Formal Model Definition of Web Service Behavior from Source Code in Rewrite Logic

Olzhas Zhangeldinov

Developers of web applications strive for implementing state-of-the-art design patterns. One of them is microservice architecture design, which increases the number of web services employed in the applications' back end. Formal verification may help to verify the safe and proper interaction between concurrent web services. Most of the current tools focus on verification of existing formal models defined using specification languages such as BPEN (Business Process Execution Language), WS-CDL (Web Service Choreography Description Language), and recently, Conductor. We propose a framework for building formal models of web service architectures using an imperative programming language called WAFL - Web Architecture Formal Language. We also provide a way to define temporal logic properties based on assertions defined using WAFL. The implementation of the framework was realized using the Maude rewrite logic language as an extension to the language itself. The advantage of such a framework is that it provides a way for software developers to model web service architectures without knowledge of formal modelling languages and with little understanding of formal verification.

Acknowledgments

I would not be able to write this thesis without the help of my professors. My supervisor, Professor Antonio Cerone, has contributed an invaluable effort to familiarizing me with rewrite logic and its real-world applications. He provided me with extensive feedback on my work. His knowledge and experience became my greatest support throughout the whole process of writing the thesis. He helped me write the methodology and corrected stylistic mistakes I had made because of my inexperience. He also introduced me to the general format for writing methodology that uses rewrite logic.

I would also like to thank Professor Ben Tyler for providing important background papers in formal verification that guided my initial steps for this work. Professors Latif Zohalib and Michael Lewis taught me to compile related works and structure the bibliography. Professor Hans de Nivelle supported my interest in rewrite logic and suggested using the Maude language for this work. I am grateful to my examiners, Professor Akhan Almagambetov and Professor Ben Tyler, for giving valuable feedback and suggesting extending the Results chapter with new examples I have added after the thesis defence.

When I struggled with the Maude implementation of the framework, Professor Peter Csaba Ölveczky shared his ideas on building Maude language extensions and using the loop mode. Thus, he helped me develop a more user-friendly interface for the tool proposed in this work.

And special thanks to my coursemates and friends Adil, Rakhat, and Azat, who motivated me and shared their perspectives about this thesis.

Introduction

Motivation

We encounter many web applications in our everyday life while using online banking, shopping, or social media applications on our mobile phones and personal computers. Web application, in general, is a software that connects users with the business . Although companies developing web applications develop an intuitive user interface for a general user, most of the heavy lifting is not performed on the devices we access the application. In fact, large companies use powerful hardware on the back-end to run a web service (or web server) that handles requests of users to make a money transaction, order a product, or load the latest posts from other users. The market of web applications is highly competitive, and therefore, most companies try to involve bleeding-edge techniques when implementing the software. One of the most common software design solutions is microservice architecture, which is dividing the back-end of applications into many microservices responsible for a limited number of tasks . Microservice is an isolated software responsible for a small number of tasks that be easily used in different business contexts .

Parallel execution of tasks on different microservices comes with a cost of problems caused by concurrency. Maintaining concurrent web services is much harder than only one web service in some occasions. highlights main challenges in adoption of microservice architectures.

One of them is the untraditional way of debugging services. When the application does not behave the way it is supposed to behave, one can not find the source of the failure by analysing the web service program sequentially. Instead, software engineers model communications between web services to find potential points of failure and eliminate all of them . Developers use state diagrams to visualize the web service behaviour and make concurrency problems apparent. The main drawback of this approach is that it requires experience in building such diagrams and finding potential bugs.

Another problem to overcome is that microservices might store outdated data about entities modified by another service . Some web services might need to operate on the same data while containing each own copy of the data in separate databases. In this case, software engineers develop an algorithm that synchronises the databases to preserve data integrity. However, failure to develop a robust synchronisation algorithm leads to an unpredicted behaviour.

The mitigation of such problems evolved over the years and today large companies use orchestration engines - software that manages execution of web services (WS) using a provided blueprint, or a workflow. Such software as Temporal , Cadence , and Conductor are used by companies to build high-level blueprints of their web service infrastructure. This makes the system easier to understand and allows the development of more reliable workflows. However, making a correct workflow is still the responsibility of a developer, who can overlook possible bugs caused by concurrent execution.

One of the approaches for finding concurrency-related bugs is formal verification. The process of formal verification includes building an abstract formal model of a system and running the model through a suitable verification tool. This reveals all possible cases in which the system might fail . However, the need to create formal models from scratch implies extra costs without clear profit for a company. Formal models should be written and maintained by a specialized person with knowledge of formal methods.

Recent studies focus on the manual construction of Petri-Nets or Event-B based formal models of web services. This approach, however, is too costly for companies to employ, considering the need for more human resources to develop and maintain such formal models. Another approach is automatic conversion from the Conductor orchestration language into Petri Nets. However other research claims that Conductor has a steep learning curve for developers of web services. Other orchestration engines, such as Temporal and Cadence, are preferred because they use imperative programming languages to define workflows.

Therefore, there is a disconnect between current research on formal verification of web services and recent trends in microservice architecture development. This is especially the case in the field of verifying orchestration engines that define their workflows using an imperative programming language. Formal verification of web architectures might be more attractive to software engineers if they were provided with more familiar tools to verify their systems. Previous research in this direction described web services using source code but was more focused on a Client-Server interaction and ignored communication between web services. The study used Maude rewriting logic, and it proved to be capable of modelling imperative programming languages.

Our work proposes WAFL - Web Architecture Formal Language - a framework and language for formal modelling web services using the source code of the services and their orchestration engine. The framework constructs a formal model of web services from a user-friendly syntax that developers can write by directly mapping their source code. Maude rewriting logic language was chosen for developing the framework due to already proven capabilities in such tasks . The framework is aimed to provide a tool for the formal verification of web architectures that utilizes familiar techniques for software engineers.

This thesis provides the conducted literature review of related works in Chapter 2. Then, the methodology of how the framework works is provided in Chapter 3. It demonstrates the application of the framework in a real-world example of electronic document exchange in Chapter 4. In Chapter 5, the thesis concludes the work done and describes the current limitations of the framework and possible improvements.

Related works

This chapter covers the literature review of the related works in the field of formal methods to provide a necessary background in formal verification and modelling. The chapter also covers the existing approaches in modelling web service architectures. First, Section 2.1 introduces the reader to the formal methods. Then, Section 2.2.1 introduces existing

frameworks and approaches to translate a web service specification into formal models defined in Section 2.1.

Background

This section introduces the basic concepts used for the development of WAFL. We start with the foundations of formal verification and its existing approaches. Then, we introduce formal modelling techniques and how such formal models are verified.

Formal Verification

Formal verification is a process of validating a system's behaviour using formal mathematical concepts. Formal verification is performed on formal models that follow basic axiomatic rules. One important feature of formal methods is the capability to handle non-deterministic processes in real life. Formal verification uses such techniques as model checking or theorem proving to infer the validity of a system's behaviour against its specifications. These approaches differ in the way they define the correctness properties of a verified system as well as how they construct the proof of the system's correctness.

Theorem Proving

Theorem proving comes from the formal logic field of mathematics. The idea is, given a set of predicates and axioms, to determine if some new logical statement is true or false using a formal proof. However, theorem proving in its pure form relies heavily on a human's decision-making and guidance for the optimal usage of the given formulas to obtain a formal proof. An algorithm for theorem proving was proposed by and later optimized by . These advancements allowed computer programs to find formal proofs for logical properties.

