steps is to detect errors before the program is executed [2].

With the type system in place, each node of the AST gets annotated with type information. Literals get their type from their respective value set membership. Identifiers (variables) are either annotated with a fixed type upon declaration or get their type using *type inference*, which is the process of deducing a type for an expression from its usage and context rather than from an explicit declaration [2, 3]. Since Langium and R-CHECK are both implemented in TypeScript, the following examples in this section are also provided using TypeScript to align with the underlying language of the project. For a variable declaration in TypeScript that reads `const x = true;` the type of `x` is inferred from the expression that is assigned. In this case the identifier `x` gets annotated with the type Boolean of the Boolean literal `true`.

Type inference also applies to function calls, requiring the types of input parameters to match the signature of the formal parameters. The return type is inferred either from an explicit definition of a return type or from the type of the expression used in the return statement [3]. Note that the same logic applies to built-in operators. For example, in TypeScript one might write `const sum = 3 + 5;`. The compiler then infers that `sum` has type `number`, since the `+` operator accepts two `number` operands and returns a `number`, as illustrated in Listing 2.1.

Type coercion, often called *type casting*, refers to the conversion of a value from one

```
// The number variable year gets converted to string
// in order to be combined with the rest of the template literal string.
const year: number = 2025;
const message = `The current year is ${year}.`;
```

Listing 2.2: Implicit type casting of number to string in TypeScript.

type to another. This conversion may occur implicitly, when the compiler automatically transforms values according to contextual rules, or explicitly, when the programmer invokes a cast [2]. In TypeScript, a `number` value may be implicitly converted to a `string` type when used within a template literal, as presented in Listing 2.2.

The type checking phase is completed once the compiler has exhaustively analysed the AST and either confirmed that all expressions satisfy the type system of the language or identified any type errors. At that point, the compiler can either advance to subsequent stages or report the detected errors to the developer [3].

2.2 The ReCiPe Formalism

ReCiPe is a formalism for modelling systems of communicating agents. Informally, agents are constructs that hold an internal state as a set of variables and that define an infinitely repeating process of message send and receive statements. Agents in ReCiPe support four different kinds of communication structures referred to as *basic processes*.

Among these basic processes the *send* and *receive* processes operate on *channels*, which are a fixed set of names defined by the user. The channelled send process can be used to communicate to any number of agents by specifying a channel. A *send-guard* expression may be used to filter the possible receivers of the message. The message body consists of a set of assignment to message variables. The send process may also include an update statement that updates the internal state of the agent. The basic receive process serves as the counterpart to send and specifies only a channel from which to receive and an update statement to process the received data. Generally, both send and receive processes are *blocking*, meaning that an agent cannot proceed to its next statement until a send process has been received by at least one other agent or a receive process has obtained a message on its specified channel. The exception to this is the communication on the special channel *broadcast*, denoted by the $\star$ symbol, which is globally available to all agents and enables non-blocking send processes.

On the other hand, the *supply* and *get* processes allow for direct communication between agents. The supply and get processes both require specifying a location as well as data and update fields with the same structure as those of the channel-based basic processes. In a supply process an agent names the location under which it supplies data and defines a set of variables as the message body. The location may be *myself*, which means the agent supplies data under its own name, or it may be *any*, allowing any agent to accept the supply. The get process acts as the counterpart to supply and similarly includes a location specifier together with a data field and an update statement. The location in this case may be the name of a specific agent, given the receiver knows the identity of the supplier. Otherwise, it may be a Boolean expression that filters incoming supplies so that any matching supply is processed. More precisely, only supplies that satisfy this condition and that specified the *any* location are considered, and exactly one of these supplies will be processed. The existence of data and update fields in the get process indicates that direct communication requires the receiver to return data to the supplier, thereby enabling a two-way exchange. It is important to note that the data part in the get process is optional, but if a supply expects data in its update statement, then a get will only match if it provides the expected data.

The complete behaviour of agents is formalized by combining basic processes. It may be specified that processes execute in a specific order, or that the agent chooses non-deterministically among several basic processes. In addition, it is possible to specify that any process or combination of processes repeats indefinitely. By combining sequencing, choice, and repetition, the complex behaviour of agents can be fully formalized.

A *system* can then be formed by creating named instances of the specified agents. The initial state (local variables) of agent instances can also be restricted in the system definition [1].

2.2.1 Specifications in LTOL

The final component of a ReCiPe model is a list of *specifications*, which define the desired properties of the system when it is executed. These specifications are written in LTOL, the temporal logic used by the R-CHECK framework. LTOL is a temporal logic language that describes the expected behaviour of a system as it evolves over time. It is mostly based on Linear Temporal Logic (LTL) [4] and therefore includes the standard temporal operators: *next* (X), *until* (U), *weak until* (W), *globally* (G), *finally* (F), and *release* (R).

In addition to these standard operators, LTOL introduces two observation-based operators: the "possibly" operator, also known as the *diamond* ($\langle\langle o \rangle\rangle$), and the "necessarily" operator, also known as the *box* ($[[o]]$). In both cases, o represents an *observation*, which is an expression over a message exchange. An observation may describe conditions such as a message being sent over the broadcast channel, or may predicate on the sender, recipients, or contents of the message. Informally, if o is an observation and φ is an LTOL property, the expression $\langle\langle o \rangle\rangle \varphi$ states that, at a given point in time, a message satisfying o occurs and, after that, φ must hold. On the other hand, $[[o]]\,\varphi$ states that, at a given point in time, if a message satisfying o occurs, then φ must hold afterwards. If the message does not occur, the entire formula is true (logical implication). This observation-based extension gives LTOL great expressive power to specify precisely the kinds of interactions expected as the system evolves.

LTOL also includes quantifiers that allow properties to be specified over sets of agents. It is possible to state that a property must hold for every agent of a certain type (*forall*) or for at least one such agent (*exists*) [1].

The main objective of the ReCiPe formalism is to determine whether these LTOL specifications hold for a given model. This model checking process is performed by the R-CHECK framework.

2.3 The R-CHECK Framework

Building on the semantics of ReCiPe, R-CHECK defines the grammar for a high-level programming language. R-CHECK programs begin with a prelude declaring global variables and data structures. The data structures include the definitions of additional channel names or enumerations of model-specific properties. Global variables fall into two categories based on their intended usage: *message-structure variables* and *property variables*. Message-structure variables (or simply, message variables) define the data that agents may put in the data part of their statements, specifying the structure of the messages they can exchange. Property variables fulfill a role similar to that of interfaces in object-oriented GPLs and define shared properties that agents can reason about. Each agent defined within a model has its own interpretation of the property variables defined through its relabel expressions. Whenever agents need to predicate about other agents, for example in send guards, they use predicates over property variables so they do not need to know the local variables of other agents and so that predicates remain valid across different types of agents. Both message and property variables must be specified with a fixed data type, which means that variables in R-CHECK are strongly typed. The prelude concludes with the definition of guards, which are predicate functions over property variables which can be used as send-guards within process definitions. These global members of a ReCiPe program are available for every agent in the model.

After the prelude come the agent definitions. Each agent is defined with a distinct name, a set of local variables, a predicate expressing initial restrictions, a section to relabel property-variables, an agent-level receive-guard and, finally, the infinitely repeated process of the agent. The receive guard specifies what channels the agent wants to participate to. Multiple instances of agents and their initial restrictions can then be used to form a system. The last part of an R-CHECK program is a series of specifications, reasoning about the behaviour of the system.

Additionally, R-CHECK provides an integrated toolkit for designing, simulating, and verifying systems of agents. The features regarding the design process of an R-CHECK program are built into a Visual Studio Code (VS Code) extension which enables syntax highlighting and other editor features like “go to definition”. The extension also provides commands to visualize agents as state automata, explore behaviours (simulate), and model check a system.

The architecture of R-CHECK builds upon Langium to perform parsing and basic semantic analysis of a source file. This is followed by a translation process of the AST to specifications that can be processed by the model checker nuXmv. This step is performed by a custom translation layer written in Java. The model checker then yields the desired results about the design of the agent system [1].

2.4 Langium

Langium is a toolkit by the *Eclipse Foundation* for engineering DSLs. Langium is written in TypeScript and the project aims to ease the development of DSLs in a web-based technology stack [5]. According to the project proposal of Langium, the functionality of the framework is derived from the preceding framework Xtext [6], meaning that the core principles of the tool are well known and thoroughly tested [7]. For editor support, Langium can be used to deploy a language server that supports the Language Server Protocol (LSP) for features such as validation, auto-completion, and cross-reference navigation [8]. The LSP is supported by the most relevant text editors and IDEs at the time of writing such as Visual Studio, VS Code, Atom, Sublime Text, and JetBrain IDEs such as IntelliJ IDEA and PyCharm [8].

In a Langium project, a DSL is formalised within a grammar file using Langium's *grammar language*. The grammar file holds information about both the *concrete syntax*, which defines how the language constructs are written and structured in text form, and the *abstract syntax*, which specifies the hierarchical structure of language elements and their properties when translated to the AST. This AST is then used for all following operations such as cross-reference resolution and validation [5].

Listing 2.3 shows a part of the grammar definition of R-CHECK as an example to illustrate the architecture of Langium. The basic elements in a Langium grammar definition are *terminals* and *parser rules*. For example, terminals such as `ID` describe valid identifiers, while `WS` matches and discards whitespace using the **`hidden`** keyword. Parser rules such as `Enum` and `Case` show how these terminals and keywords combine to define higher-level constructs. The `Enum` rule specifies that an enum starts with the keyword `'enum'`, followed by an identifier (`name=ID`), and contains one or more `Case` elements separated by commas. The inclusion of the `Case` parser rule within the `Enum` rule illustrates how the abstract syntax is defined through the grammar language by nesting and structuring related elements. The **`entry`** `Model` rule describes how multiple elements, like `Enum`, `MsgStruct`, or `Agent`, can be grouped together and combined to describe the complete structure of a valid R-CHECK file. Each parser rule is given a name that determines how the parsed elements are organised in the resulting AST [9].

When the grammar is compiled using the Langium CLI, it is transformed into TypeScript files that define the data structures used to create an AST during the parsing of a program. It is important to note that while the AST generation includes information about *cross-references*, which allow parts of the AST, such as an identifier used in one place, to point to its corresponding declaration elsewhere in the same file or across files. When a program is parsed to form an AST, the AST still contains gaps where cross-references are expected. These gaps have to be resolved during a cross-reference resolution step. In Langium, this is typically achieved through a combination of a scope computation and a

```
grammar RCheck

entry Model:
    //Global section
    (
        (enums+=Enum)
        | ('message-structure' ':' msgStructs+=MsgStruct (',' msgStructs+=MsgStruct)*)
        | ('property-variables' ':' propVars+=PropVar (',' propVars+=PropVar)*)
        | guards+=Guard
    )*
    //Agents and instantiation
    (agents+=Agent)*
    'system' '=' (system+=Instance ('||' system+=Instance)*)
    // Specs
    ('SPEC' specs+=Ltol ';'?)*
    ;
// [...]
Enum:
    'enum' name=ID '{' cases+=Case (',' cases+=Case)* '}';

Case: name=ID;
// [...]
hidden terminal WS: /\s+/;
terminal ID: /[_a-zA-Z][\w_]*/;
```

Listing 2.3: Excerpt from the grammar definition of R-CHECK.

scope provider, which together collect possible candidates and determine the valid target for each reference [9].

After resolving the cross-references, the AST is complete and can be further validated to ensure the semantic correctness of the program. Validators can be implemented using the features of the AST, and integrated into the parser by registering them in the model definition of the language. The version of Langium used for this project is version 3.4.0.

2.5 Typir and Typir-Langium

As mentioned before, an important part of semantic analysis is type checking. Typir is a set of utilities that simplify type checking operations on an AST. The package is developed and maintained by TypeFox, the same company that also develops Langium [10], and makes it straightforward to integrate a type-checking validator into a Langium project [9]. Typir's core features include methods for easily defining primitives, functions, classes, and operators. Additionally, it provides type-checking services such as tests for assignability and equality, type inference and conversion (both implicit and explicit), as well as subtyping.

Typir-Langium is a subpackage of Typir that allowes a Typir-based type checker to be integrated into a Langium project without major setup. Typir can then directly be used with the data structures that make up a Langium AST [10]. The version of Typir and Typir-Langium used for this project is version 0.2.0.

CHAPTER 3

Formalization of the ReCiPe Type System

In this chapter, the type system of ReCiPe is formally introduced. The typing rules are expressed using a variant of the *natural deduction* notation used by Cardelli [11]. An *inference rule* consists of a set of premises $P_1, \ldots, P_n$ above a horizontal line and a conclusion C below it:

$$\frac{P_1 \quad \ldots \quad P_n}{C} \text{ Inference rule} \qquad \frac{}{C} \text{ Axiom}$$

This notation means that the conclusion holds exactly when all premises hold. In the special case of an *axiom*, the premises are omitted entirely to indicate that the conclusion is unconditionally valid. Additionally, rules may be composed into proof trees by using the conclusion of one rule as a premise of another. Doing so produces a derivation tree whose root is the final judgment. The leaves of this tree are either axioms or inference rules whose premises consist solely of basic logical expressions, which can be validated directly without invoking further rules.

Sometimes, when multiple conclusions share the same premises, they are written one underneath the other within a single inference rule. While not a standard, this notation is used for compactness and clarity, allowing related derivations to be represented more concisely.

Throughout this type system, two distinct styles of inference rules are used to express different judgments:

$$\frac{P_1 \quad \ldots \quad P_n}{\Gamma_1, \ldots, \Gamma_n \vdash x \Rightarrow (\Gamma'_1, \ldots, \Gamma'_n)} \qquad \frac{P_1 \quad \ldots \quad P_n}{\Gamma_1, \ldots, \Gamma_n \vdash x : \tau}$$

Every rule begins with a list of sets $\Gamma_1, \ldots, \Gamma_n$ called *typing contexts* (also called environments) and the turnstile $\vdash$, which is read as "under contexts $\Gamma_1, \ldots, \Gamma_n$, it is inferred that." In the first form, $x \Rightarrow (\Gamma'_1, \ldots, \Gamma'_n)$ means that processing x produces the new contexts $(\Gamma'_1, \ldots, \Gamma'_n)$. The second form, $x : \tau$, asserts that x is well typed and has type τ. Notably, within this style, the symbol $\diamond$ is used as a special case of $x : \tau$. When $x : \diamond$ is used, it denotes that x is well typed without introducing an additional data type for x. Finally, the typing contexts may be omitted to indicate that the conclusion is valid regardless of the context.

A typing context is simply the bookkeeping structure that is used to build up information about the existing types, classes, functions, and variables as the type checker traverses the AST. In the following definitions, four typing contexts with distinct purposes are used. Let Δ be a set of existing types. Let Γ be a set of mappings from identifiers (e. g. variables) to their inferred type. Let Ψ be a set of mappings from guard names to an ordered list of types (the types of their parameters). Let Σ be a set of mappings from agent names to an internal typing context that follows the structure of Γ and holds mappings from variable names to their inferred type.

Before proceeding to the concrete typing rules, a few auxiliary definitions are introduced that will simplify notation in the remainder of this chapter.

1. Let R be a relation from S to T. The *domain* of R written $\mathsf{dom}(R)$, is the set of all $s \in S$ for which there exists some $t \in T$ with $(s, t) \in R$ [12].
2. Again, let R be a relation from S to T. The notation $R(s)$ is equal to t for all $(s, t) \in R$. If $s \notin \mathsf{dom}(R)$ then $R(s)$ is undefined and translates to 'false' as a premise.
3. Similarly to the notation of the type assertion $x : \tau$, a relation between types written $\tau_1 <: \tau_2$ specifies that τ_1 is a *subtype* of τ_2. When read in reverse, τ_2 is a *supertype* of τ_1.
4. The notation $[x :: \mathit{tail}]$ denotes a non-empty list structure in which x is the first element (the head) and *tail* represents the remainder of the list. The empty list is denoted by ϵ.

3.1 Typing Rules

3.1.1 Boolean Literals

Rule 3.1 defines two axioms for the Boolean literals `true` and `false`. Under any context, these symbols type to **`bool`**.

$$\frac{}{\vdash \texttt{true} : \texttt{\textbf{bool}}} \qquad \frac{}{\vdash \texttt{false} : \texttt{\textbf{bool}}} \tag{3.1}$$

3.1.2 Location Literals

Similarly, rule 3.2 defines the axioms for **location** literals.

$$\frac{}{\vdash \texttt{myself} : \textbf{location}} \qquad \frac{}{\vdash \texttt{any} : \textbf{location}} \tag{3.2}$$

3.1.3 Channel Literals

The same can be done for **channel** literals in rule 3.3.

$$\frac{}{\vdash \texttt{chan} : \textbf{channel}} \qquad \frac{}{\vdash \star : \textbf{channel}} \tag{3.3}$$

3.1.4 Number Literals

The symbol INT used in rule 3.4 denotes the set of valid symbols that are specified by the integer terminal of the ReCiPe grammar. The rule specifies that, in any context, n types to a **range** ($[l\,..\,u]$) with the upper and lower bound both set to n under the premise that n is a valid integer terminal in ReCiPe.

$$\frac{n \in \textsc{int}}{\vdash n : [n\,..\,n]} \tag{3.4}$$

3.1.5 Subtype Relations

While number literals will type to **range**, variables can still be specified explicitly to be of type **int** within variable declarations. The **int** type can also be inferred from any **range** as specified in rule 3.5. In other words, every valid **range** is a subtype of **int**.

$$\frac{l \leq u}{\vdash [l\,..\,u] <: \textbf{int}} \tag{3.5}$$

Similarly, rule 3.6 states that every **range** type $[l_1\,..\,u_1]$ is a subtype of another **range** type $[l_2\,..\,u_2]$ exactly when the interval of $[l_1\,..\,u_1]$ is contained within the interval of $[l_2\,..\,u_2]$ (expressed in the conditions $l_2 \leq l_1$ and $u_1 \leq u_2$).

$$\frac{l_2 \leq l_1 \quad u_1 \leq u_2}{\vdash [l_1\,..\,u_1] <: [l_2\,..\,u_2]} \tag{3.6}$$

Additionally, every type is a subtype of itself and a subtype of the type **any** (rule 3.7).

$$\frac{}{\vdash \tau <: \tau} \qquad \frac{}{\vdash \tau <: \textbf{any}} \tag{3.7}$$

Rule 3.8 states that every expression e of type τ' also types to τ if τ' is a subtype of τ. This rule allows a type to be promoted to its supertype whenever required in the subsequent rules.

$$\frac{\Delta, \Gamma \vdash e : \tau' \quad \vdash \tau' <: \tau}{\Delta, \Gamma \vdash e : \tau} \tag{3.8}$$

3.1.6 Boolean Operators

Boolean operators, both unary and binary, require all of their operands a and b to type to **bool** in order to infer **bool** for the resulting expression. Rule 3.9 below captures both cases in a single formulation. The left formula covers the unary Boolean negation `!`, as well as the temporal operators and quantifiers described in Section 2.2.1. The right formula defines the binary Boolean operators conjunction `&`, disjunction `|`, implication `->`, and bi-implication `<->`, along with the temporal operators presented in Section 2.2.1. In all cases, the operands must type to **bool**, and the result is also assigned type **bool**.

$$\frac{\vdash a : \mathbf{bool}}{\begin{array}{l} \vdash \mathtt{!}a : \mathbf{bool} \\ \vdash \mathtt{F}\ a : \mathbf{bool} \\ \vdash \mathtt{G}\ a : \mathbf{bool} \\ \vdash \mathtt{X}\ a : \mathbf{bool} \\ \vdash \mathtt{forall(}a\mathtt{)} : \mathbf{bool} \\ \vdash \mathtt{exists(}a\mathtt{)} : \mathbf{bool} \end{array}} \qquad \frac{\vdash a : \mathbf{bool} \quad \vdash b : \mathbf{bool}}{\begin{array}{l} \vdash a\ \mathtt{\&}\ b : \mathbf{bool} \\ \vdash a\ \mathtt{|}\ b : \mathbf{bool} \\ \vdash a\ \mathtt{->}\ b : \mathbf{bool} \\ \vdash a\ \mathtt{<->}\ b : \mathbf{bool} \\ \vdash a\ \mathtt{U}\ b : \mathbf{bool} \\ \vdash a\ \mathtt{R}\ b : \mathbf{bool} \\ \vdash a\ \mathtt{W}\ b : \mathbf{bool} \\ \vdash \mathtt{<<}a\mathtt{>>}b : \mathbf{bool} \\ \vdash \mathtt{[[}a\mathtt{]]}b : \mathbf{bool} \end{array}} \tag{3.9}$$

3.1.7 Arithmetic Operators

Basic arithmetic operators are defined for both **int** and **range** types.

Rule 3.10 specifies that the result of an arithmetic operation will type to **int** if both of the operands a and b type to **int**.

$$\frac{\vdash a : \mathbf{int} \quad \vdash b : \mathbf{int}}{\begin{array}{l} \vdash a + b : \mathbf{int} \\ \vdash a - b : \mathbf{int} \\ \vdash a \star b : \mathbf{int} \\ \vdash a\ /\ b : \mathbf{int} \end{array}} \tag{3.10}$$

For operations performed purely using **range** types, the principles of *interval arithmetic* are employed to infer a resulting **range** $[l\,..\,u]$. The lower bound l and upper bound u of the resulting **range** are calculated based on the input ranges a and b. In each case, the bounds are determined by applying the respective operation to the bounds of a and b, ensuring the resulting range is the narrowest range that contains all possible outcomes of the operation [13]. These operators are formalized within rules 3.11 through 3.14.

$$\frac{\vdash a : [l_a\,..\,u_a] \quad \vdash b : [l_b\,..\,u_b] \quad l = l_a + l_b \quad u = u_a + u_b}{\vdash a + b : [l\,..\,u]} \tag{3.11}$$

$$\frac{\vdash a : [l_a\,..\,u_a] \quad \vdash b : [l_b\,..\,u_b] \quad l = l_a - u_b \quad u = u_a - l_b}{\vdash a - b : [l\,..\,u]} \tag{3.12}$$

$$\frac{\begin{array}{ll} \vdash a : [l_a\,..\,u_a] & \vdash b : [l_b\,..\,u_b] \\ p_1 = l_a \cdot l_b & p_2 = l_a \cdot u_b \\ p_3 = u_a \cdot l_b & p_4 = u_a \cdot u_b \\ l = \min(p_1, p_2, p_3, p_4) & u = \max(p_1, p_2, p_3, p_4) \end{array}}{\vdash a \star b : [l\,..\,u]} \tag{3.13}$$

Note that, in rule 3.14, the result of the divisions is rounded down ($\lfloor x \rfloor$) to perform integer division.

$$\frac{\begin{array}{ll} \vdash a : [l_a\,..\,u_a] & \vdash b : [l_b\,..\,u_b] \\ d_1 = \lfloor l_a \div l_b \rfloor & d_2 = \lfloor l_a \div u_b \rfloor \\ d_3 = \lfloor u_a \div l_b \rfloor & d_4 = \lfloor u_a \div u_b \rfloor \\ l = \min(d_1, d_2, d_3, d_4) & u = \max(d_1, d_2, d_3, d_4) \end{array}}{\vdash a \,/\, b : [l\,..\,u]} \tag{3.14}$$

3.1.8 Comparison Operators

Comparisons type to **bool** when both operands a and b type to **int**. The supported comparison operators are presented in rule 3.15. As is common practice within programming languages, the grammar of R-CHECK defines the symbols <= and >= for $\leq$ and $\geq$ respectively.

$$\frac{\vdash a : \textbf{int} \quad \vdash b : \textbf{int}}{\begin{array}{l} \vdash a < b : \textbf{bool} \\ \vdash a <= b : \textbf{bool} \\ \vdash a > b : \textbf{bool} \\ \vdash a >= b : \textbf{bool} \end{array}} \tag{3.15}$$

3.1.9 Equivalence

The subtype relation between every type and the type **any** presented in rule 3.7 allows for easy formulation of rule 3.16 to support equivalence expressions. The type system hereby allows the comparison between two arbitrary typed expressions. While the symbols = and == are both used to assert equivalence, != is used to assert inequality.

$$\frac{\vdash a : \texttt{any} \quad \vdash b : \texttt{any}}{\begin{array}{c}\vdash a = b : \texttt{bool} \\ \vdash a \;!= b : \texttt{bool} \\ \vdash a == b : \texttt{bool}\end{array}} \tag{3.16}$$

3.1.10 Assignments

Assignment statements as specified in rule 3.17 also leverage subtype relations to express that the receiving symbol a has to type to a supertype of the assigned value b in order to form a well-typed assignment. In ReCiPe, assignments are represented as either relabel statements (<−) or standard assignment statements (:=), both of which are treated equivalently for type checking purposes.

$$\frac{\vdash a : \tau_a \quad \vdash b : \tau_b \quad \tau_b <: \tau_a}{\begin{array}{c}\vdash a := b : \diamond \\ \vdash a <- b : \diamond\end{array}} \tag{3.17}$$

For future reference, rule 3.18 extends the assignment logic to lists of assignments. As a base case, the empty list ϵ is always well-typed. For non-empty lists, the rule ensures that the first assignment of the list, either $a := b$ or $a <- b$, is well-typed using rule 3.17 for single assignments. The tail of the list, $tail$, is recursively checked for validity to ensure that the entire list of expressions is well-typed.

$$\frac{}{\vdash \epsilon : \diamond} \qquad \frac{\vdash a := b : \diamond \quad \vdash tail : \diamond}{\vdash [a := b :: tail] : \diamond} \qquad \frac{\vdash a <- b : \diamond \quad \vdash tail : \diamond}{\vdash [a <- b :: tail] : \diamond} \tag{3.18}$$

3.1.11 Field Access

Fields, meaning variables and other symbols defined within the typing context Γ, will type to their stored type τ when used. Rule 3.19 states that an expression x will type to τ if the mapping $x \mapsto \tau$ is present in the typing context Γ. In certain scenarios, when referencing the property variables of a model, the additional symbol @ must be prepended in order to access the variable. For type checking purposes, both notation styles are treated equivalently, as underlined by the fact that the two conclusions share the same premise.

$$\frac{\Gamma(x) = \tau}{\begin{array}{l}\Gamma \vdash x : \tau \\ \Gamma \vdash \texttt{@}x : \tau\end{array}} \tag{3.19}$$

3.1.12 Guard Call

In ReCiPe, guard calls allow for evaluating Boolean expressions based on parameters. Rule 3.20 formalizes the typing of guard calls. It asserts that for a guard g with parameter types $[\tau_1, \ldots, \tau_n]$ in Ψ, each argument e_i must type to the corresponding type τ_i in the context Γ. The result of a guard call is always of type **bool**.

$$\frac{\Psi(g) = [\tau_1, \ldots, \tau_n] \quad \Gamma \vdash e_i : \tau_i \text{ for } i \in \{1, \ldots, n\}}{\Gamma, \Psi \vdash g(e_1, \ldots, e_n) : \mathtt{bool}} \tag{3.20}$$

3.1.13 Quantified Formulas

Each agent A_i $(1 \leq i \leq n)$ in the quantified formula possesses a local context stored in Σ, accessible via $\Sigma(A_i)$. The notation $\bigcap_{i=1}^{n} \Sigma(A_i)$ denotes the intersection of the contexts for all specified agent types, capturing precisely those fields $x \mapsto \tau$ that are common to every A_i. To reference this shared set of fields with the identifier k, the context Γ_k is constructed by adding a mapping $(k\text{-}x) \mapsto \tau$ for each such field. The identifiers $k\text{-}x$ are constructed by concatenating the identifier k, a dash (-), and each variable name x of the common fields of every A_i. This additional context, combined with the original context Γ, is then employed to type-check the expression e, which must have Boolean type. This works for both the `forall` and `foreach` keywords, as formalized in rule 3.21.

$$\frac{\Gamma_k = \{(k\text{-}x) \mapsto \tau \mid x \mapsto \tau \in \bigcap_{i=1}^{n} \Sigma(A_i)\} \quad (\Gamma \cup \Gamma_k), \Psi, \Sigma \vdash e : \mathtt{bool}}{\begin{array}{c}\Gamma, \Psi, \Sigma \vdash \texttt{forall}\ k\texttt{:}\ A_1 \texttt{|} \ldots \texttt{|} A_n\ \texttt{.}\ e : \mathtt{bool} \\ \Gamma, \Psi, \Sigma \vdash \texttt{exists}\ k\texttt{:}\ A_1 \texttt{|} \ldots \texttt{|} A_n\ \texttt{.}\ e : \mathtt{bool}\end{array}} \tag{3.21}$$

3.1.14 Specifications

For completeness, the rule 3.22 states that a list of specifications is well typed, if every specification of the list types to a Boolean. It also defines the base case for an empty list of specifications ϵ, which is unconditionally considered well-typed.

$$\frac{}{\Gamma, \Psi, \Sigma \vdash \epsilon : \diamond} \qquad \frac{\Gamma, \Psi, \Sigma \vdash \mathit{tail} : \diamond \quad \Gamma, \Psi, \Sigma \vdash e : \mathtt{bool}}{\Gamma, \Psi, \Sigma \vdash [\texttt{SPEC}\ e :: \mathit{tail}] : \diamond} \tag{3.22}$$

3.2 Typing Context

With most of the validity rules defined, the construction of the typing contexts can now begin. In the subsections 3.2.1 through 3.2.6, several rules are defined that describe the composition of the typing contexts.