Verification of software using theorem proving was improved by introducing pre-conditions and post-conditions . Pre-conditions are predicates about the state of a program before executing a part of its code. Valid post-conditions are then inferred from the pre-conditions and the executed code. The obtained post-conditions are then used as pre-conditions to the next part of the code. This approach was extended to use post-conditions to infer a valid pre-condition for the post-conditions to hold . This theorem-proving method of computer software is based on Hoare logic, which was named after the author of this approach . However, proofs obtained using theorem proving are too large for real-world applications . Therefore, current state-of-the-art formal verification of software tends to focus on model checking .

Model Checking

Model checking is a more automated approach to formal verification in comparison to theorem proving . This approach keeps track of the possible states of a system when it executes. Each state contains some information about the system at that moment of the execution, e.g. variable values or messages sent from one agent to another. When some action occurs, the state changes and it is represented as a transition from the current state to the next state. The set of all reachable states and transitions between them is called a

transition system (TS). There are two main languages in which transition systems can be represented, namely Kripke Structures (KS) and Labeled Transition Systems (LTS) . These languages are interchangeable and have the same expressive power . Both of the approaches have a set of states and transitions between them. The difference between them is in the way they represent information about the changes in the system state. KS labels states according to their conformance to predefined atomic propositions. LTS, on the other hand, gives labels to transitions instead of states.

Model checking is a formal verification technique that argues about the correctness of a subject using the transition system generated by the behaviour of the subject. The transition systems are evaluated by exploring the state space, i.e. finding all possible paths on the TS. Therefore, transition systems usually must contain a finite number of reachable states for the model checking to terminate. However, there are trivial cases where inifinite state space can be model checked by representing infinite TS as finite-state automata .

The correctness of transition systems is determined by finding whether they satisfy some properties. The two most common languages for defining properties are Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), also referred to as branching temporal logic. These logics are based on a set of basic atomic propositions about a state in the transition system and a set of temporal operators. The atomic propositions are user-defined and differ depending on the purpose of the model checking. The $Ms \vDash p$ operator means that the atomic proposition p holds for the model M at state s. Temporal operators combine the atomic propositions into a single formula that checks a path with several states on the transition system. The atomic propositions are combined using LTL modal operators. Two main modal operators in the LTL are *next*, denoted as X or $\circ$, and *until*, denoted as U. The formula Xp means that the property p holds in every state reachable in one transition from the current state. The formula pUq claims that the property p holds if there is a path leading to a state where the property q holds. These two basic operators define more expressive operators, such as *always* and *eventually*. The operator $\square p$ is the *always* operator, meaning that the property p holds for every state reachable from the current state. The operator $\Diamond p$ is the *eventually* operator, meaning that the current state necessarily reaches a state that satisfies the property p. CTL uses the same temporal operators but in a different context. It also introduces two new operators - *all* and *exists*. These operators specify the application of the following property. The *all* operator, or Ap, means that the property p holds for all paths from the current state. The *exists* operator, or Ep, means that there exists at least one path on which the property p holds. ALthough CTL contains more operators, it is not more powerful that LTL. Each of the logics can express a language that can not be translated into another logic. CTL and LTL are often extended with time constraints as atomic propositions . In these languages, transitions between states have the additional property of how much time passes when the transition is activated.

Although CTL can reason about individual branches of TS execution, it is not so useful when verifying real-world systems . Moreover, LTL formulas tend to be easier for humans to read and understand.

Formal Modelling Languages

Transition systems by themselves are too abstract and become very large when they represent real-world systems. Therefore, there are many formal languages to model systems, which can be translated into TS for model checking or relevant formulas for theorem proving. Formal modelling is a process of creating an abstract model of a real system using one of the formal languages. We chose the model-checking approach because of its relevance and ease of use. We will cover current techniques in generating transition systems from different formal languages. These languages differ depending on the target systems that the language tries to model.

Finite State Automata

Finite state automata are the most basic modelling approach in model checking because they can directly translated into Kripke structures. The automata consist of a finite number of states, initial states that belong to the set of all states, and directed transitions between them. Atomic propositions of an automaton are represented with a simple function determining whether the system is currently at a specific state. Finite state automata are often used with extensions that add the time needed for transitions (Timed Automata) . The timed automata are then verified using timed CTL or LTL properties.

Petri Nets

Petri nets represent systems in terms of their internal flow of messages (tokens) . A Petri net consists of a finite number of places, where a message might be located, transitions, which control passing tokens from a source place to a destination place, and arcs connecting a place and a transition. Petri nets are great at representing conditional control flows of a system, which is one of the key concepts of software and logical hardware modelling. Petri Nets have many extensions and tools to take into account different aspects such as time (Time Petri Nets) and variables (Coloured Petri Nets) .

Process Algebra and CSP

Process algebra is a formal model specification that is designed for verifying concurrent systems . Process algebras reflect a simple imperative programming language, making it easy to model software. The process algebra languages model such concepts as variables, functions and control flow. The advantage of process algebras is that they allow the definition of concurrent processes. The concurrent processes can be executed in any order, therefore simulating the behaviour of several programs executing at the same time. A close relative of process algebras is Communicating Sequential Processes (CSP) developed by Hoare . Another formal language that borrows concepts from CSP and process algebra is Promela . Promela models are checked using the SPIN model checker, which verifies LTL properties. The atomic propositions are C-style conditions checking for values of global variables.

Interpreted Systems Programming Language

The study introduces an Interpreted Systems Programming Language (ISPL). This language can define multiple agents that work concurrently. An agent is defined by its current state, protocol, actions, and evolution. The current state contains local variables invisible to other agents. The protocol defines which actions the agent can perform at the given state. The actions are events observable by other agents and can be used as conditions for executing a protocol or an evolution. Evolution is the modification of the local state of an agent when its condition is satisfied. The ISPL specifications are verified using the Model Checker for Multi-Agent Systems (MCMAS) .

Event-B

Event-B is a formal language that defines the behaviour of a system using events. The system state is defined by a finite number of variables that can only contain discrete values. The evolution of the state is triggered by events, in which an action is associated with a guard. Guards define conditions under which the event is fired, and actions change the variables. Event-B models are checked by several different verification tools, such as Atelier B, Rodin, ProB, and BMotionWeb .

Rewriting Logic and Maude

Rewrite logic has a very flexible formal definition consisting of equations and rewrite rules . Rewrite logic can be used at a meta-level to define any other formal specification languages naturally. The language syntax is defined through the equational theory and its semantics by rewrite rules. This approach produces an extended rewrite theory, which can be reused in different contexts. Such extensions reduce the amount of effort for the definitions of more complex theories, which ends up providing a concise and expressive syntax with reliable semantics for the formal verification of large systems.

Maude is a formal specification language and verification tool that uses rewrite logic to define models and verify their correctness. It provides tools to explore the state space using searching for a specific state and LTL model checking. Infinite state spaces may also be verified using the search with a limited depth. However, Maude has a steep learning curve, which makes it difficult for newcomers to write formal models in the language.

Existing Web Service Verification

There are different approaches to formal modelling of web services. There are three main points of difference between studies: initial web service specification, formal language for model abstraction, and tool used for model checking.

Web Service Specification Languages

To overcome the difficulty of specifying formal models of web services, researchers developed tools that use existing approaches already employed by developers. The tools convert a conventional specification of a web service into a formal model and check it. Normally, when a company has many web services running and intercommunicating with

each other, the developers already have some specifications of web services in some specification languages. Previously, the most widespread approaches to web service specification were BPEL and WSCDL , hence a lot of research was done on verifying them.