3.2.1 Enumeration Types

Recall that Δ is a set of type names, including primitive (pre-defined) and user-defined types and that Γ is a set of mappings from variable names to their types. Rule 3.23 presents the base case for building up Δ and Γ. This rule states that, under the contexts of Δ and Γ, when an empty list of symbols ϵ is encountered, two contexts are returned that are identical to the original contexts.

$$\frac{}{\Delta, \Gamma \vdash \epsilon \Rightarrow (\Delta, \Gamma)} \tag{3.23}$$

Enum types are the only user-defined types to consider in ReCiPe. When encountered, enums are checked for validity in the context (Δ, Γ) using rule 3.24. The premises of the rule specify several conditions for valid enums. First, the names of the enum values must be distinct. This means that no two enum values (literals) can share the same name. Second, the enum values must be distinct from any existing identifiers in the variable context Γ, ensuring that no conflicts arise with previously defined variables. Moreover, the rule ensures that the enum name E either is not already present in Δ or, in the special case of the enum `channel`, the pre-defined type **`channel`** can be extended rather than requiring it to be absent from Δ.

$$\frac{\begin{array}{l}(E = \texttt{channel} \ \vee \ E \notin \Delta) \\ v_i \neq v_j \text{ for } i, j \in \{1, \ldots, n\}, i < j \quad v_i \notin \mathit{dom}(\Gamma) \text{ for } i \in \{1, \ldots, n\}\end{array}}{\Delta, \Gamma \vdash \mathsf{enum}\, E\ \{\, v_1, \ldots, v_n \,\} : \diamond} \tag{3.24}$$

To augment the contexts Δ and Γ with all enum definitions, rule 3.25 is used. It states that given the context of Δ and Γ, processing a list of enum declarations gives two new contexts, Δ' and Γ'. The rule proceeds by induction on the list. In the first step, the tail of the list *tail* is elaborated under the original contexts, yielding intermediate contexts Δ_t and Γ_t. In the second step, the head declaration $\mathsf{enum}\, E\, \{v_1, \ldots, v_n\}$ is validated within these intermediate contexts using rule 3.24. In the final step, the updated contexts are obtained by adding E to Δ_t to form Δ' and by extending Γ_t with the mappings $v_i \mapsto E$ for all $i = 1, \ldots, n$ to form Γ'. Under these premises, the rule produces the resulting contexts (Δ', Γ').

$$\frac{\begin{array}{ll}\Delta, \Gamma \vdash \mathit{tail} \Rightarrow \Delta_t, \Gamma_t & \Delta_t, \Gamma_t \vdash \mathsf{enum}\, E\ \{\, v_1, \ldots, v_n \,\} : \diamond \\ \Delta' = \Delta_t \cup \{E\} & \Gamma' = \Gamma_t \cup \{v_i \mapsto E\}_{i=1}^{n}\end{array}}{\Delta, \Gamma \vdash [\mathsf{enum}\, E\ \{\, v_1, \ldots, v_n \,\} :: \mathit{tail}] \Rightarrow (\Delta', \Gamma')} \tag{3.25}$$

3.2.2 Variables

Similar to the inclusion of enum types into the environment, variable mappings are incorporated into the variable context Γ through a recursive process over a list of variable declarations. This process relies on Rules 3.26 and 3.27, which respectively handle the base case and the inductive step.

Rule 3.26 captures the base case where the list of variable declarations is the empty list ϵ. In this situation, no new mappings are introduced and the original context Γ remains unchanged.

$$\frac{}{\Delta, \Gamma \vdash \epsilon \Rightarrow \Gamma} \tag{3.26}$$

Rule 3.27 defines the inductive step for processing a non-empty list of variable declarations of the form $[x : \tau :: \mathit{tail}]$. First, the tail of the list *tail* is processed recursively to yield an intermediate context Γ_{tail}. Next, the head declaration is considered valid if the variable x is not already declared within Γ_{tail}, and its type τ exists in the current typing context Δ. If these conditions hold, a new context Γ' is produced by extending Γ_{tail} with the mapping $x \mapsto \tau$.

$$\frac{\begin{array}{ll}\Delta, \Gamma \vdash \mathit{tail} \Rightarrow \Gamma_{\mathit{tail}} & x \notin \mathit{dom}(\Gamma_{\mathit{tail}}) \\ (\tau \in \Delta \ \vee \ (\tau = [l\,..\,u] \wedge l \leq u)) & \Gamma' = \Gamma_{\mathit{tail}} \cup \{x \mapsto \tau\}\end{array}}{\Delta, \Gamma \vdash [x : \tau :: \mathit{tail}] \Rightarrow \Gamma'} \tag{3.27}$$

3.2.3 Guards

In ReCiPe, guards serve as named Boolean predicates that may be parametrized with typed variables. Recall that Ψ is a set of mappings from guard names to an ordered list of types. In order to later type-check the signature of the guard calls, their declarations are added to the guard context Ψ. The process of checking and incorporating guards into Ψ is handled by three rules, beginning with the rule for guard validity (Rule 3.28), followed by the base case for an empty list of declarations (Rule 3.29), and concluding with the recursive rule for a non-empty list of guard declarations (Rule 3.30).

Rule 3.28 checks a single guard declaration against a set of conditions within the current context (Δ, Γ, Ψ). A guard declaration is considered valid if the guard name g is not already declared in Ψ, all types τ_i of the parameters p_i are present in the typing context Δ, none of the parameter names p_i is already used as an identifier within Γ, and the parameter names are pairwise distinct.

$$\frac{\begin{array}{ll}g \notin \mathsf{dom}(\Psi) & \tau_i \in \Delta \ \mathsf{for}\ i \in \{1, \ldots, n\} \\ p_i \notin \Gamma \ \mathsf{for}\ i \in \{1, \ldots, n\} & p_i \neq p_j \ \mathsf{for}\ i, j \in \{1, \ldots, n\}, i < j\end{array}}{\Delta, \Gamma, \Psi \vdash \texttt{guard}\ g\,(p_1 : \tau_1, \ldots, p_n : \tau_n)\ \texttt{:=}\ e\texttt{;}\ : \diamond} \tag{3.28}$$

Rule 3.29 defines the base case for processing a list of guard declarations. If the list is the empty list ϵ, the guard context Ψ remains unchanged.

$$\frac{}{\Delta, \Gamma, \Psi \vdash \epsilon \Rightarrow \Psi} \tag{3.29}$$

Lastly, rule 3.30 describes the recursive case for processing a list of guard declarations. It first processes the tail of the list *tail* to obtain an intermediate context Ψ_{tail}. The guard parameters (in essence, a list of variable declarations) are used to form a temporary context Γ_{guard}, re-using the variable declaration rule (Rule 3.27). Then, the head declaration `guard` $g(p_1 : \tau_1, \ldots, p_n : \tau_n) := e;$ is validated using rule 3.28. Finally, the expression e in the guard body is type-checked under the context Γ_{guard} and must be of type **`bool`**. If all of these conditions are satisfied, the resulting guard context Ψ' is formed by extending Ψ_{tail} with the mapping of the guard name to the signature of its parameter list $g \mapsto [\tau_1, \ldots, \tau_n]$.

$$\frac{\begin{array}{ll} \Delta, \Gamma, \Psi \vdash tail \Rightarrow \Psi_{tail} & \Delta, \Gamma \vdash [p_1 : \tau_1, \ldots, p_n : \tau_n] \Rightarrow \Gamma_{guard} \\ \Delta, \Gamma, \Psi_{tail} \vdash \texttt{guard}\ g\ \texttt{(}p_1 : \tau_1, \ldots, p_n : \tau_n\texttt{)}\ \texttt{:=}\ e\texttt{;}\ : \diamond & \\ \Psi' = \Psi_{tail} \cup \{g \mapsto [\tau_1, \ldots, \tau_n]\} & \Gamma_{guard} \vdash e : \mathbf{\texttt{bool}} \end{array}}{\Delta, \Gamma, \Psi \vdash [\texttt{guard}\ g\ \texttt{(}p_1 : \tau_1, \ldots, p_n : \tau_n\texttt{)}\ \texttt{:=}\ e\texttt{;}\ :: tail] \Rightarrow \Psi'} \tag{3.30}$$

3.2.4 Processes

Formally, each process consist of a unique name n, a Boolean guard predicate ψ, and takes one of several atomic forms (send `!`, receive `?`, `GET` or `SUPPLY`). However in the implementation this is not the case, as the name n may be left out. In this case the variable context Γ simply is not extended. Rule 3.31 shows how processing an atomic action extends the variable context Γ with a mapping from the process name n to Boolean if the identifier n is not already present in Γ.

$$\frac{n \notin \mathsf{dom}(\Gamma) \quad \Gamma' = \{n \mapsto \mathbf{\texttt{bool}}\}}{\begin{array}{l} \Gamma \vdash n\texttt{:}\ \texttt{\{}\psi\texttt{\}}\ c\texttt{!}\ \texttt{(}g\texttt{)}\ \texttt{(}D\texttt{)}\ \texttt{[}U\texttt{]} \Rightarrow \Gamma' \\ \Gamma \vdash n\texttt{:}\ \texttt{\{}\psi\texttt{\}}\ c\texttt{?}\ \texttt{[}U\texttt{]} \Rightarrow \Gamma' \\ \Gamma \vdash n\texttt{:}\ \texttt{\{}\psi\texttt{\}}\ \texttt{GET@}\texttt{(}l\texttt{)}\ \texttt{(}D\texttt{)}\ \texttt{[}U\texttt{]} \Rightarrow \Gamma' \\ \Gamma \vdash n\texttt{:}\ \texttt{\{}\psi\texttt{\}}\ \texttt{SUPPLY@}\texttt{(}l\texttt{)}\ \texttt{(}D\texttt{)}\ \texttt{[}U\texttt{]} \Rightarrow \Gamma' \end{array}} \tag{3.31}$$

Since processes can be combined using several operators (`;`, `+`, `rep`, `()`), the formulas in rule 3.32 are used to propagate the addition of process names to the context Γ through a complex process definition. Here, p and q each denote a process expression.

$$\frac{\Gamma \vdash p \Rightarrow \Gamma_p \quad \Gamma_p \vdash q \Rightarrow \Gamma'}{\begin{array}{c}\Gamma \vdash p\texttt{;}\ q \Rightarrow \Gamma' \\ \Gamma \vdash p + q \Rightarrow \Gamma'\end{array}} \qquad \frac{\Gamma \vdash p \Rightarrow \Gamma'}{\begin{array}{c}\Gamma \vdash \texttt{rep}\ p \Rightarrow \Gamma' \\ \Gamma \vdash \texttt{(}p\texttt{)} \Rightarrow \Gamma'\end{array}} \tag{3.32}$$

Before extending the typing context, processes must be checked for validity ($\diamond$). The following rules 3.33 through 3.36 follow a similar principle to the rules used to gather the names of the processes. Rule 3.33, rule 3.34, and rule 3.35 are used to type-check the atomic actions. In this case, some actions cannot share the same premises because of the structural differences or differing typing restrictions.

All type checking of processes, and indeed every subsequent type check, is carried out under the context (Γ, Ψ) to enable the use of guards. In a typical ReCiPe model, guards appear only in the send guard expression g of a send action, which types to Boolean ($\Gamma, \Psi \vdash g : \texttt{bool}$). In the formal definition presented here, however, guards may be used wherever the syntax allows.

$$\frac{\Gamma, \Psi \vdash \psi : \texttt{bool} \quad \Gamma, \Psi \vdash c : \texttt{channel} \quad \Gamma, \Psi \vdash g : \texttt{bool} \quad \Gamma, \Psi \vdash D : \diamond \quad \Gamma, \Psi \vdash U : \diamond}{\begin{array}{c}\Gamma, \Psi \vdash n\texttt{:}\{\psi\}\ c\texttt{!}\ \texttt{(}g\texttt{)}\ \texttt{(}D\texttt{)}\ \texttt{[}U\texttt{]} : \diamond \\ \Gamma, \Psi \vdash n\texttt{:}\{\psi\}\ c\texttt{?}\,\texttt{[}U\texttt{]} : \diamond\end{array}} \tag{3.33}$$

$$\frac{\Gamma, \Psi \vdash \psi : \texttt{bool} \quad \Gamma, \Psi \vdash l : \texttt{bool} \mid \texttt{location} \quad \Gamma, \Psi \vdash D : \diamond \quad \Gamma, \Psi \vdash U : \diamond}{\Gamma, \Psi \vdash n\texttt{:}\{\psi\}\ \texttt{GET@}\texttt{(}l\texttt{)}\ \texttt{(}D\texttt{)}\ \texttt{[}U\texttt{]} : \diamond} \tag{3.34}$$

$$\frac{\Gamma, \Psi \vdash \psi : \texttt{bool} \quad \Gamma, \Psi \vdash l : \texttt{location} \quad \Gamma, \Psi \vdash D : \diamond \quad \Gamma, \Psi \vdash U : \diamond}{\Gamma, \Psi \vdash n\texttt{:}\{\psi\}\ \texttt{SUPPLY@}\texttt{(}l\texttt{)}\ \texttt{(}D\texttt{)}\ \texttt{[}U\texttt{]} : \diamond} \tag{3.35}$$

The formulas in rule 3.36 is again used to propagate through complex process definitions.

$$\frac{\Gamma \vdash p : \diamond \quad \Gamma \vdash q : \diamond}{\begin{array}{c}\Gamma \vdash p\texttt{;}\ q : \diamond \\ \Gamma \vdash p + q : \diamond\end{array}} \qquad \frac{\Gamma \vdash p : \diamond}{\begin{array}{c}\Gamma \vdash \texttt{rep}\ p : \diamond \\ \Gamma \vdash \texttt{(}p\texttt{)} : \diamond\end{array}} \tag{3.36}$$

3.2.5 Agents

In ReCiPe, each agent A is declared with a unique name and consists of four components: an initialization condition $A.\texttt{init}$, a relabelling map $A.\texttt{relabel}$, a receive-guard predicate $A.\texttt{receive-guard}$, and a repeating process $A.\texttt{repeat}$.

Recall that Σ is a set of mappings from agent names to typing contexts following the structure of Γ. Before an agent can be added to Σ it must satisfy the validity constraints given by rule 3.37. Rule 3.37 ensures that A does not already appear in the agent environment Σ, that its initialization and receive-guard expressions both have Boolean type, that its list of relabelling assignments is valid under (Γ, Ψ), and that its process $A.\texttt{repeat}$ type-checks under (Γ, Ψ).

$$\frac{\begin{array}{ll} A \notin \mathsf{dom}(\Sigma) & \\ \Gamma \vdash A.\texttt{init} : \textbf{bool} & \Gamma, \Psi \vdash A.\texttt{relabel} : \diamond \\ \Gamma \vdash A.\texttt{receive-guard} : \textbf{bool} & \Gamma, \Psi \vdash A.\texttt{repeat} : \diamond \end{array}}{\Delta, \Gamma, \Psi, \Sigma \vdash \texttt{agent}\ A : \diamond} \tag{3.37}$$

The agent environment Σ is constructed by processing a sequence of agent declarations. Rule 3.38 handles the base case of an empty list by leaving Σ unchanged.

$$\frac{}{\Delta, \Gamma, \Psi, \Sigma \vdash \epsilon \Rightarrow \Sigma} \tag{3.38}$$

When a non-empty list of agents is encountered, rule 3.39 describes how to extend Σ. First, the tail of the list *tail* is evaluated to yield Σ_{tail}. Next, the local variable declarations $A.\texttt{local}$ are elaborated under Γ to produce a local context Γ_{loc}, after which the agent A itself is checked for validity under $(\Delta, (\Gamma \cup \Gamma_{loc}), \Psi, \Sigma)$. The repeating process $A.\texttt{repeat}$ is then processed under $(\Delta, (\Gamma \cup \Gamma_{loc}), \Psi)$ to obtain Γ_{rep} as described in the section above. Finally, the updated environment Σ' is formed by uniting Σ_{tail} with a mapping from A to the union of its local variable context Γ_{loc}, its repeat-process context Γ_{rep}, and the implicit `automaton-state` field of type **`int`**.

$$\frac{\begin{array}{ll} \Delta, \Gamma, \Psi, \Sigma \vdash \mathit{tail} \Rightarrow \Sigma_{tail} & \Delta, \Gamma \vdash A.\texttt{local} \Rightarrow \Gamma_{loc} \\ \Delta, (\Gamma \cup \Gamma_{loc}), \Psi, \Sigma \vdash \texttt{agent}\ A : \diamond & \Delta, (\Gamma \cup \Gamma_{loc}), \Psi \vdash A.\texttt{repeat} \Rightarrow \Gamma_{rep} \\ \Sigma' = \Sigma_{tail} \cup \{A \mapsto (\Gamma_{loc} \cup \Gamma_{rep} \cup \{\texttt{automaton-state} \mapsto \texttt{\textbf{int}}\})\} & \end{array}}{\Delta, \Gamma, \Psi, \Sigma \vdash [\texttt{agent}\ A :: \mathit{tail}] \Rightarrow \Sigma'} \tag{3.39}$$

3.2.6 System

The system definition in a ReCiPe model is a list of agent instances, split by the parallel composition operator (`||`). A valid instantiation as described in rule 3.40 is formed by calling the agent name A as a constructor with two parameters: the instance name i and an initialization constraint expression e. The instance name must not exist in the typing context Γ and the instantiation expression must type to Boolean under the agent's internal context $\Sigma(A)$.

$$\frac{i \notin \mathsf{dom}(\Gamma) \quad \Sigma(A) \vdash e : \texttt{\textbf{bool}}}{\Gamma, \Sigma \vdash A\,(i, e)\ : \diamond} \tag{3.40}$$

As for other context definitions, a base case for the recursive process is provided in rule 3.41.

$$\frac{}{\Gamma, \Sigma \vdash \epsilon \Rightarrow \Gamma} \tag{3.41}$$

Using rule 3.42, the list of all instances is evaluated to form the typing context Γ'. The tail of the list *tail* is evaluated to form Γ_{tail}, leaving only the instance $A\,(i, e)$ to be processed. Rule 3.40 is employed to check the validity of the instance. Next Γ_i is formed by concatenating the instance name i, a dash $(-)$, and each variable name x in the agent's internal context $\Sigma(A)$ to create identifiers $i{-}x$, each of which is mapped to the same type τ that x had in $\Sigma(A)$. Finally, the resulting context Γ' is defined by extending the tail context Γ_{tail} with the mappings in Γ_i and the binding $\{i \mapsto \texttt{\textbf{location}}\}$, thereby assigning the instance name i the **`location`** type.

$$\frac{\begin{array}{cc} \Gamma, \Sigma \vdash A\,(i, e)\ : \diamond & \Gamma_i = \{(i{-}x) \mapsto \tau \mid x \mapsto \tau \in \Sigma(A)\} \\ \Gamma, \Sigma \vdash tail \Rightarrow \Gamma_{tail} & \Gamma' = \Gamma_i \cup \{i \mapsto \texttt{\textbf{location}}\} \cup \Gamma_{tail} \end{array}}{\Gamma, \Sigma \vdash [A\,(i, e)\ :: tail] \Rightarrow \Gamma'} \tag{3.42}$$

3.3 Model Rule

Bringing everything together, the entry point of the type system is given by rule 3.43. It states that a ReCiPe model M is well typed $(\vdash M : \diamond)$ precisely when its full typing context $(\Delta, \Gamma, \Psi, \Sigma)$ is built up in sequence from the model's enumeration declarations $M.\texttt{enums}$, its global variable declarations $M.\texttt{msgStructs} \cdot M.\texttt{propVars}$, its guard declarations $M.\texttt{guards}$, its agent declarations $M.\texttt{agents}$, and finally its system definition $M.\texttt{system}$.

The initial typing contexts Γ_{init}, Ψ_{init}, and Σ_{init} are declared empty while Δ_{init} is prefilled with ReCiPe's built-in types. The final premise then requires that the model's specifications $M.\texttt{spec}$ type-check under the completed context (Γ, Ψ, Σ), using rule 3.22.

$$\frac{\begin{array}{lr} \Delta_{init} = \{\texttt{\textbf{bool}}, \texttt{\textbf{int}}, \texttt{\textbf{location}}, \texttt{\textbf{channel}}\} & \Delta_{init}, \Gamma_{init} \vdash M.\texttt{enums} \Rightarrow \Delta, \Gamma_{enum} \\ \Gamma_{init} = \{\} & \Delta, \Gamma_{enum} \vdash M.\texttt{msgStructs} \cdot M.\texttt{propVars} \Rightarrow \Gamma_{vars} \\ \Psi_{init} = \{\} & \Delta, \Gamma_{vars}, \Psi_{init} \vdash M.\texttt{guards} \Rightarrow \Psi \\ \Sigma_{init} = \{\} & \Delta, \Gamma_{vars}, \Psi \vdash M.\texttt{agents} \Rightarrow \Sigma \\ \Gamma, \Psi, \Sigma \vdash M.\texttt{spec} : \diamond & \Gamma_{vars}, \Sigma \vdash M.\texttt{system} \Rightarrow \Gamma \end{array}}{\vdash M : \diamond} \tag{3.43}$$

CHAPTER 4

Implementation in R-CHECK

This chapter contains a quick summary of the process of integrating the theoretical type system into the existing R-CHECK framework. First, the Typir-based type checker implementation is explored. What follows are some details about the integration of the type checker into the Langium architecture.

4.1 Typir-Based Type Checker

The functions provided by the Typir library enable an easy translation of the formal rules of the type system to a functional implementation. Primitive types can be defined within the initialization function of the type checker class. Additionally, it may hold definitions of operators, static inference rules, and type constraints on certain language features. The type inference for user defined types like enums happens in a function that is evoked once for each AST node during the tree walk. This function also handles other dynamic rules like guard and agent definitions with their inference rules for members and calls.

To illustrate this translation from the formal type system to a concrete implementation, Listing 4.1 shows how the primitive type **`channel`** is implemented using the factory utilities provided by Typir. The primitive type is created with the name `"channel"` and static inference rules are applied to the new type before finishing the configuration chain using `.finish()`. Inference rules can either be defined by passing a type guard into the `filter` parameter of the `.inferenceRule()` call, or by specifying the language keys of AST nodes and additionally supplying the `matching` parameter with a function that acts as a predicate on the specified nodes. Specifically, the inference rules at lines 4–5 in Listing 4.1 specify that AST nodes of type `ChannelRef` and `Broadcast` will unconditionally be of type **`channel`**. Examining the grammar definition for these nodes in Listing 4.2 shows that the two inference rules correspond exactly to the axioms presented within subsection 3.1.3 of the formal type system definition. The inference rule at lines 6–10 in Listing 4.1 corresponds to the rule for building the variable context Γ within subsection 3.2.2. The

```
1  const typeChannel = typir.factory.Primitives.create({
2    primitiveName: "channel",
3  })
4    .inferenceRule({ filter: isChannelRef })
5    .inferenceRule({ filter: isBroadcast })
6    .inferenceRule({
7      languageKey: [Local, Param, MsgStruct, PropVar],
8      matching: (node: Local | Param | MsgStruct | PropVar) =>
9        node.customType?.ref?.name === "channel",
10   })
11   .inferenceRule({
12     languageKey: Enum,
13     matching: (node: Enum) => node.name === "channel",
14   })
15   .finish();
```

Listing 4.1: Implemenation of the primitive type **channel** using the factory utility of Typir.

```
BaseExpr infers CompoundExpr:
// [...]
    | {infer ChannelRef} currentChannel='chan'
// [...]
    | {infer Broadcast} value="*"
// [...]
    ;
```

Listing 4.2: R-CHECK grammar definitions for nodes related to the **channel** type.

implementation ensures that any variable declared in a `Local`, `Param`, `MsgStruct`, or `PropVar` node is assigned the type **channel** if its referenced `customType` has the name `"channel"`. The last inference rule at lines 11–14 in Listing 4.1 has no direct counterpart within the formal type system. The rule exists to infer the type **channel** in an enum definition node with the name `"channel"`. The propagation of the **channel** type to the literals of the enum definition is handled elsewhere. For a complete listing of the implementation, please refer to appendix (page 49).

When an inconsistency or violation is detected during type inference, the type checker generates structured messages that include a clear description of the issue, a reference to the affected language node and property, and a severity level. Each message distinguishes between errors and warnings. Errors indicate that the program is ill-typed and cannot proceed while warnings highlight potential logical issues that do not prevent execution.

4.2 Langium Integration

As described earlier, the original R-CHECK pipeline performs parsing of the ReCiPe grammar, construction of AST, resolution of cross-references and basic structural validations, and finally the translation of the validated model into nuXmv specifications. The static type checker is inserted immediately after the basic semantic validations and before any model-to-code transformation takes place. At that point the parser has produced a fully resolved AST, Langium has identified all syntax and structural errors, but no type information has yet been enforced.

To enable type checking, several targeted grammar modifications were necessary. These changes fall into three categories. The fist category are changes that added or updated the identifiers of parser rule properties in order to reference these properties during type checking. The second category include the prevention of ambiguities within parser rules that formerly defined an `ID` terminal instead of a cross-reference to another parser rule or re-used identifiers for different symbols. The third category involves structural changes to the grammar that introduce abstract rules, allowing related parser rules to be grouped under a common concept. This makes it possible to refer to multiple concrete rules collectively during implementation. The complete diff of the grammar changes can be found in the appendix (page 69).

The type checker itself integrates with Langium by adding a new service class to the type `AddedServices` of the language module file. In the case of R-CHECK, the field `typir` of type `TypirLangiumServices<RCheckAstType>` was added to `RCheckAddedServices`. The actual type system class `RCheckTypeSystem` implements the interface `LangiumTypeSystemDefinition<RCheckAstType>` that is provided by the typir-langium binding package. This class is then injected into the `createRCheckModule` function according to the `RCheckAddedServices` type from before. The typir-langium package also provides the matching function `createTypirLangiumServices` that returns an instance of `TypirLangiumServices<RCheckAstType>`.

At this point, the type system is initialized together with the other validation services that are implemented into Langium. Typir reports errors and warnings to Langium's build-in `ValidationProblemAcceptor` service, which in turn displays these messages like the other messages already provided for parsing errors.

CHAPTER 5

Evaluation

The following section will demonstrate the behaviour of the type checker in a multitude of warning and error scenarios. Afterwards, the limitations of the implementation in its current state will be highlighted in Section 5.2.

5.1 Demonstrative Examples

A minimal syntactically correct project will serve as a testing ground. The program, shown in Listing 5.1, extends the `enum channel` with two additional communication methods (literals). Next, the property variable `friendly : bool` is declared as a Boolean. The sole agent within the model is defined with minimal examples for most of its properties. Lastly, a system of two agents is defined, each without initial constraints (`true`).

Additionally, the testing setup employs VS Code as the code editor, with the R-CHECK extension enabled. In its current form, the code in Listing 5.1 is well typed and the code editor shows no warnings or errors. In the following sections, the program is altered with ill-typed scenarios to showcase the capabilities of the type checker.

5.1.1 Primitive Types and Operators

Since many of the elements within ReCiPe type to Boolean, comparison operators are often used. To produce a warning, the `init` field of the agent definition is set to a semantically questionable value of `3 == true`. When comparing the two literals of differing types, the type checker issues a warning as shown in Figure 5.1. The warning states, that the result of the comparison will be constant (`false`) because of the type mismatch. One can see that the message also provides the inferred types of the operands in the correct order, which is especially helpful when working with variables.

Another kind of type mismatch occurs when the operands of arithmetic operators such as '+' do not have the correct type. To produce this error, the expression `(1 + mood) == 1` is

```
enum channel {radio, speaker}

property-variables: friendly : bool

agent Droid
    local: status : int, mood : bool
    init: true
    relabel:
        friendly <- mood
    receive-guard: true
    repeat: {true} *? [mood := true]

system = Droid(R2D2, true) || Droid(C3PO, true)
```

Listing 5.1: Minimalistic, syntactically correct ReCiPe model.