However, these specifications have become obsolete, with more modern orchestration languages gaining popularity, such as Temporal , Cadence , and Conductor . However, there is little research on the verification of these modern orchestration engines. The following sections will describe studies grouped by the web service specification language they used to build formal models.

BPEL

BPEL is a web service specification language developed by IBM that represents an internal execution flow of web services . The main advantage of BPEL is that it can prove the correctness and completeness of the information flow from the endpoint that receives requests to every request (to other services) that the server invokes.

A study translates BPEL specifications into ISPL and checks them using MCMAS . The study provides a generic approach to generate an MCMAS formal model as well as temporal logic properties based on the BPEL specification of the web service. Other studies focus on translating BPEL processes into different formal models, such as rewrite logic in Maude , High-Level Petri-Nets , and Event-B specification .

WS-CDL

Web Service Choreography Description Language is another language for web service behaviour specification . The study proposes a standard way of translating from WS-CDL into timed automata, which can be then model checked by most of the available tools. WS-CDL was also translated into Petri-Nets and Event-B . However, BPEL and WS-CDL lost their relevance and it is unlikely that companies will use them to describe new systems. Therefore, it is important to consider more relevant techniques in formal modelling.

Log Tracing

Another approach to model web services is by request logs generated at runtime. A study presented a method for generating Petri-Net models of web services based on the produced log records of microservices. The practical aspect of the approach was proposed by . The method implies the usage of web proxy tools that log every incoming request and further communication between web services. They chose Fiddler proxy for debugging and obtaining log traces of requests. Based on the obtained log trace, the authors propose an algorithm to generate Promela code for the SPIN model checker . Then, they provide an example of correctness properties in Linear Temporal Logic (LTL), which is the main method for model checking in SPIN.

A newer study , on the other hand, focuses on the theoretical aspect of verifying that a generated Petri-Net represents the real system correctly. While log tracing may seem like a prospective solution for formal modelling of web services, the main drawback of such an approach is apparent. It is a post hoc debugging of failures and bugs that have already

occurred. This approach cannot find architectural design mistakes before the system is deployed into production.

Conductor DSL

Conductor is an orchestration language and engine developed by Netflix to manage web service compositions consisting of hundreds of independent microservices . The most novel approach was proposed by the study that translated the Conductor orchestration language into Petri-Nets. The research proposed an automatic technique for translating Conductor DSL into Time Basic Petri Nets. The authors also provided a tool named Conductor2PN. It translates Conductor specifications into Petri-Nets that can be verified by any available Petri-Net model checker. Then, the study provides examples of properties defined in Timed Computation Tree Logic (TCTL) . These properties can also be directly fed into Petri Nets verifiers.

Another study showed the scalability of the Conductor2PN approach for real-world microservice architectures. They tested the verification of a Train Ticket system and found seven violations of correctness properties.

Source Code

The internal behaviour of web services might also be modelled directly from the source code of the web service. The study about rewrite logic in web service description used an imperative programming language called WebScript. Although WebScript cannot send requests to other web services, this shows how the rewrite logic approach can fully replace other modelling methods because it can be modelled directly from the source code of web services. Such modelling would allow developers to generate a formal model of the system with compliance to its real behaviour not spending resources to properly abstract the web service into a finite state model. Moreover, developers would translate their orchestration code directly if they use Temporal or Cadence as orchestration engines.

Modelling from source code might close the gap between academic advancements in formal verification and the industrial use of orchestration engines. There are studies that use source code of programs to build formal models . However, there is little research done in the field of web services. The only study the author is aware of lacks key features of web services required for formal verification. These limitations will be discussed in Section 2.2.2.

The comparison of existing approaches in web service composition modelling are presented in Table [comparison].

Specification	Source	Formal model	Limitations
BPEL	[1]	ISPL	Obsolescence
	[2]	Rewrite Logic	
	[3]	Petri Nets	
	[4]	Even-B	
WS-CDL	[5]	Timed automata	
	[6]	Petri Nets	
	[7]	Event-B	
Log trace	[8]	Petri Nets	Need for prior system deployment
	[9]	Petri Nets	
Conductor	[10]	Petri Nets	Learning DSL
	[11]	Petri Nets	
Source code	[12]	Rewrite Logic	No communication between WS
	Proposed framework	Rewrite Logic	High complexity

WebScript Limitations

The proposed framework will take key concepts of WebScript to produce rigorous definitions. This section will describe the limitations of WebScript that our framework should overcome.

Although WebScript does allow defining formal models of web services from a source code, it has serious limitations making it unsuitable for verification of web service architecture. First of all, WebScript-defined services cannot send requests to other web services. This limits the usage of the framework to verification of interaction between only two services (client and server).

Also, WebScript limits its syntax due to conflicts with internal Maude syntax. For example, mathematical operators and boolean operators need to be preceded with an apostrophe (’+

instead of *+*). This framework also restricts the usage of variables in the code. The variables, like operators, need to be prefixed with an apostrophe.

This limitation of the framework expects developers to change the syntax of operations inconveniently. Most developers would often forget about this and cause syntax errors in their web service specification.

Furthermore, WebScript does not have any function definitions and calls, which might cause code repetition. This can be solved by defining a new operator in Maude that is reduced to the WebScript statements, however, this only inserts an inline code making it unsuitable for recursive functions.

Such problems with WebScript are caused by the limitations of Maude and conflicts with its core syntax. However, Maude also provides rich tools for the creation of meta-interpreters that can extend the core syntax of Maude with new features, or introduce a completely new syntax of a program. The proposed framework will utilize these tools to allow users without experience in formal verification to define formal models of web services directly from a source code.

Therefore, the proposed framework will target the practical application of formal verification for web service architectures. It will allow developers to use imperative languages similar to contemporary orchestration engines such as Temporal and Cadence. The advantage of this approach over Conductor DSL is that it will not require extra training for user of a new Domain-Specific Language but will utilize an already obtained experience in imperative languages.

Methodology

This chapter introduces WAFL - Web Architecture Formal Language - and how it works. The proposed framework is built as a Maude language extension using rewrite logic. First, Section 3.1 provides the necessary background for the Maude language because the further descriptions rely on the syntax of Maude. Next, the structure of the framework is overviewed in Section 3.2. Section 3.3 introduces PLIXE - Programming Language modelling Interactions with an External Environment, an imperative language that WAFL executes inside web services. It defines how the PLIXE works and how it is designed to be executed as part of web services. Then, Section 3.4 shows the behaviour of web service architectures including how services execute a PLIXE program and communicate with each other. Finally, Section 3.5 defines the assertion-based LTL model checking and how it is implemented in WAFL.

Maude Language

Maude is a modelling language that implements rewrite logic to design the behaviour of real-world systems. The rewrite logic operates on equational algebra using rewrite rules that describe how a system evolves with each application of a rewrite rule. The logic is split between modules defined within the `mod M is ... endm` construct, where `M` is the name of the module. Modules may import other modules using the `protecting M'` or `including M'`

statements, where `M'` is the name of another module. To avoid clashing definitions inside imported modules, it is possible to rename definitions with the following syntax:

```
protectingM'*(sortStoS').
```

where `S` is some sort in the module `M'` and `S'` is the new name for the sort that will be used in the importing module.