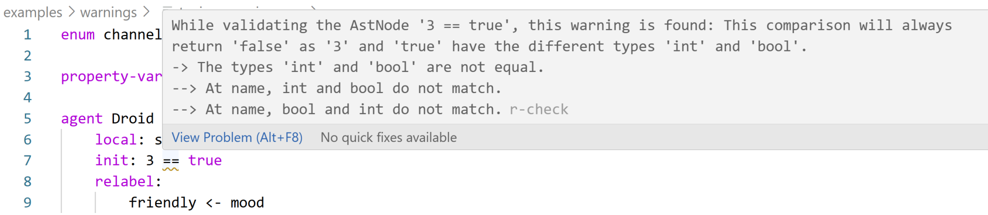


Figure 5.1: Comparison type mismatch warning.

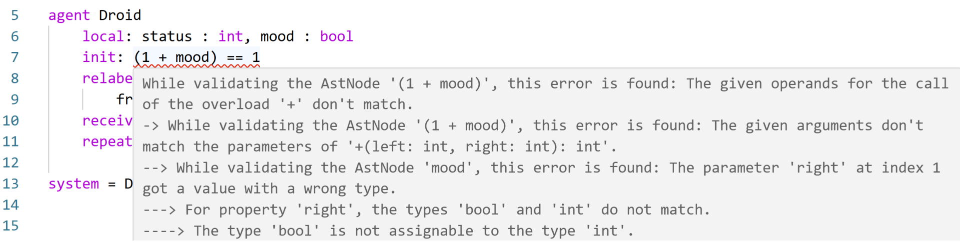


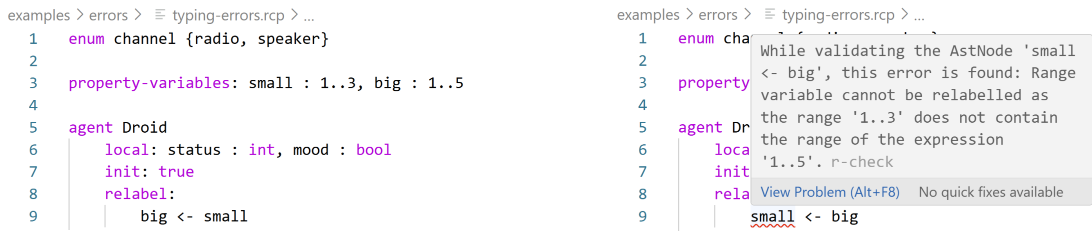
Figure 5.2: Operand type mismatch error.

set as the agent's initialization condition. An error message for this scenario can be seen in Figure 5.2. Structurally similar error messages are provided for all other operators that can be used within ReCiPe. This example also shows that the type of the variable `mood` is correctly retrieved from the type context.

5.1.2 Assignments

Recall that ReCiPe provides two assignment operators: `<-` and `:=`. For type checking purposes, both operators are processed identically. The **`relabel`** property of the agent is updated to read `friendly <- radio`. The error shown in Figure 5.3 explains in detail why this operation is not permitted.

Figure 5.3: Invalid assignment operation error.



(a) Relabel wide range with narrow range.

(b) Relabel narrow range with wide range.

Figure 5.4: Valid and invalid assignments between ranges.

The subtype relation of the integer and range types can also be showcased using assignments. A variable of type **`int`** can be relabelled with an expression of type `range` but not the other way around. Furthermore, a value of type `1..3` may be assigned to a variable declared as `1..5`, whereas the reverse assignment is disallowed. Figure 5.4 shows a valid case (Figure 5.4a), alongside the invalid one (Figure 5.4b). To produce the illustrated error, two new property variables have been introduced into the program: `small : 1..3` and `big : 1..5`.

5.1.3 Integer and Range Arithmetic

Generally, the type checker allows arithmetic operations to be performed with both integer and range values (or any combination thereof). For the range type, additional static checks compute the resulting range bounds and emit warnings when a comparison or assignment can never succeed or may exceed declared bounds.

Building upon the example presented in Figure 5.4, it can be shown that the range bounds of entire expressions containing range and integer values will be inferred and correctly compared to the type of the receiving variable. Suppose `small` is declared as `small : 1..3` and `big` as `big : 1..5`. The relabelling assignment `small <- ((big + 1) / 2)` type-checks successfully because adding one to any value in `1..5` and dividing by two (integer division) always yields a result within the interval `1..3`.

For comparisons, the checking of range bounds is used to warn the programmer about static outcomes. An example is shown in Figure 5.5.

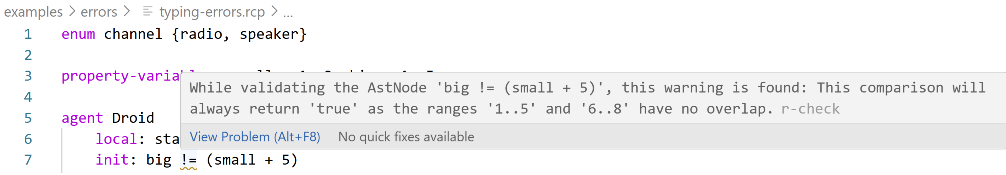


Figure 5.5: Comparison between two ranges with no overlap warning.

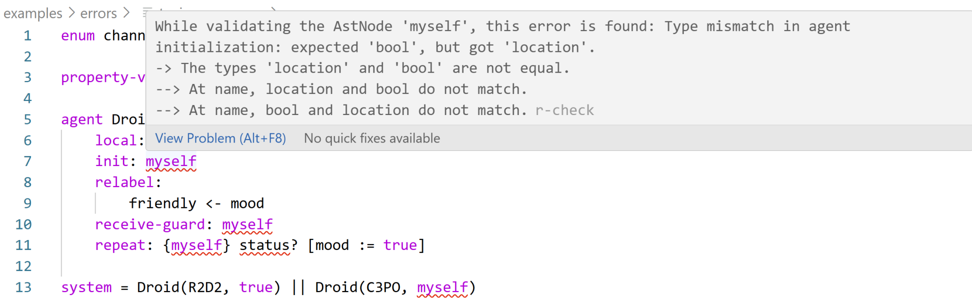


Figure 5.6: Type errors for fields with predetermined types.

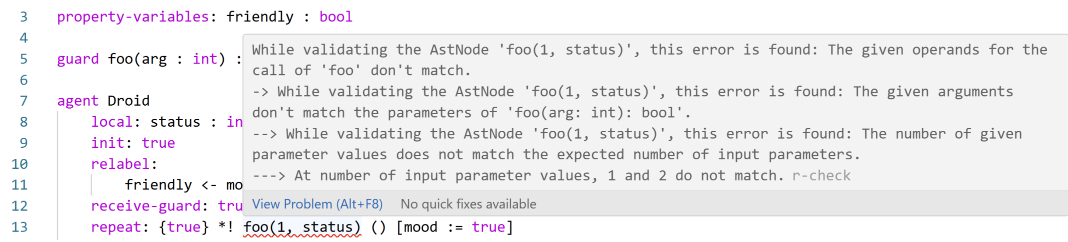
5.1.4 Agent, Process, and System

Certain parts of an agent definition, the repeating process of an agent, and the system definition must use predetermined types. The type checker has simple conditions built in for these parts of the program. The example in Figure 5.6 highlights some of the fields that demand correctly typed expressions. In this case, errors arise from substituting the expected Boolean expressions with the location literal `myself`. The only exception is the symbol `status` in the message-receive instruction `{myself} status? [mood := true]`, where the usage of an identifier is statically enforced and the required type is `channel`. Only the first error message is shown since the subsequent errors follow the same pattern.

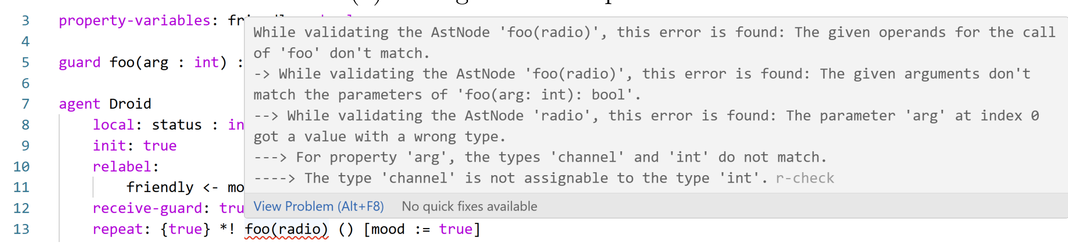
5.1.5 Guards

As previously demonstrated, ReCiPe allows the use of user-defined functions called guards. A very simple guard definition to use as an example would be `guard foo(arg : int) := true;`. The other alteration made to the example program is found within the agent process, which now consists of a send command that reads `repeat: {true} *! foo(1) () [mood := true]`. The guard is called within the message-send and produces no type error, since the types of the parameter list of the declaration and the call match exactly.

When changing the number or the types of the parameters within either the guard declaration or call, the type checker will present errors to the programmer. The error messages for incorrect guard calls are shown in Figure 5.7. Figure 5.7a illustrates the error produced when the number of arguments does not match the declaration, while Figure 5.7b demonstrates an error caused by a mismatch of types for guard parameters.



(a) Wrong number of parameters.



(b) Parameters with wrong types.

Figure 5.7: Type errors when working with guards.

5.1.6 Specifications

Similar to other fields of a ReCiPe model, specifications defined after the `SPEC` keyword must type to Boolean. Unlike ordinary expressions, specifications allow the use of additional temporal operators such as 'next' (`X expr`) or 'diamond' (`<<expr>>expr`). All of these operators behave as unary or binary Boolean operators and will function similarly to the operators that are described in the previous sections.

Since specifications can be used to reason about the inner workings of agents, the programmer is able to access their internal members such as local variables or named processes. The type checker is able to provide type safety in this scenario by inferring the type of any expression consisting of an agent instance's identifier followed by the name of a local variable or process. For example, in the running example from Listing 5.1, writing `SPEC F R2D2-status == 1` is perfectly valid, since the type checker can infer the type `int` of the local variable `status`.

ReCiPe also supports quantified formulas using the syntax `SPEC forall x : Droid . F x-status == 1`. This is not only possible for a single agent type, but allows for reasoning about a combination of multiple agent types. Suppose the example program is extended with the agent definition in Listing 5.2.

Now, quantified formulas such as `forall x : Droid | Robot` can make use of both agent types at once, forming a composite type. Recall that the agent type `Droid` has the two local variables `status : int`, `mood : bool`. The agent type `Robot` shares the variable `status : int` (in name and type), but has no property `mood`. Additionally, agents of type `Robot` include a labelled process called 'process'. Labelled processes can also be accessed like local variables and type to Boolean. When now trying to access a field such as

```
agent Robot
    local: status : int
    init: true
    relabel:
        friendly <- false
    receive-guard: true
    repeat: process: {true} *? []
```

Listing 5.2: Additional agent type for the example program.

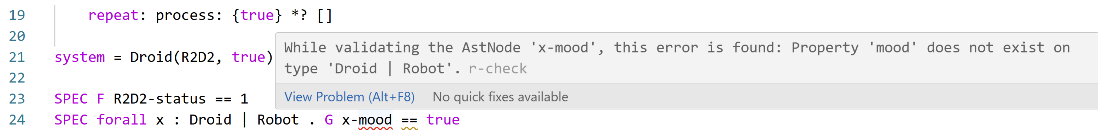


Figure 5.8: Error when trying to access a field that is not present in an composite agent type.

`SPEC forall x : Droid | Robot . G x-status == 1` of the composite agent type, the type checker will emit no warnings, since `status` is present (and is an `int`) in both `Droid` and `Robot`. Should the programmer attempt to access a field that is only present in one of the agents, the error presented in Figure 5.8 is displayed.

5.2 Limitations

In its current form, the implementation of the type system has a few limitations. Firstly, the payload being processed during a send or receive process remains untyped. This means that the programmer may specify a payload such as `(status := 1)` in a send command, that will never be considered in a receive command on the same channel. Such mismatched command pairs, while not causing runtime errors, will never allow a message to be exchanged during execution. If not considered, this can potentially lead to a *deadlock*, where no further message exchange is possible in a system.

Another aspect not enforced by the type system is the matching of send and receive commands in general. Since channelled communication in ReCiPe is blocking, the system may go into a deadlock state when all agents commence a send on a channel that no other agent is willing to receive messages on. At present, there is no static analysis or warning to alert the programmer to this possibility. Detecting these potential deadlocks without resorting to intensive verification techniques such as model checking is an interesting challenge.

A final limitation becomes apparent when initializing an agent (for example, with `Droid (R2D2, true)`). Although the type checker requires the second argument to be a `bool`, it performs no further semantic checks on that expression. In practice, an initialization expression should refer only to local variables declared on the agent, but currently nothing prevents arbitrary Boolean expressions from being used. This may be solved in the future by a more in-depth analysis of the variables being referenced in these expressions.

CHAPTER 6

Conclusion

6.1 Summary of Contributions

This work has presented the design and implementation of a static type system for the ReCiPe formalism, fully integrated into the existing R-CHECK framework. The type system formalizes rules for primitive types, operators, guards, and the key constructs of ReCiPe, enabling early detection of type errors and supporting safer specification of reconfigurable MASs. The integration with Typir and Langium demonstrates how formal definitions can be systematically translated into practical tooling for domain experts. The evaluation shows that the resulting type checker provides clear feedback for common and subtle error scenarios, contributing to more robust modelling workflows.

6.2 Future Work

A key direction for future work is to implement more advanced consistency checks, such as verifying payload alignment between send and receive processes and detecting potential deadlocks caused by blocking channel communication. These improvements would further strengthen the reliability and robustness of ReCiPe and R-CHECK for designing reconfigurable MASs.

List of Figures

5.1 Comparison type mismatch warning. 30
5.2 Operand type mismatch error. 30
5.3 Invalid assignment operation error. 31
5.4 Valid and invalid assignments between ranges. 31
5.5 Comparison between two ranges with no overlap warning. 32
5.6 Type errors for fields with predetermined types. 32
5.7 Type errors when working with guards. 33
5.8 Error when trying to access a field that is not present in an composite agent type. 34

Listings

2.1 Type inference examples in TypeScript. 4
2.2 Implicit type casting of number to string in TypeScript. 5
2.3 Excerpt from the grammar definition of R-CHECK. 9
4.1 Implemenation of the primitive type **`channel`** using the factory utility of Typir. 26
4.2 R-CHECK grammar definitions for nodes related to the **`channel`** type. 26
5.1 Minimalistic, syntactically correct ReCiPe model. 30
5.2 Additional agent type for the example program. 34
1 Full implementation of the R-CHECK type system. 49
2 Full implementation of used utility functions. 62
3 Full listing of the grammar changes. 69

Glossary

Langium A TypeScript framework for developing domain-specific languages and editors. vii, ix, 4, 7–10, 25, 27, 37

LTOL An extension of linear temporal logic (LTL) with extra operators, used in ReCiPe to specify system properties. vii, ix, 6, 7

nuXmv A symbolic model checker for verifying finite- and infinite-state systems. 7, 27

R-CHECK A tool for parsing, simulating, and verifying ReCiPe models. vii, ix, 1–4, 6–9, 15, 25–27, 29, 37, 41

ReCiPe A formalism for modelling reconfigurable multi-agent systems. vii, ix, 1–3, 5–7, 11, 13, 16–19, 21–23, 27, 29, 30, 32–34, 37, 41

TypeScript A statically typed programming language that builds on JavaScript. 1, 4, 5, 8, 41

Typir A TypeScript library for building static type systems and performing type inference. vii, ix, 9, 10, 25–27, 37, 41

Acronyms

AST abstract syntax tree. 3–5, 7–10, 12, 25, 27

CLI command line interface. 8

DSL domain-specific language. vii, ix, 1, 8

GPL general-purpose language. 1, 7

LSP Language Server Protocol. 8

LTL Linear Temporal Logic. 6

MAS multi-agent system. vii, ix, 1, 2, 37

VS Code Visual Studio Code. vii, ix, 7, 8, 29

Bibliography

[1] Y. A. Alrahman, S. Azzopardi, L. Di Stefano, and N. Piterman, "Language support for verifying reconfigurable interacting systems," *International Journal on Software Tools for Technology Transfer*, vol. 25, no. 5-6, pp. 765–784, Dec. 2023. [Online]. Available: https://link.springer.com/10.1007/s10009-023-00729-8

[2] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, and S. Bansal, *Compilers, principles, techniques, and tools*, updated second edition ed. Uttar Pradesh, India: Pearson India Education Services Pvt. Ltd, 2023.

[3] K. D. Cooper and L. Torczon, *Engineering a compiler*, 2nd ed. Amsterdam Heidelberg: Elsevier, Morgan Kaufmann, 2012.

[4] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, Oct. 1977, pp. 46–57, iSSN: 0272-5428. [Online]. Available: https://ieeexplore.ieee.org/document/4567924

[5] M. T. Delgado, "Eclipse Langium | projects.eclipse.org," Jun. 2023. [Online]. Available: https://projects.eclipse.org/projects/ecd.langium

[6] M. Spönemann, "Eclipse Langium | projects.eclipse.org," May 2023. [Online]. Available: https://projects.eclipse.org/proposals/eclipse-langium

[7] D. Barros, S. Peldszus, W. K. G. Assunção, and T. Berger, "Editing support for software languages: implementation practices in language server protocols," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. Montreal Quebec Canada: ACM, Oct. 2022, pp. 232–243. [Online]. Available: https://dl.acm.org/doi/10.1145/3550355.3552452

[8] Microsoft, "Tools supporting the LSP," 2024. [Online]. Available: https://microsoft.github.io/language-server-protocol/implementors/tools/

[9] M. Rudolph, "Langium's workflow," 2024. [Online]. Available: https://langium.org/docs/learn/workflow/

[10] TypeFox, "TypeFox/typir," Jun. 2025, original-date: 2023-03-08T14:54:05Z. [Online]. Available: https://github.com/TypeFox/typir

[11] L. Cardelli, “Type Systems,” in *Computer science handbook*, 2nd ed., A. B. Tucker, Ed. Boca Raton, Fla: Chapman & Hall/CRC, 2004.

[12] B. C. Pierce, *Types and programming languages.* Cambridge, Mass: MIT Press, 2002.

[13] T. Hickey, Q. Ju, and M. H. Van Emden, “Interval arithmetic: From principles to implementation,” *Journal of the ACM*, vol. 48, no. 5, pp. 1038–1068, Sep. 2001. [Online]. Available: https://dl.acm.org/doi/10.1145/502102.502106