Maude operates by rewriting or replacing terms with other terms that belong to the same sort. A sort `S` is declared with the `sort S .` statement. Several sorts can be declared using the following statement - `sorts S S' S" .` Terms are constructed using operators. The operators are declared with the following syntax:

```
opF:S1S2...Sn->S[attrs].
```

Where S_1 S_2 ... S_n are the sorts of the arguments and `S` is the sort of the operator. The operator name `F` may be a single identifier, in which case the term is constructed as a prefix with arguments in parentheses. Alternatively, the name may contain `n` underscores (one for each argument) in any position of a mixfix form. The sort of the operator is on the right side of the arrow. The operator may contain additional attributes in the brackets. These attributes may include `ctor` denoting that the operator is a constructor. Constructor terms are in their normal form and are not rewritten into other terms. The `assoc` and `comm` attributes identify a mixfix operator as associative and commutative, respectively. Operators that are not in their constructor form are reduced to their simplified form using equations. An equation is defined with the following syntax:

```
eqT=T'.
ceqT=T'ifC.
```

The first line describes a simple equation. Each term that matches the term `T` is replaced with the term `T'`. The second line is a conditional equation that matches with an additional condition `C`. The terms `T` and `T'` may contain variables to generalize some parts of the term that do not need an exact match. Variables are defined as follows:

```
varV:S.
```

where the variable `V` belongs to the sort `S`. It is sometimes important that some operators do not match variables because they need to be resolved first using equations. To prevent them from matching variables, operators may belong to a *kind* instead of a sort.

```
opF:S1S2...Sn->[S].
—or
opF:S1S2...Sn >S.
```

This notation shows that the operator `F` does not belong to the exact sort `S` but to its *kind*, meaning that it will be replaced by another operator of this sort using equations. Maude provides the `red` console command to reduce terms to their normal form:

```
redT.
```

The command uses the defined equations to rewrite the term `T` until it uses only constructors.

Structure of the Framework

We propose WAFL - a new syntax for web service specification and a framework for verification of the models. This syntax is based on a simple imperative programming language that defines the execution of web services.

We will describe the extended rewrite logic of the verification framework for web services modelled using the proposed syntax. The framework is written completely in the Maude language to support the automatic verification of the formal models. It contains Maude modules that define different parts of the framework. Those parts are combined to define a single rewrite theory of web service behaviour. This section will overview the whole structure of the rewrite theory for a high-level understanding of the framework's purpose and capabilities.

A large part of the WAFL framework is PLIXE - the imperative language that is used to model the source code of web services. The framework includes modules for parsing and execution of PLIXE. These modules are then extended to the broader scope of WAFL, specifications of which may contain more than one instance of a PLIXE program running. The framework itself is a Maude extension. It has an input handler and a database. The input handler reads the user input and decides the action to perform. The database may be accessed to refer to previous actions such as model definitions or results of model checking. The overall structure of the framework from handling input to verifying models is presented in Figure 3.1.

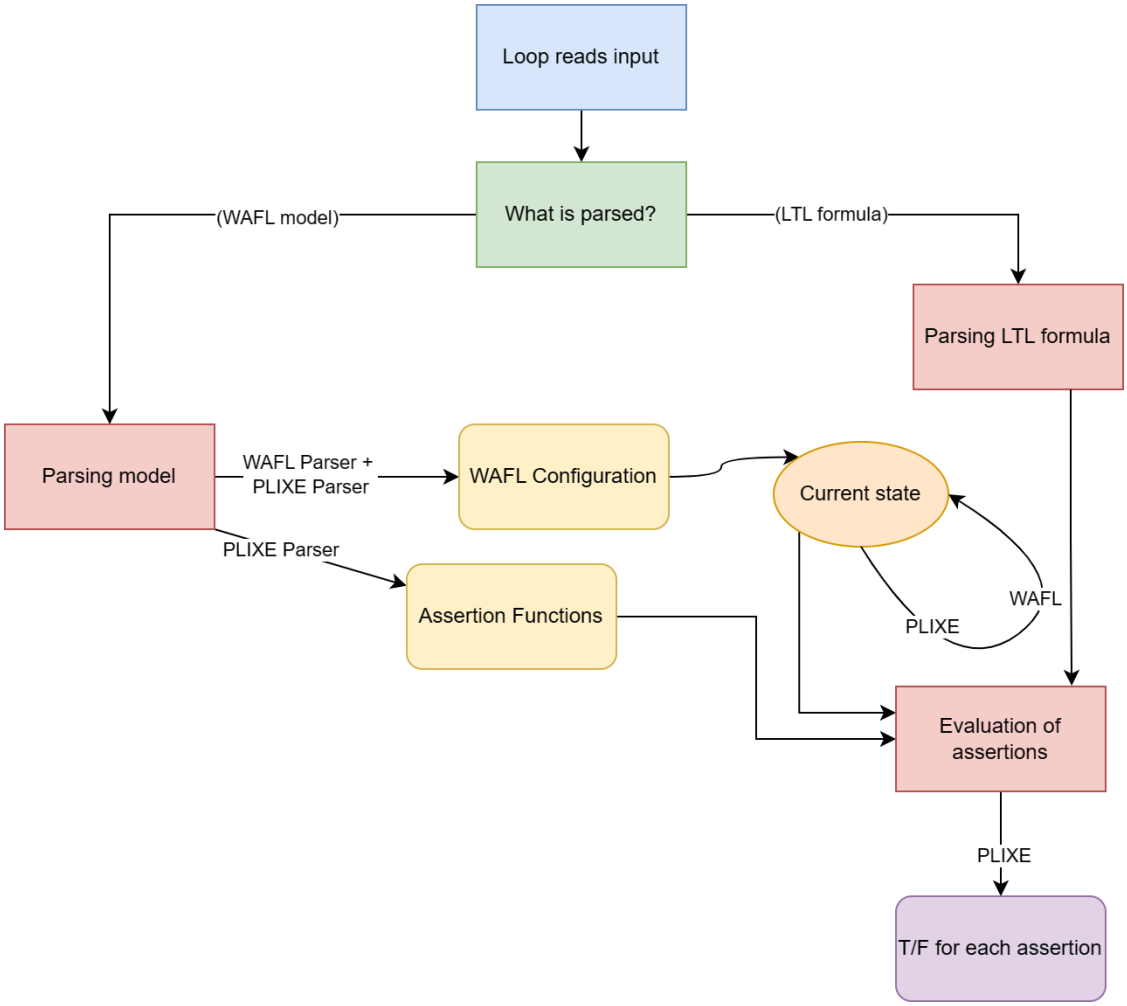

Figure 3.1. Structure of the Framework

First, the running loop decides the type of the input. It can define a new WAFL specification or verify an already defined model. In the case of importation of WAFL specifications, the framework parses the input and extracts two parts of the model - configuration and assertion functions. Configuration is a set of objects that defines the states of the architecture. These objects are modified with each rewrite step executed. On the other hand, assertion functions are separated from the configuration. These functions are written in PLIXE and verify some properties of a configuration, e.g. checking that each HTTP request follows the authorisation policy. The assertion functions serve as atomic propositions for the configuration. Both configuration and assertion functions are stored in a database to refer to them.