Source Code

R-CHECK Type System

```
import { LangiumTypeSystemDefinition, TypirLangiumServices } from "typir-langium";
import { Agent, Assign, AutomatonState, BinExpr, BinObs, Box, Diamond, Enum, ExistsObs, Finally,
  ForallObs, Get, Globally, Guard, GuardCall, isAgent, isAssign, isBinExpr, isBinObs, isBoolLiteral,
  isBox, isBroadcast, isCase, isChannelObs, isChannelRef, isDiamond, isEnum, isExistsObs, isForallObs,
  isGet, isGuard, isInstance, isLiteralObs, isLocal, isLtolMod, isLtolQuant, isMsgStruct, isMyself,
  isNeg, isNumberLiteral, isParam, isPropVar, isRange, isReceive, isRelabel, isSend, isSenderObs,
  isSupply, isUMinus, Local, MsgStruct, Neg, Next, Param, PropVar, QualifiedRef, RCheckAstType,
  Receive, Relabel, Send, Supply, SupplyLocationExpr, UMinus } from "./generated/ast.js";
import { assertUnreachable, AstNode } from "langium";
import { InferOperatorWithMultipleOperands, InferOperatorWithSingleOperand,
  InferenceRuleNotApplicable, NO_PARAMETER_NAME, Type, TypirServices,
  ValidationProblemAcceptor, isClassType } from "typir";
import { getClassDetails, getTypeName, intersectMaps, IntRange, isComparisonOp, validateAssignment } from "./util.js";

export class RCheckTypeSystem implements LangiumTypeSystemDefinition<RCheckAstType> {
  onInitialize(typir: TypirLangiumServices<RCheckAstType>): void {
    // Define the primitive types
    const typeBool = typir.factory.Primitives.create({ primitiveName: "bool" })
      .inferenceRule({ filter: isBoolLiteral })
      .inferenceRule({ filter: isLiteralObs })
      .inferenceRule({ filter: isChannelObs })
      .inferenceRule({ filter: isSenderObs })
      .inferenceRule({
        languageKey: [Local, Param, MsgStruct, PropVar],
        matching: (node: Local | Param | MsgStruct | PropVar) => node.builtinType === "bool",
      })
```

```
    .finish();

  const typeInt = typir.factory.Primitives.create({ primitiveName: "int" })
    .inferenceRule({ filter: isNumberLiteral })
    .inferenceRule({
      languageKey: [Local, Param, MsgStruct, PropVar],
      matching: (node: Local | Param | MsgStruct | PropVar) => node.builtinType === "int",
    })
    .finish();

  const typeRange = typir.factory.Primitives.create({
    primitiveName: "range",
  })
    .inferenceRule({ filter: isRange })
    .inferenceRule({
      languageKey: [Local, Param, MsgStruct, PropVar],
      matching: (node: Local | Param | MsgStruct | PropVar) => node.rangeType !== undefined,
    })
    .finish();

  typir.Conversion.markAsConvertible(typeRange, typeInt, "IMPLICIT_EXPLICIT");

  const typeLocation = typir.factory.Primitives.create({
    primitiveName: "location",
  })
    .inferenceRule({
      languageKey: [Local, Param, MsgStruct, PropVar],
      matching: (node: Local | Param | MsgStruct | PropVar) => node.builtinType === "location",
    })
    .inferenceRule({ filter: isMyself })
    .inferenceRule({
      languageKey: SupplyLocationExpr,
      matching: (node: SupplyLocationExpr) => node.myself !== undefined || node.any !== undefined,
    })
    .inferenceRule({ filter: isInstance })
    .finish();

  const typeChannel = typir.factory.Primitives.create({
    primitiveName: "channel",
  })
    .inferenceRule({ filter: isChannelRef })
    .inferenceRule({ filter: isBroadcast })
    .inferenceRule({
      languageKey: [Local, Param, MsgStruct, PropVar],
      matching: (node: Local | Param | MsgStruct | PropVar) => node.customType?.ref?.name === "channel",
    })
    .inferenceRule({
```

```
      languageKey: Enum,
      matching: (node: Enum) => node.name === "channel",
    })
    .finish();

  const typeAny = typir.factory.Top.create({}).finish();

  // Inference rule for binary operators
  const binaryInferenceRule: InferOperatorWithMultipleOperands<AstNode, BinExpr> = {
    filter: isBinExpr,
    matching: (node: BinExpr, name: string) => node.operator === name,
    operands: (node: BinExpr) => [node.left, node.right],
    validateArgumentsOfCalls: true,
  };

  // Binary operators
  for (const operator of ["+", "-", "*", "/"]) {
    typir.factory.Operators.createBinary({
      name: operator,
      signatures: [
        { left: typeInt, right: typeInt, return: typeInt },
        { left: typeRange, right: typeRange, return: typeRange },
        { left: typeInt, right: typeRange, return: typeRange },
        { left: typeRange, right: typeInt, return: typeRange },
      ],
    })
      .inferenceRule({ ...binaryInferenceRule })
      .finish();
  }
  for (const operator of ["<", "<=", ">", ">="]) {
    typir.factory.Operators.createBinary({
      name: operator,
      signatures: [
        { left: typeInt, right: typeInt, return: typeBool },
        { left: typeRange, right: typeRange, return: typeBool },
        { left: typeInt, right: typeRange, return: typeBool },
        { left: typeRange, right: typeInt, return: typeBool },
      ],
    })
      .inferenceRule(binaryInferenceRule)
      .finish();
  }
  for (const operator of ["&", "|", "->", "U", "R", "W"]) {
    typir.factory.Operators.createBinary({
      name: operator,
      signature: { left: typeBool, right: typeBool, return: typeBool },
    })
      .inferenceRule(binaryInferenceRule)
      .finish();
  }
  // The syntax allows this only for numbers, but the type system allows it for all types
```

```
    for (const operator of ["=", "!=", "=="]) {
      typir.factory.Operators.createBinary({
        name: operator,
        signature: { left: typeAny, right: typeAny, return: typeBool },
      })
        .inferenceRule({
          ...binaryInferenceRule,
          validation: (node, _operatorName, _operatorType, accept, typir) => {
            const leftType = typir.Inference.inferType(node.left);
            const rightType = typir.Inference.inferType(node.right);
            if (
              (leftType === typeRange && rightType === typeInt) ||
              (leftType === typeInt && rightType === typeRange) ||
              (leftType === typeRange && rightType === typeRange)
            ) {
              const leftRange = IntRange.fromRangeExpr(node.left);
              const rightRange = IntRange.fromRangeExpr(node.right);
              if (!leftRange.intersects(rightRange)) {
                accept({
                  message: `This comparison will always return '${
                    node.operator === "!=" ? "true" : "false"
                  }' as the ranges '${leftRange}' and '${rightRange}' have no overlap.`,
                  languageNode: node,
                  languageProperty: "operator",
                  severity: "warning",
                });
              }
            } else {
              typir.validation.Constraints.ensureNodeIsEquals(node.left, node.right, accept, (actual, expected) => ({
                message: `This comparison will always return '${node.operator === "!=" ? "true" : "false"}' as '${
                  node.left.$cstNode?.text
                }' and '${node.right.$cstNode?.text}' have the different types '${getTypeName(
                  actual
                )}' and '${getTypeName(expected)}'.`,
                languageNode: node,
                languageProperty: "operator",
                severity: "warning",
              }));
            }
          },
        })
        .finish();
    }
    typir.factory.Operators.createBinary({
      name: "<-",
      signature: { left: typeAny, right: typeAny, return: typeAny },
    })
      .inferenceRule({
```

```
        filter: isRelabel,
        matching: () => true,
        operands: (node: Relabel) => [node.var.ref!, node.expr],
        validation: (node, _operator, _functionType, accept, typir) =>
          validateAssignment(node, getTypeName, accept, typir),
        validateArgumentsOfCalls: true,
      })
      .finish();
    typir.factory.Operators.createBinary({
      name: ":=",
      signature: { left: typeAny, right: typeAny, return: typeAny },
    })
      .inferenceRule({
        filter: isAssign,
        matching: () => true,
        operands: (node: Assign) => [node.left.ref!, node.right],
        validation: (node, _operator, _functionType, accept, typir) =>
          validateAssignment(node, getTypeName, accept, typir),
        validateArgumentsOfCalls: true,
      })
      .finish();
    for (const operator of ["&", "|", "->", "<->"]) {
      typir.factory.Operators.createBinary({
        name: operator,
        signature: { left: typeBool, right: typeBool, return: typeBool },
      })
        .inferenceRule({
          filter: isBinObs,
          matching: (node: BinObs, name: string) => node.operator === name,
          operands: (node: BinObs) => [node.left, node.right],
          validateArgumentsOfCalls: true,
        })
        .finish();
    }
    for (const operator of ["Diamond", "Box"]) {
      typir.factory.Operators.createBinary({
        name: operator,
        signature: { left: typeBool, right: typeBool, return: typeBool },
      })
        .inferenceRule({
          filter: isDiamond,
          matching: (_node: Diamond, name: string) => name === "Diamond",
          operands: (node: Diamond) => [node.obs, node.expr],
          validateArgumentsOfCalls: true,
        })
        .inferenceRule({
          filter: isBox,
          matching: (_node: Box, name: string) => name === "Box",
          operands: (node: Box) => [node.obs, node.expr],
          validateArgumentsOfCalls: true,
        })
        .finish();
    }
```

```

  // Inference rule for unary operators
  type UnaryExpression = UMinus | Neg | Finally | Globally | Next |
  ForallObs | ExistsObs;
  const isUnaryExpression = (node: AstNode): node is UnaryExpression => {
    return isUMinus(node) || isNeg(node) || isLtolMod(node) || isForallObs(
  node) || isExistsObs(node);
  };
  const unaryInferenceRule: InferOperatorWithSingleOperand<AstNode,
  UnaryExpression> = {
    filter: isUnaryExpression,
    matching: (node: UnaryExpression, name: string) => node.operator ===
  name,
    operand: (node: UnaryExpression) => node.expr,
    validateArgumentsOfCalls: true,
  };

  // Unary operators
  typir.factory.Operators.createUnary({
    name: "-",
    signatures: [
      { operand: typeInt, return: typeInt },
      { operand: typeRange, return: typeRange },
    ],
  })
    .inferenceRule(unaryInferenceRule)
    .finish();
  for (const operator of ["!", "F", "G", "X", "forall", "exists"]) {
    typir.factory.Operators.createUnary({
      name: operator,
      signature: { operand: typeBool, return: typeBool },
    })
      .inferenceRule(unaryInferenceRule)
      .finish();
  }

  // Handle variable references
  typir.Inference.addInferenceRulesForAstNodes({
    Ref: (languageNode) => {
      const ref = languageNode.variable.ref;
      if (isLocal(ref)) {
        return ref;
      } else if (isCase(ref)) {
        return ref.$container;
      } else if (isParam(ref)) {
        return ref;
      } else if (isMsgStruct(ref)) {
        return ref;
      } else if (isPropVar(ref)) {
        return ref;
      } else if (isSend(ref)) {
        return InferenceRuleNotApplicable;
      } else if (isReceive(ref)) {
```

```
          return InferenceRuleNotApplicable;
        } else if (isGet(ref)) {
          return InferenceRuleNotApplicable;
        } else if (isSupply(ref)) {
          return InferenceRuleNotApplicable;
        } else if (isInstance(ref)) {
          return ref;
        } else if (ref === undefined) {
          return InferenceRuleNotApplicable;
        } else {
          assertUnreachable(ref);
        }
      },
      PropVarRef: (languageNode) => {
        const ref = languageNode.variable.ref;
        if (isPropVar(ref)) {
          return ref;
        } else {
          return InferenceRuleNotApplicable;
        }
      },
      QualifiedRef: (languageNode) => {
        const instance = languageNode.instance.ref;
        if (isInstance(instance)) {
          // Case already handled in class declaration
          return InferenceRuleNotApplicable;
        } else if (isLtolQuant(instance)) {
          if (instance.kinds.some((k) => k.ref === undefined)) {
            throw new Error("Not a valid agent instance.");
          }

          const agentFields = instance.kinds.map((k) => {
            const agentType = typir.Inference.inferType(k.ref!);

            if (agentType instanceof Type) {
              if (isClassType(agentType)) {
                return agentType.getFields(false);
              } else {
                throw new Error("Encountered unexpected non-class type.");
              }
            } else if (agentType instanceof Array) {
              throw new Error("Encountered duplicate class type.");
            } else {
              assertUnreachable(agentType);
            }
          });

          const intersection = intersectMaps(agentFields);
          const variableType = intersection.get(languageNode.variable.$refText);

          if (variableType === undefined) {
            // Field does not exist on agent intersection
```

```
          typir.validation.Collector.addValidationRule((node, accept) => {
            if (node === languageNode) {
              accept({
                languageNode: node,
                languageProperty: "variable",
                severity: "error",
                message: `Property '${languageNode.variable.$refText}' does
 not exist on type '${instance.kinds
                  .map((k) => k.ref?.name)
                  .join(" | ")}'.`,
              });
            }
          });
          return typeAny;
        } else {
          return variableType;
        }
      } else if (instance === undefined) {
        return InferenceRuleNotApplicable;
      } else {
        assertUnreachable(instance);
      }
    },
    ChannelExpr: (languageNode) => {
      if (languageNode.bcast !== undefined) {
        return typeChannel;
      } else if (languageNode.channel?.ref !== undefined) {
        return languageNode.channel.ref;
      } else {
        return InferenceRuleNotApplicable;
      }
    },
    GetLocationExpr: (languageNode) => languageNode.predicate,
    SupplyLocationExpr: (languageNode) => {
      const location = languageNode.location?.ref;
      if (location !== undefined) {
        return location;
      } else {
        return InferenceRuleNotApplicable;
      }
    },
  });

  const validateCmdHeader = (
    node: Send | Receive | Get | Supply,
    accept: ValidationProblemAcceptor<AstNode>,
    typir: TypirServices<AstNode>
  ) => {
    typir.validation.Constraints.ensureNodeIsEquals(node.psi, typeBool,
  accept, (actual, expected) => ({
      message: `Type mismatch in command guard expression: expected '${
  getTypeName(
        expected
```

```
      )}', but got '${getTypeName(actual)}'.`,
      languageProperty: "psi",
      languageNode: node,
    }));
  };
  const validateChannelExpr = (
    node: Send | Receive,
    accept: ValidationProblemAcceptor<AstNode>,
    typir: TypirServices<AstNode>
  ) => {
    typir.validation.Constraints.ensureNodeIsEquals(node.chanExpr, typeChannel, accept, (actual, expected) => ({
      message: `Type mismatch in command channel expression: expected '${getTypeName(
        expected
      )}', but got '${getTypeName(actual)}'.`,
      languageProperty: "chanExpr",
      languageNode: node,
    }));
  };
  const validateSupplyLocation = (
    node: Supply,
    accept: ValidationProblemAcceptor<AstNode>,
    typir: TypirServices<AstNode>
  ) => {
    typir.validation.Constraints.ensureNodeIsEquals(node.where, typeLocation, accept, (actual, expected) => ({
      message: `Type mismatch in command where: expected '${getTypeName(expected)}', but got '${getTypeName(
        actual
      )}'.`,
      languageProperty: "where",
      languageNode: node,
    }));
  };
  const validateGetLocation = (
    node: Get,
    accept: ValidationProblemAcceptor<AstNode>,
    typir: TypirServices<AstNode>
  ) => {
    const actual = typir.Inference.inferType(node.where);
    if (actual instanceof Type && actual.getIdentifier() !== "bool" && actual.getIdentifier() !== "location") {
      accept({
        message: `Type mismatch in command where: expected 'bool | location', but got '${
          actual instanceof Type ? typir.Printer.printTypeName(actual) : "inference problem"
        }'.`,
        languageProperty: "where",
        languageNode: node,
        severity: "error",
      });
```

```
    }
  };

  typir.validation.Collector.addValidationRulesForAstNodes({
    Ltol: (node, accept, typir) => {
      typir.validation.Constraints.ensureNodeIsEquals(node.expr, typeBool,
accept, () => ({
        message: "SPEC needs to evaluate to 'bool'.",
        languageProperty: "expr",
        languageNode: node,
      }));
    },
    ChannelObs: (node, accept, typir) => {
      // Do not need to check broadcast symbol
      if (node.bcast !== undefined) return;
      typir.validation.Constraints.ensureNodeIsEquals(node.chan?.ref?.
$container, typeChannel, accept, () => ({
        message: "Channel reference needs to evaluate to 'channel'.",
        languageProperty: "chan",
        languageNode: node,
      }));
    },
    Send: (node, accept, typir) => {
      validateCmdHeader(node, accept, typir);
      validateChannelExpr(node, accept, typir);
      typir.validation.Constraints.ensureNodeIsEquals(node.sendGuard,
typeBool, accept, (actual, expected) => ({
        message: `Type mismatch in command guard: expected '${getTypeName(
expected)}', but got '${getTypeName(
          actual
        )}'.`,
        languageProperty: "sendGuard",
        languageNode: node,
      }));
    },
    Receive: (node, accept, typir) => {
      validateCmdHeader(node, accept, typir);
      validateChannelExpr(node, accept, typir);
    },
    Get: (node, accept, typir) => {
      validateCmdHeader(node, accept, typir);
      validateGetLocation(node, accept, typir);
    },
    Supply: (node, accept, typir) => {
      validateCmdHeader(node, accept, typir);
      validateSupplyLocation(node, accept, typir);
    },
    Guard: (node, accept, typir) => {
      typir.validation.Constraints.ensureNodeIsEquals(node.body, typeBool,
accept, (actual, expected) => ({
        message: `Type mismatch in guard definition: expected '${
getTypeName(expected)}', but got '${getTypeName(
          actual
```

```
        )}'.`,
        languageProperty: "body",
        languageNode: node.body,
      }));
    },
    Agent: (node, accept, typir) => {
      typir.validation.Constraints.ensureNodeIsEquals(node.init, typeBool, accept, (actual, expected) => ({
        message: `Type mismatch in agent initialization: expected '${ getTypeName(expected)}', but got '${getTypeName(
          actual
        )}'.`,
        languageProperty: "init",
        languageNode: node.init,
      }));
      typir.validation.Constraints.ensureNodeIsEquals(node.recvguard, typeBool, accept, (actual, expected) => ({
        message: `Type mismatch in agent receive-guard: expected '${ getTypeName(expected)}', but got '${getTypeName(
          actual
        )}'.`,
        languageProperty: "recvguard",
        languageNode: node.recvguard,
      }));
    },
    Instance: (node, accept, typir) =>
      typir.validation.Constraints.ensureNodeIsEquals(node.init, typeBool, accept, (actual, expected) => ({
        message: `Type mismatch in instance initialization: expected '${ getTypeName(
          expected
        )}', but got '${getTypeName(actual)}'.`,
        languageProperty: "init",
        languageNode: node.init,
      })),
    CompoundExpr: (node, accept, typir) => {
      if (node.$type !== "BinExpr" || !isComparisonOp(node.operator)) {
        return;
      }
      const leftType = typir.Inference.inferType(node.left);
      const rightType = typir.Inference.inferType(node.right);
      if ((leftType === typeRange || leftType === typeInt) && (rightType === typeInt || rightType === typeRange)) {
        const leftRange = IntRange.fromRangeExpr(node.left);
        const rightRange = IntRange.fromRangeExpr(node.right);
        const { isAlwaysTrue, isAlwaysFalse } = IntRange.isStaticOutcome( leftRange, rightRange, node.operator);
        if (!isAlwaysTrue && !isAlwaysFalse) {
          return;
        }
        let reason;
        switch (node.operator) {
          case "<":
```

```
            reason = isAlwaysTrue
              ? `every value of '${leftRange}' is strictly less than every value of '${rightRange}'`
              : `every value of '${leftRange}' is greater than or equal to every value of '${rightRange}'`;
            break;
          case "<=":
            reason = isAlwaysTrue
              ? `the max of '${leftRange}' is less than or equal to the min of '${rightRange}'`
              : `the min of '${leftRange}' is greater than the max of '${rightRange}'`;
            break;
          case ">":
            reason = isAlwaysTrue
              ? `every value of '${leftRange}' is strictly greater than every value of '${rightRange}'`
              : `every value of '${leftRange}' is less than or equal to every value of '${rightRange}'`;
            break;
          case ">=":
            reason = isAlwaysTrue
              ? `the min of '${leftRange}' is greater than or equal to the max of '${rightRange}'`
              : `the max of '${leftRange}' is less than the min of '${rightRange}'`;
            break;
        }

        accept({
          message: `This comparison will always return '${isAlwaysTrue ? "true" : "false"}' as ${reason}.`,
          languageNode: node,
          languageProperty: "operator",
          severity: "warning",
        });
      }
    },
  });
}

onNewAstNode(languageNode: AstNode, typir: TypirLangiumServices<RCheckAstType>): void {
  if (isEnum(languageNode)) {
    // Exclude channel enum here
    if (languageNode.name === "channel") return;

    // The container of Enum node is always the root node
    const documentUri = languageNode.$container.$document!.uri;
    const enumName = `${documentUri}::${languageNode.name}`;

    // Skip type definition in case of duplicates
    if (typir.factory.Primitives.get({ primitiveName: enumName }) !==
```

```
undefined) return;

      // Create new enum type
      typir.factory.Primitives.create({ primitiveName: enumName })
        .inferenceRule({
          languageKey: [Local, Param, MsgStruct, PropVar],
          matching: (node: Local | Param | MsgStruct | PropVar) =>
    languageNode === node.customType?.ref,
        })
        .inferenceRule({
          languageKey: Enum,
          matching: (node: Enum) => languageNode === node,
        })
        .finish();
    }

    if (isGuard(languageNode)) {
      typir.factory.Functions.create({
        functionName: languageNode.name,
        outputParameter: { name: NO_PARAMETER_NAME, type: "bool" },
        inputParameters: languageNode.params.map((p) => ({
          name: p.name,
          type: p,
        })),
        associatedLanguageNode: languageNode,
      })
        .inferenceRuleForDeclaration({
          languageKey: Guard,
          matching: (node: Guard) => languageNode === node,
        })
        .inferenceRuleForCalls({
          languageKey: GuardCall,
          matching: (node: GuardCall) => languageNode === node.guard.ref,
          inputArguments: (node: GuardCall) => node.args,
          validateArgumentsOfFunctionCalls: true,
        })
        .finish();
    }

    if (isAgent(languageNode)) {
      // Skip class definition in case of duplicates
      if (languageNode.name === undefined || typir.factory.Classes.get(
    languageNode.name).getType() !== undefined) {
        return;
      }

      typir.factory.Classes.create(getClassDetails(languageNode))
        .inferenceRuleForClassDeclaration({
          languageKey: Agent,
          matching: (node: Agent) => languageNode === node,
        })
        .inferenceRuleForFieldAccess({
          languageKey: QualifiedRef,
```

```
          matching: (node: QualifiedRef) => {
            const qualifier = node.instance.ref;
            // Handle LtolQuant inference separately
            if (isLtolQuant(qualifier)) return false;

            return qualifier?.agent.ref === languageNode;
          },
          field: (node: QualifiedRef) => node.variable.ref!.name!,
        })
        .inferenceRuleForFieldAccess({
          languageKey: AutomatonState,
          matching: (node: AutomatonState) => languageNode === node.instance.ref?.agent.ref,
          field: () => "automaton-state",
        })
        .finish();
    }
  }
}
```

Listing 1: Full implementation of the R-CHECK type system.

Utility Functions

```
import { NodeFileSystem } from "langium/node";
import { extractAstNode } from "../cli/cli-util.js";
import { Agent, Assign, BaseProcess, BinExpr, CompoundExpr, isBinExpr, isChoice, isGet, isLocal, isMsgStruct, isNumberLiteral, isParam, isPropVar, isPropVarRef, isQualifiedRef, isReceive, isRef, isRelabel, isRep, isSend, isSequence, isSupply, isTarget, isUMinus, Local, Model, PropVar, Relabel, Sequence, Target } from "./generated/ast.js";
import { createRCheckServices } from "./r-check-module.js";
import { AstNode } from "langium";
import { ClassTypeDetails, AnnotatedTypeAfterValidation, ValidationProblemAcceptor, TypirServices } from "typir";

export async function parseToJson(fileName: string) {
    const services = createRCheckServices(NodeFileSystem).RCheck;
    const model = await extractAstNode<Model>(fileName, services);
    return JSON.stringify(model, getAstReplacer());
}

export function parseToJsonSync(fileName: string) {
    let result = "";
    (async () => await parseToJson(fileName).then((x) => result = x))();
    return result;
}

const getAstReplacer = () => {
    /**
     * Used with JSON.stringify() to make a JSON of a Langium AST.
     */
```

```
    // Extra measure to remove circular references. See
    // https://stackoverflow.com/a/53731154
    const seen = new WeakSet();
    return (key: any, value: any) => {
        // Remove Langium nodes that we won't need
        if (
            key === "references" || key === "$cstNode" || key === "$refNode" ||
            key === "_ref" || key === "ref"
            ||
            key === "$nodeDescription" || key === "_nodeDescription") {
            return;
        }
        // Remove seen nodes
        if (typeof value === "object" && value !== null) {
            if (seen.has(value)) {
                return;
            }
            seen.add(value);
        }
        return value;
    };
};

type ComparisonOp = "<" | "<=" | ">" | ">=";

export class IntRange {
  private lower: number;
  private upper: number;

  constructor(lower: number, upper: number) {
    this.lower = lower;
    this.upper = upper;
  }

  public static fromRangeExpr(expr: CompoundExpr | PropVar | Target): IntRange {
    if (isRef(expr) || isPropVar(expr) || isPropVarRef(expr) || isTarget(expr) || isQualifiedRef(expr)) {
      const decl = isPropVar(expr) || isTarget(expr) ? expr : expr.variable.ref;
      if (isLocal(decl) || isParam(decl) || isMsgStruct(decl) || isPropVar(decl)) {
        if (decl.rangeType !== undefined) {
          return new this(decl.rangeType.lower, decl.rangeType.upper);
        } else if (decl.builtinType === "int") {
          return new this(Number.NEGATIVE_INFINITY, Number.POSITIVE_INFINITY);
        } else {
          throw new Error(
            `Encountered declaration with unexpected type: ${decl.builtinType ?? decl.customType?.ref?.name}.`
```

```
          );
        }
      } else {
        throw new Error("Unexpected target found.");
      }
    } else if (isNumberLiteral(expr)) {
      return new this(expr.value, expr.value);
    } else if (isBinExpr(expr)) {
      const leftRange = IntRange.fromRangeExpr(expr.left);
      const rightRange = IntRange.fromRangeExpr(expr.right);
      switch (expr.operator) {
        case "+":
          return leftRange.plus(rightRange);
        case "-":
          return leftRange.minus(rightRange);
        case "*":
          return leftRange.times(rightRange);
        case "/":
          return leftRange.dividedBy(rightRange);
        default:
          throw new Error("Unexpected operator found.");
      }
    } else if (isUMinus(expr)) {
      return new this(0, 0).minus(IntRange.fromRangeExpr(expr.expr));
    } else {
      throw new Error(`Unexpected expression found: '${expr.$type}'.`);
    }
  }

  public static isStaticOutcome(
    leftRange: IntRange,
    rightRange: IntRange,
    operator: ComparisonOp
  ): { isAlwaysTrue: boolean; isAlwaysFalse: boolean } {
    switch (operator) {
      case "<":
        return {
          isAlwaysTrue: leftRange.upper < rightRange.lower,
          isAlwaysFalse: leftRange.lower >= rightRange.upper,
        };
      case "<=":
        return {
          isAlwaysTrue: leftRange.upper <= rightRange.lower,
          isAlwaysFalse: leftRange.lower > rightRange.upper,
        };
      case ">":
        return {
          isAlwaysTrue: leftRange.lower > rightRange.upper,
          isAlwaysFalse: leftRange.upper <= rightRange.lower,
        };
      case ">=":
        return {
          isAlwaysTrue: leftRange.lower >= rightRange.upper,
```

```
          isAlwaysFalse: leftRange.upper < rightRange.lower,
        };
    }
  }

  public plus(other: IntRange): IntRange {
    return new IntRange(this.lower + other.lower, this.upper + other.upper);
  }

  public minus(other: IntRange): IntRange {
    return new IntRange(this.lower - other.upper, this.upper - other.lower);
  }

  public times(other: IntRange): IntRange {
    const p1 = this.lower * other.lower;
    const p2 = this.lower * other.upper;
    const p3 = this.upper * other.lower;
    const p4 = this.upper * other.upper;
    return new IntRange(Math.min(p1, p2, p3, p4), Math.max(p1, p2, p3, p4));
  }

  public dividedBy(other: IntRange): IntRange {
    if (other.lower === 0 || other.upper === 0) {
      throw new Error("Division by a range that includes zero is not supported.");
    }
    const d1 = Math.trunc(this.lower / other.lower);
    const d2 = Math.trunc(this.lower / other.upper);
    const d3 = Math.trunc(this.upper / other.lower);
    const d4 = Math.trunc(this.upper / other.upper);

    return new IntRange(Math.min(d1, d2, d3, d4), Math.max(d1, d2, d3, d4));
  }

  public intersects(other: IntRange): boolean {
    return this.lower <= other.upper && other.lower <= this.upper;
  }

  public contains(other: IntRange): boolean {
    return this.lower <= other.lower && this.upper >= other.upper;
  }

  public toString(): string {
    if (isFinite(this.lower) && isFinite(this.upper)) {
      return this.lower === this.upper ? `${this.lower}` : `${this.lower}..${this.upper}`;
    }
    return "int";
  }
}

export const isComparisonOp = (o: BinExpr["operator"]): o is ComparisonOp => {
```

```
  return o === "<" || o === "<=" || o === ">" || o === ">=";
};

export const getClassDetails = (agent: Agent): ClassTypeDetails<AstNode> => {
  const fieldNames = new Set<string>(["automaton-state"]);

  const locals = agent.locals
    .map((l) => {
      if (fieldNames.has(l.name)) {
        return undefined;
      }
      fieldNames.add(l.name);
      return { name: l.name, type: l };
    })
    .filter((l): l is { name: string; type: Local } => l !== undefined);

  const processes = getProcessNames(agent)
    .map((n) => {
      if (fieldNames.has(n)) {
        return undefined;
      }
      fieldNames.add(n);
      return { name: n, type: "bool" };
    })
    .filter((p): p is { name: string; type: string } => p !== undefined);

  return {
    className: agent.name,
    fields: [{ name: "automaton-state", type: "int" }, ...processes, ...
    locals],
    methods: [],
  };
};

export const getProcessNames = (agent: Agent): string[] => {
  const stack: (BaseProcess | Sequence)[] = [agent.repeat];
  const processNames: string[] = [];

  while (stack.length !== 0) {
    const process = stack.pop();
    if (isSend(process) || isReceive(process) || isGet(process) || isSupply(
    process)) {
      if (process.name) {
        processNames.push(process.name);
      }
    }
    if (isChoice(process) || isSequence(process)) {
      stack.push(process.left);
      if (process.right !== undefined) {
        stack.push(process.right);
      }
    }
    if (isRep(process)) {
```

```
      stack.push(process.process);
    }
  }

  return processNames;
};

export const getTypeName = (type: AnnotatedTypeAfterValidation): string |
    undefined => {
  return type.name.split("::").pop();
};

export const intersectMaps = <K, V>(maps: Map<K, V>[]): Map<K, V> => {
  if (maps.length === 0) {
    return new Map<K, V>();
  }
  if (maps.length === 1) {
    return new Map(maps[0]);
  }

  const resultMap = new Map<K, V>();
  const firstMap = maps[0];

  // Iterate over the entries of the first map
  for (const [key, value] of firstMap.entries()) {
    let isInAllMaps = true;

    // Check if this key exists in all other maps with the same value
    for (let i = 1; i < maps.length; i++) {
      const currentMap = maps[i];
      if (!currentMap.has(key) || currentMap.get(key) !== value) {
        isInAllMaps = false;
        break;
      }
    }
    // If the key and value matched across all maps, add it to the result
    if (isInAllMaps) {
      resultMap.set(key, value);
    }
  }

  return resultMap;
};

export const validateAssignment = (
  node: Relabel | Assign,
  getTypeName: (type: AnnotatedTypeAfterValidation) => string | undefined,
  accept: ValidationProblemAcceptor<AstNode>,
  typir: TypirServices<AstNode>
) => {
  const targetNode = isRelabel(node) ? node.var.ref! : node.left.ref!;
  const exprNode = isRelabel(node) ? node.expr : node.right;
  const property = isRelabel(node) ? "var" : "left";
```

```
  const typeInt = typir.factory.Primitives.get({ primitiveName: "int" });
  const typeRange = typir.factory.Primitives.get({ primitiveName: "range" });

  const targetType = typir.Inference.inferType(targetNode);
  const exprType = typir.Inference.inferType(exprNode);

  if ((targetType === typeRange && exprType === typeInt) || (targetType === typeRange && exprType === typeRange)) {
    const targetRange = IntRange.fromRangeExpr(targetNode);
    const exprRange = IntRange.fromRangeExpr(exprNode);

    if (!targetRange.contains(exprRange)) {
      accept({
        message: `Range variable cannot be ${
          property === "var" ? "relabelled" : "assigned"
        } as the range '${targetRange}' does not contain the range of the expression '${exprRange}'.`,
        languageNode: node,
        languageProperty: property,
        severity: "error",
      });
    }
  } else {
    typir.validation.Constraints.ensureNodeIsAssignable(exprNode, targetNode, accept, (actual, expected) => ({
      message: `${property === "var" ? "Variable" : "Expression"} of type '${getTypeName(
        property === "var" ? expected : actual
      )}' cannot be ${
        property === "var" ? "relabelled with expression of type" : "assigned to variable of type"
      } '${getTypeName(property === "var" ? actual : expected)}'.`,
      languageNode: node,
      languageProperty: property,
      severity: "error",
    }));
  }
};
```
Listing 2: Full implementation of used utility functions.

Grammar Changes

```
diff --git a/src/language/r-check.langium b/src/language/r-check.
    langium
index 952f2f6..58f4ed1 100644
--- a/src/language/r-check.langium
+++ b/src/language/r-check.langium
@@ -35,7 +35,7 @@ Agent:
     'repeat' ':' repeat=Choice
     ;

-Relabel: var=[PropVar] '<-' CompoundExpr;
+Relabel: var=[PropVar] '<-' expr=CompoundExpr;

 Choice:
     left=Sequence ({infer Choice.left=current} '+' right=Sequence)*;
@@ -69,10 +69,9 @@ Assign: left=[Target] ':=' right=CompoundExpr;

 ChannelExprRef: Case | Local;
 ChannelExpr: (channel=[ChannelExprRef] | bcast = '*');
-LocationExprRef: Instance | Local;
-LocationExpr: (location=[LocationExprRef]);
+LocationExprRef: Local;
 SupplyLocationExpr: (location=[LocationExprRef] | myself="myself" |
    any="any");
-GetLocationExpr: (location=[LocationExprRef] | predicate=
    CompoundExpr);
+GetLocationExpr: predicate=CompoundExpr;


 fragment TypedDeclaration:
@@ -91,30 +90,32 @@ Param: TypedDeclaration;
 MsgStruct: TypedDeclaration;
 PropVar: TypedDeclaration;

-
 CompoundExpr:
-    left=Comparison ({infer CompoundExpr.left=current} operator
    =('&'|'|'|'->'|'U'|'R'|'W') right=Comparison)*;
```

```
+    AddSub;
+
+Logical infers CompoundExpr:
+    Comparison ({infer BinExpr.left=current} operator=('&'|'|'|'->'|'U'|'R'|'W') right=Comparison)*;

-Comparison:
-    left=AddSub ({infer Comparison.left=current} operator=('<'|'<='|'>'|'>='|'='|'!='|'==') right=AddSub)?;
+Comparison infers CompoundExpr:
+    BaseExpr ({infer BinExpr.left=current} operator=('<'|'<='|'>'|'>='|'='|'!='|'==') right=BaseExpr)?;

-AddSub:
+AddSub infers CompoundExpr:
     MulDiv ({infer BinExpr.left=current} operator=('+' | '-') right=MulDiv)*;

-MulDiv:
-    BaseExpr ({infer BinExpr.left=current} operator=('*' | '/') right=BaseExpr)*;
+MulDiv infers CompoundExpr:
+    Logical ({infer BinExpr.left=current} operator=('*' | '/') right=Logical)*;

 Qualifier : Instance | LtolQuant;

-BaseExpr:
-    '(' CompoundExpr ')'
+BaseExpr infers CompoundExpr:
+    '(' AddSub ')'
     | {infer AutomatonState} instance=[Instance] '-automaton-state'
     | {infer QualifiedRef} instance=[Qualifier] '-' variable=[Target]
     | {infer Ref} variable=[Target]
     | {infer PropVarRef} variable=[PropVar:PV]
-    | {infer UMinus} '-' expr=BaseExpr
-    | {infer Neg} '!' expr=BaseExpr
-    | {infer Ref} currentChannel='chan'
+    | {infer UMinus} operator='-' expr=BaseExpr
+    | {infer Neg} operator='!' expr=BaseExpr
+    | {infer ChannelRef} currentChannel='chan'
     | {infer Myself} myself='myself'
     | {infer Broadcast} value="*"
     | {infer NumberLiteral} value=INT
@@ -124,16 +125,14 @@ BaseExpr:
     | {infer LtolBase} LtolBase
     ;

```

```
-type Expr = BaseExpr | BinExpr | CompoundExpr | Comparison ;
-
 Ltol: (quants+=LtolQuant)* expr=CompoundExpr;

 LtolQuant: op=('forall'|'exists') name=ID ':' (anyKind='Agent' | kinds+=[Agent] ('|' kinds+=[Agent])*) '.';

 LtolMod infers Ltol:
-     {infer Finally} 'F' expr=CompoundExpr
-     | {infer Globally} 'G' expr=CompoundExpr
-     | {infer Next} 'X' expr=CompoundExpr
+     {infer Finally} operator='F' expr=CompoundExpr
+     | {infer Globally} operator='G' expr=CompoundExpr
+     | {infer Next} operator='X' expr=CompoundExpr
      ;

 LtolBase infers Ltol:
@@ -147,10 +146,10 @@ CompoundObs infers Obs:

 BaseObs : LiteralObs | ChannelObs | SenderObs | ForallObs | ExistsObs;
 LiteralObs: value=('true'|'false');
-ChannelObs: 'chan' ('=='|'='|'!=') (chan=ID | chan='*');
-SenderObs: 'sender' ('=='|'='|'!=') sender=ID;
-ForallObs: 'forall' '(' (pred=CompoundExpr) ')';
-ExistsObs: 'exists' '(' (pred=CompoundExpr) ')';
+ChannelObs: 'chan' ('=='|'='|'!=') (chan=[Case] | bcast='*');
+SenderObs: 'sender' ('=='|'='|'!=') sender=[Instance];
+ForallObs: operator='forall' '(' (expr=CompoundExpr) ')';
+ExistsObs: operator='exists' '(' (expr=CompoundExpr) ')';


 hidden terminal WS: /\s+/;
```

Listing 3: Full listing of the grammar changes.