The loop can receive a command to verify the specification loaded before. The command triggers model checking of the specification on its adherence to an LTL formula. Then, it loads the current WAFL configuration from the database. Using the PLIXE defined execution and WAFL evolution of the configuration, it explores the state space of the web service configuration. The LTL formula is supposed to consist of calls to assertion functions as atomic propositions and classic temporal operators. Therefore, the loop loads previously

defined assertion functions, and for each function call in the LTL formula, it extracts information from the configuration and runs the assertion function with the configuration information available. The function asserts boolean conditions representing if the configuration satisfies the defined description. Then, the loop performs conjunction on the results of the asserted conditions and inserts the result into LTL operators of the formula.

For example, we can verify a web service architecture on that each HTTP request is responded to. We input a WAFL specification of existing services into the loop. The loop saves the descriptions as objects in a single configuration. Then, we extend the configuration by inputting a PLIXE program to send the HTTP requests. The initiation program is also stored in the configuration. When we want to check the model, we first define two assertion functions - *requestExists(field)* and *responseExists(field)*. The first function will check if there is an unresponded request with a certain field. The second function checks if there is a response to a request with the same field. To check the model we can run the command with the following LTL formula:

$$\square(requestExists(field) \Longrightarrow \Diamond responseExists(field))$$

The formula checks that if there is such a state of the configuration that there is a request with a certain field, it will necessarily reach a state with a response to the request.

Definition of PLIXE - a Simple Imperative Language

We define the formal executable semantics of a simple programming language to be used in the WAFL framework. The design of the language serves two purposes - to model the behaviour of web services and to make assertions from the LTL formulas. Therefore, the language is designed to pass values to the environment executing a PLIXE program. Such an environment might be, but not limited to, requests to web services or shared storage in databases, and LTL operators asserting a property. PLIXE includes all necessary features to define the abstract behaviour of web services, such as control flow, functions, lists, dictionaries, and interaction with the external environment. In this section, we will first define the programming language and its features. Then, in Section 3.3.2 we will describe the state of a PLIXE program during execution. Section 3.3.4 will define PLIXE's interaction with an external environment, such as sending requests to servers, responses to clients, and accessing shared storage.

Abstract Syntax Tree of PLIXE

PLIXE is a simple language defined using Maude rewrite logic. It has all the necessary features for modelling abstract behaviour of web services. Programs in PLIXE are constructed as Maude terms. Therefore, in this section, we will focus on Maude operator declarations for different language constructs in PLIXE. Construction of Maude terms has the limitation that it is not possible to use unquoted identifiers like variables and function names. This limitation will be overcome by parsing custom syntax at the meta level and converting the parsed syntax into the Maude terms that define the PLIXE program.

The PLIXE program is an abstract syntax tree (AST) containing a list of statements. Each operator corresponds to a node in the AST. All these nodes are defined in the `LANG-AST` module:

```
fmodLANG-ASTis
sortAST.
sortsExprASTStmtASTDeclAST.
subsortExprASTDeclAST<StmtAST<AST.
```

The `AST` sort contains all terms that can be at the top level of the program. The `StmtAST` subsort defines executable statements and function declarations. It will be used to represent PLIXE programs. Statements are combined using the associative operator `_;_`. That is, individual statements are separated with a semicolon, and their grouping does not affect the result. `ExprAST` and `DeclAST` are subsorts of the `StmtAST` sort. The `ExprAST` sort contains expressions such as mathematical and boolean operators. The `DeclAST` sort represents function declarations. Subsort declaration specifies that only statements of sort `StmtAST` are located at the top level. Moreover, the `StmtAST` sort is a supersort of the `ExprAST` and `DeclAST` sorts. These sorts represent expressions and function declarations respectively and they are also considered statements.

We define constructors of statement, expression, and declaration terms as prefix operators. For example, the `if` control statement is defined as follows:

```
op$ifelse:ExprASTStmtASTStmtAST->StmtAST[ctor].
```

The first argument is a condition for the `if` statement, and other arguments are conditional branches. Next, we define binary operators for expressions. They are also constructor operators in the prefix form. Terms like $E + E'$, which is an addition of two expressions `E` and `E'`, are denoted in PLIXE as `$add(E, E')`. These operators take two expressions as arguments and return a new expression. We also define literals for different atomic data types. The literals for strings, integers, floating point numbers, and booleans are terms of sort `Value`. They use corresponding predefined types in Maude to store actual values. For example, a string `"hello, world!"` is represented as `strVal("hello, world!")`. Variables are constructors of sort `Ident`. A variable `var` is represented with the term `id("var")`.

Program State during PLIXE Execution

In PLIXE, later statements of the program may refer to previous function and variable declarations. We store such information in the program state that changes after execution of a PLIXE program. We define this program state in Maude:

```
sortState.
op[_|_|_]:FuncStorageVarStorageStack->State.
```

The operator above defines a program state constructor containing three arguments representing declarations and computations that PLIXE executed. All defined functions are stored in the first argument of the state. The argument is of sort `FuncStorage`, which is a semicolon-separated unordered set of functions each having the following form:

```
func(strVal(S),params(id(V),...,id(V')),CODE)
```

where S is a string representing the name of the function, V and V’ are named parameters of the function, and CODE is a list of statements in the body of the function. Defined variables are stored in a term of sort VarStorage:

S:=N;S’:=N’;...;S”:=N”

where S terms are strings containing names of variables. The terms N, N’, and N” are natural numbers pointing to an index on the program stack, where actual values are stored.

The last part of the program state is its value stack. It holds all values that variables refer to. It is defined as an array of values with tracked size:

stack(N,0|->V_0;1|->V_1;...;N-1|->V_{N-1})

The size of the stack is given by the natural number N, and each term V_i of sort Value is associated with its index i on the left of the |-> operator.

PLIXE Program Execution

Execution of PLIXE programs is defined using equational logic in Maude. We define operator evalStmt to evaluate programs. The main idea of execution semantics is the modification of the program state after executing a statement. The evalStmt operator takes a statement with an initial program state and changes it to the final state. For example, if we consider an assignment var = 5, the evaluation will be of the following form:

```
evalStmt(assign(id("var"),intVal(5)),[...|...|stack(...)])
```

The evalStmt takes the assignment AST as the first argument and the initial program state as the second argument. The evaluation function modifies the given state and generates a new state as a result:

```
[...|"var":=0|stack(1,0|->intVal(5))]
```

The final state contains a variable var in the storage of variables and value 5 in the value stack. The variable points to the first element on the stack, which is value 5. Besides returning the resulting state after the execution, sometimes we need to compute a value returned from mathematical or other operations. Therefore, the signature of the evaluation contains three functions:

```
–evaluationofanexpressionwithresultingvalue
opevalExpr:ExprASTState->[EvalResult].
–evaluationofreferencevalue
opevalRefExpr:ExprASTState->[EvalResult].
–evaluationofastatementwithoutresult
opevalStmt:StmtASTState->[EvalResult].
```

All of them return a term of kind EvalResult, which is a pair of the resulting value from operations and the program state after execution. The evalExpr function returns an R-value as a value, evalRefExpr returns an L-value as an index to the program stack, and evalStmt returns only program state without value. We decide what function to call according to the expected behaviour of AST. For example, considering an assignment statement "a = 5 ;",

the whole statement is passed into the `evalStmt` function. The function then calls the `evalRefExpr` on the left side to obtain an index on the stack. Next, it calls `evalExpr` to get the resulting value on the right side and assigns this value to the index on stack obtained from the `evalRefExpr` execution. The result of these functions is of sort `EvalResult`, which is a pair of a value and a state embraced in curly braces:

The evaluation result above can be obtained by calling `evalExpr` on any expression that results in number 5. The resulting value is then assigned to a stack item with the index obtained from executing the `evalRefExpr` function.

Interaction of PLIXE with External Environment

In the context of web services, source code is not executed in isolation from the external environment. Instead, the execution relies on sending requests to other services and accessing shared storage. This environment is non-deterministic because other executions may access the same resources at the same time, and the order of access can not be inferred. Therefore, we define a semantics of communication between the PLIXE program and its environment. The communication relies on two new states of the execution - `dispatched` and `locked`. PLIXE program can enter these states using special functions `$dispatch` and `$lock`. The `dispatched` state is a way to send some information to the external environment and it does not interrupt execution. The `locked` state pauses execution of the program waiting for some value to unlock the state. A good example of such communication is reading a value from a persistent data storage on a web service. The PLIXE program will signal the service to read some resource on the data storage using the `$dispatch` function and then the program will lock its execution using the `$lock` function until it receives a value of the resource. We can group this behaviour into a new function `getPersistent`:

```
func("getPersistent",params(id("key")),
$dispatch(id("key"),getPersistentValue);
$lock(id("res"),getPersistentValue);
return(id("res"))
)
```

The code above shows the function definition with the name `getPersistent`, one argument `key` representing the name of the requested resource, and the function body with `$dispatch`, `$lock`, and `return` statements. The `$dispatch` statement sends a message containing the value of the `key` parameter through a communication channel named `getPersistentValue`. The communication channel is defined separately using the following operator definition:

```
sortIO.
opsetPersistentValue:->IO[ctor].
```

The IO communication channels are used to differentiate between different types of communication. The environment takes different actions depending on the IO channel it receives a request from. Next, the `getPersistent` function locks the state using the `$lock` statement. It waits for a value sent over the `getPersistent` channel and writes the value

into the `res` variable. The variable is then returned from the function. In the context of web services, The function can be used by different concurrent processes to retrieve some resources available.

The `$dispatch` statement generates a partially evaluated state of the following form:

```
dispatched(,strVal(SRC),getPersistentValue)
```

The state is embraced in the `dispatched` operator, which prevents further program evaluation using the generated state. The `dispatched` operator contains a valid evaluation result inside the curly brackets with a value `V` and a valid state. The second argument is a string value `SRC` representing the name of the resource being accessed. And the last argument is the `getPersistentValue` communication channel.

Locking the program state results in the evaluation result of the following form:

The `$lock` function wraps the program state into the `locked` operator to prevent evaluations of the next statements. It contains an actual valid state that is used to continue execution, a natural number `N` which is an index on a stack where the received value will be put, and the communication channel. Unlocking the state is performed using the `unlock` operator, which passes a value into the locked state. The value is then assigned to the `N`-th element on the stack.

Web Architectures Configuration in WAFL

Next, we define the configuration of a web service architecture and its behaviour in rewrite logic. The configuration is a medium between objects that combines them into a single term. The objects in the configuration can interact with any other object if some rewrite rule allows it. In this section, we will describe how a web service architecture is defined using Maude terms as a configuration and how the state of such a configuration changes according to the behaviour defined with rewrite rules. WAFL uses Maude's predefined `CONFIGURATION` module to define a blueprint for creating a medium for objects to interact with each other.

Web Service Definition

Web services are defined with an operator that produces an object term. The definitions of objects present in the web service architecture are given in the `SERVER-AST` module:

```
modSERVER-ASTis
...
sortSemServer.
opserver:SemComponentList->SemServer[ctor].

sortSemService.
subsortSemService<Object.
opservice:SemServiceIdSemServer->SemService[ctorobject].
endm
```

The web service is defined with the `service` operator. This operator produces a term of the `SemService` sort, which is a subsort of `Object`. This makes services usable as parts of a configuration because configuration consists of objects. A service contains a unique `SemServiceId` that distinguishes it from other services and makes it referencable as a target for HTTP requests. Next, the service contains a `SemServer` term constructed with the `server` operator. The server is an environment that contains all information about the service. The server contains `SemComponentList`, which is an unordered list of such components as currently executing processes, values stored in a database, and endpoint definitions. Here is a valid term describing a service object:

```
service("systemB",
server(
code(...),
routes(
route("/create","create"),
route("/retract","retract"),
route("/send","send")
),
locks(empty),
persistent("agreements",map(empty))
)
)
```

This is a parsed representation of a WAFL service as a Maude term in its prefix form. The service is identified by the string `"systemB"`. All information needed for the execution of the service is contained inside the `server` block. The `code` component contains the AST of a PLIXE program. The program represents a source code of the web service. HTTP endpoints are parsed into the `routes` term, existing locks on resources are present in the `locks` operator, and persistent resource `agreements` is defined as a separate component equal to an empty map. Persistent resources represent a database item that can be read by concurrent processes of the service and persists after the processes end their execution.

The execution of workflows inside the services is controlled by the `process` component. The process contains an execution environment for a PLIXE program and information about the connection that initiated the process. The behaviour of processes is defined in the `SERVER-SEMANTICS` module:

```
modSERVER-SEMANTICSis
opprocess:SemConnectionEnvironment->SemComponent[ctor].
opprocess:SemConnectionEnvironmentNat->SemComponent[ctor].
```

The first definition of the `process` operator has information about the incoming HTTP connection that initiated the execution of the process as the first argument. The second argument is the PLIXE program execution environment. The second definition of the `process` operator has an additional argument for processes that wait for a synchronous HTTP request to return a response from another service. The natural number corresponds to the id of a connection the process waits for. The `Environment` sort is an environment for executing PLIXE programs. It is defined in another module named `LANG-EXECUTION`:

```
modLANG-EXECUTIONis
sortEnvironment.
op:Configuration->Environment[ctor].
opinitiate:StmtASTFuncStorage->Environment.
eqinitiate(CODE,FUNCS)=
.
```

The code above describes how an execution environment is initiated inside a process. The environment contains an executor object. The object has an evaluation result in the `currentResult` attribute. All the communication from the PLIXE program will be read from the `dispatched` terms that bubble up on top of this term. The `waiting` attribute distinguishes if the program is locked and waits for an external input to unlock the `currentResult`. The `currentResult` attribute contains a starting state of a service program after executing its `code` block that is passed as the `CODE` variable to the `initiate` operator. The `FUNCS` variable contains predefined functions for sending requests, accessing storage, and working with exclusive locks. Therefore, it contains all defined user functions and handlers for external triggers provided in the `code` block. When something triggers an execution of the code, it should give the name of a function to execute. It is passed with a special message named `callController`:

```
callController(C,V)
```

where `C` is the name of a handler function, and `V` is a payload of an incoming request. The message is consumed by the `executor` object and its `currentResult` updates to the result of the `CODE` program appended with calling the function `C`.

Communication Between Services

Communication with web services is handled by the PLIXE program inside the code block of a service. The outcoming requests are dispatched as values inside the execution environment. We need to translate the dispatched values into messages to other services. The process of a synchronous HTTP request from a PLIXE environment to other services and backwards is illustrated in Figure 3.2.

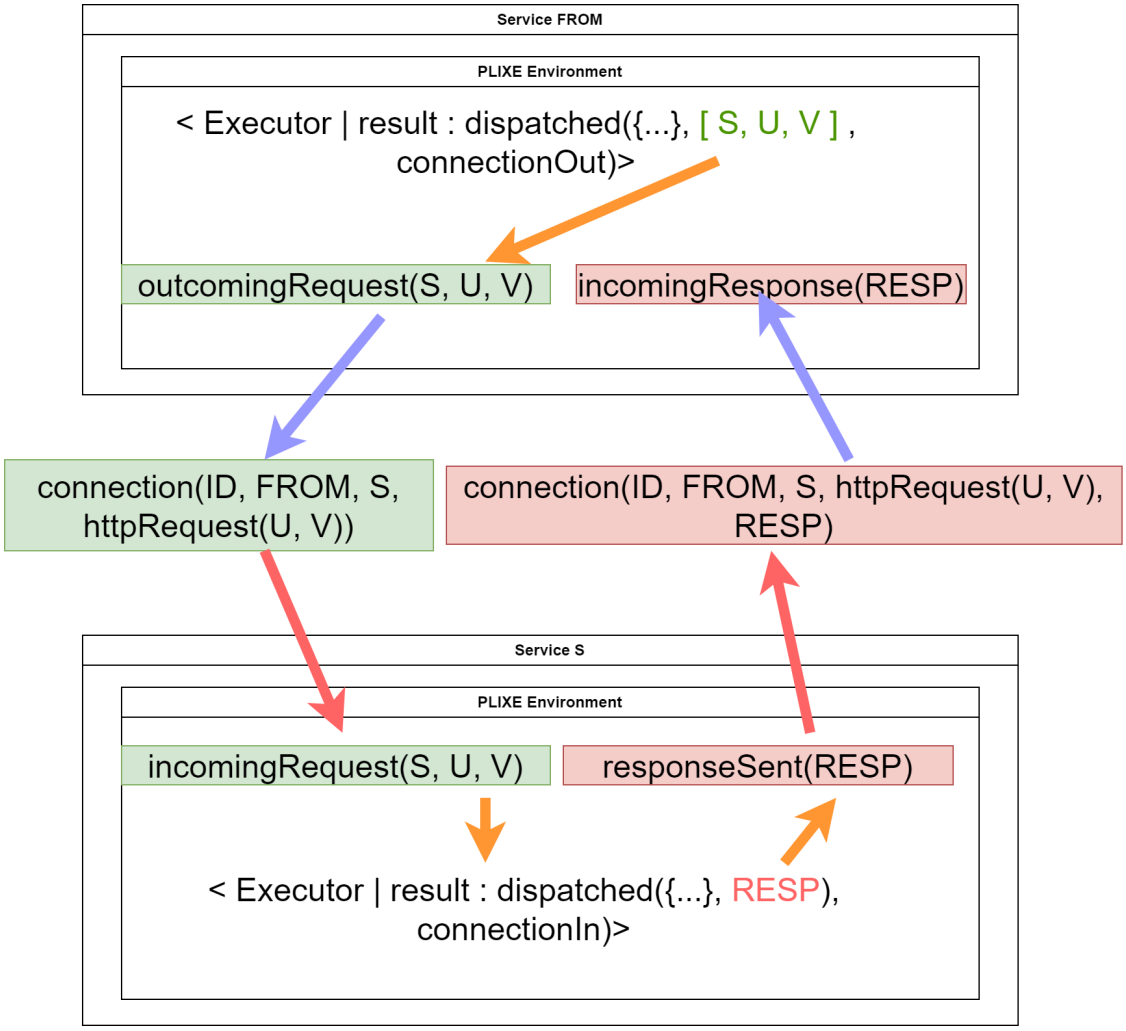

Figure 3.2. Communication of PLIXE program with services via HTTP request.

If there is a dispatched value in the current evaluation result, the environment expects it to be a list of three values - the name of a target service (S), the URL of the endpoint (U), and the payload value of the request(V). These values generate the `outcomingRequest` message. The message is read by a service and a new connection is generated. The connection contains its unique id, the name of the service sending the request, and the values emitted by the PLIXE program. When the target service sends a response to the connection, the response is read by the sending server and is passed into the `incomingResponse` message for the executor. The evaluation result of the executor was locked after sending the request. When the message is consumed, the executor unlocks the result with the payload of the response.

Verification of WAFL Models

To verify WAFL models we need to define properties of the web services. The properties should correspond to atomic propositions for model checking. Our approach is assertion-based properties defined by developers as functions that assert the validity of the

state using assertion functions. We want to comply with common techniques software engineers encounter when testing their services. Therefore, we treat each state of the composition as a unit test case. The validity of the state is decided by running a PLIXE program that makes a number of assertions. If all assertions are true, then the state is valid. Otherwise, the state is invalid. This approach is familiar and intuitive for most programmers that who ever encountered unit testing. We approach this by extending Maude's predefined Linear Temporal Logic (LTL) model checking logic. The extended logic allows the use of assertion code together with common LTL modalities, such as *always* and *eventually*. Section 3.5.1 will describe how a PLIXE program can make assertions about web services, and Section 3.5.3 will describe fusing these assertions into the Maude LTL checking. Section 3.5.4 will present the combined syntax of LTL and PLIXE's assertions.

Accessing Web Services' State from PLIXE program

To make assertions about web services inside the PLIXE code, we need to transfer the state information about the web services into the PLIXE's program state. Therefore, we define a function that converts information about web services into the PLIXE state. The global state of a set of web services is of sort `Configuration` as defined in Section 3.4. The conversion is defined in the separate module `COMPOSITION-TO-LANGUAGE` with the `toState` operator, which makes the conversion.

```
modCOMPOSITION-TO-LANGUAGEis
protectingSERVER-SEMANTICS.
protectingCONVERSION.
optoState:Configuration >State.
optoState:ConfigurationState >State.
opassertion:->IO.
opserverToIOVal:SemComponentList->IOValue.
opconnectionToIOVal:SemConnection->IOValue.
```

The module imports the `SERVER-SEMANTICS` module that defines the behaviour of a WAFL configuration. The module `CONVERSION` is a predefined Maude module that makes conversions between strings and numbers. The first declaration of the `toState` function takes the web services environment and generates a PLIXE program state that contains all information about web services. The second declaration is an auxiliary function overload that combines the generated program state with another state in the second argument. The `>` arrow means that the sorts of the operators are *kinds* of `State` so that the term does not match for rules that expect terms of the sort `State`. Specifically, using the *kind* prevents terms of the form `toState(...)` be used as valid states in PLIXE code evaluation equation described in 3.3.3. The `toState` function encodes all information about web services as variables and adds an assertion function to communicate the assertion results to the model checker. The `assertion` operator defines the communication channel over which PLIXE passes the result of the assertions to the model checker. The operator `serverToIOVal` translates all important components of a web service into a state-independent `IOValue`. The `IOValue` sort embraces all possible values of PLIXE but lists and dictionaries contain actual values, and not references to a value stack. This simplifies the construction of values by ignoring the current stack and program state and passing this job to the already defined

function `toVal` that integrates an `IOValue` into the program state. The operator `connectionToIOVal` serves the same task but specifically for open connections not yet processed by the receiving end of a request or a response.

Now, we can define the `toState` operator. This operator defines all variables that will be used for populating the state and introduces the assertion function. Afterwards, it calls the second overload of the `toState` operator to fill the defined variables:

```
varCONF:Configuration.
eqtoState(CONF)=toState(CONF,[
func("assert",params(id("value")),
$dispatch(id("value"),assertion))
|"services":=0;"connections":=1
|stack(2,0|->dictVal(empty);1|->listVal(emptyList))
]).
```

The `toState` function generates the blueprint state and passes it to the auxiliary function for filling it out. The blueprint state contains one predefined `assert` function, the `services` variable referencing an empty dictionary, and the `connections` variable referencing an empty list on the stack. The `assert` function is an IO function that communicates with the external environment. It takes one argument `value` and dispatches it to the defined `assertion` IO channel. The value is expected to be a boolean value representing if the state is valid. The constructed state is then passed to the second overload of the `toState` operator.

The second definition of the `toState` operator fills the `services` dictionary and the `connections` according to web services and connections in the `CONF` configuration. It finds a term of form `service(S, server(SERV))` in the configuration. The term corresponds to a web service with the name `S` and components `SERV`. The function uses the `serverToIOVal` operator to convert `SERV` into an IO value and then inserts the obtained value into the resulting program state. The function puts the converted value of the server into the `S` field of the `services` variable. The information about service `S` will be available through the expression `services.S` in a PLIXE program. The `toState` operator finds open connections as well. For each open connection, we push its information into the `connections` variable. The `toState` operator finds a connection object `CONN` of sort `SemConnection`. The connection is translated using `connectionToIOVal`, pushed to the value stack of a program state, and its address is then appended to the list of connections under the `connections` variable.

Here is an example of the usage of the `toState` operator:

```
modCOMPOSITION-LANGUAGE-TESTisprCOMPOSITION-TO-LANGUAGE.
opconfig:->Configuration.
eqconfig=service("s",server(persistent("v",intVal(5)))).
endm

Maude>redtoState(config).
...
resultState:[func("assert",...)|
```

```
"connections":=1;"services":=0|
stack(5,0|->dictVal(d("s",4));...)]
```

The module above defines a configuration `config` with a service with the name `"s"`. The `toState` operator translates it into a PLIXE program state with the `services` variable equal to the dictionary of the form . The dictionary contains the information about the persistent variable `v` inside of the service `s`. This translation can now be used to verify the global state of the web services from a PLIXE program using `assert` function.

Defining Atomic Propositions using Assertions

In this section, we will describe the integration process of our assertion-based verification into Maude's predefined LTL model checking. The `assert` function defined in Section 3.5.1 dispatches a boolean value to the `assertion` IO channel. As described in Section 3.3.4, the dispatched value bubbles on top of the state via the `dispatched` constructor and waits to be read by the external environment. In this case, the external environment is an operator that reads all assertion results from a program and resolves them using logical conjunction.

We introduce the `satisfies` operator defined in a separate module. The goal of the operator is to reduce a web service configuration paired with a code that makes assertions into a single boolean value representing if the configuration satisfies the assertions made in the code. The `satisfies` operator takes the web services configuration and PLIXE code with assertions, checks that the code runs successfully, ensures that web services do not generate errors, and returns the result of assertions.

```
opsatisfies:ConfigurationStmtAST >Bool.
varCONF:Configuration.
varCODE:StmtAST.
eqsatisfies(CONF,CODE)=if
checkValid(evalStmt(CODE,toState(CONF)))andnoErrors(CONF)
then
checkAssertions(evalStmt(CODE,toState(CONF)))
elsefalsefi.
```

The first argument is the web service configuration `CONF`. The second argument is the PLIXE program `CODE`. The operator uses the `toState` function defined in the previous section to convert the configuration into a program state. Then, it evaluates the program using the generated state. The `checkValid` function checks that the program executes correctly, and the `noErrors` function finds errors that were generated by web services while processing requests. If there are no errors, then the `satisfies` operator returns the result of computing all the assertions. Otherwise, it returns false, which is equivalent to the case when some assertions failed.

The `checkAssertions` operator recursively reads all dispatched boolean values bubbled on top of the program evaluation. For example, it reads evaluation results of the following form:

```
dispatched(
dispatched(,boolVal(B),assertion),
boolVal(B'),assertion)
```

In this case, two boolean values `B` and `B'` were dispatched to the `assertion` channel. That is, the `assert()` function was called twice, first asserting some expression that evaluates to `B'`, and then another one that is equal to `B`. The `checkAssertions` function computes $B' \wedge B$ when called on the example term.

LTL Model Checking with Maude

Finally, we integrate the defined `satisfies` operator into the Maude's predefined LTL model checking. We define an interface for the model checker to specify LTL formulas that use PLIXE programs to describe some of the properties. For example, a formula `[] (JohnSigned() -> <> SignVisible())` means that if John sends a request to sign an electronic document, his signature will eventually be visible to everyone. Operators `[]`, `<>`, and `->` are predefined LTL operators in Maude representing the ▫, ◊, and logical implication operators. The ▫ and the ◊ are the modal operators defined in Section 2.1.1.2. The `JohnSigned()` and `SignVisible()` properties are PLIXE function calls that dispatch assertions.

The module named `COMPOSITION-PREDS` contains definitions required for using PLIXE programs in the Maude's predefined LTL model checking.

```
modCOMPOSITION-PREDSis
including(SATISFACTION+LTL+MODEL-CHECKER)
*(sortStatetoCompositionState).
protectingASSERTION-CHECK.

opstatePair:ConfigurationStmtAST
->CompositionState[ctor].
opassertions:StmtAST->Prop[ctor].
```

The module includes predefined modules `SATISFACTION`, `LTL`, `MODEL-CHECKER`. We renamed the `State` sort from these modules into `CompositionState` because the name clashes with the sort for the PLIXE's program state, which is also called `State`. The `CompositionState` sort embraces terms that denote the states remembered by the model checker. We define an operator `statePair` of sort `CompositionState`. This operator declares that model checking will run over pairs of a web service configuration and a starter code that defines assertion functions. For example, the code may declare a function asserting that there is a request sent from service A to service B. This function can then be used in the `assertions` property defined on the next line. It takes a code that runs assertions and constructs an LTL property of sort `Prop` from the `SATISFACTION` module. The described behaviour is defined using the following equation:

```
varCONF:Configuration.
varsCODECODE':StmtAST.
eqstatePair(CONF,CODE')|=assertions(CODE)
=satisfies(CONF,CODE';CODE).
```

The |= operator is defined in the SATISFACTION module and corresponds to the ⊨ operator defined in Section 2.1.1.2. We define that a state pair satisfies some assertions in the same cases as when the satisfies function returns true. The satisfies function is called on the provided configuration and on the code with assertions prepended by the code defining assertion functions.

Therefore, we defined an assertion-based model checking for web services and integrated it into the Maude predefined LTL module. Now the assertion code can be used in the Maude modelCheck function.

```
Maude>redinCOMPOSITION-PREDS:
[]assertions(call("test",args(empty))).
reduceinCOMPOSITION-PREDS:
[]assertions(call("test",args(empty))).
rewrites:1in0mscpu(0msreal)( rewrites/second)
resultFormula:FalseRassertions(call("test",args(empty)))
```

The lines above show that Maude successfully parses and reduces an LTL formula containing an assertions property. The formula is an *always* operator applied on the assertions made by calling a function named test without arguments.

And finally, we can use Maude's MODEL-CHECKER module to find counterexamples to the defined properties. We can define some shortcut operators to further simplify commands within a new module PRED-TEST:

```
modPRED-TESTis
protectingCOMPOSITION-PREDS.

opconfig:->Configuration.
eqconfig=none.

opfunctions:->StmtAST.
eqfunctions=function("test",params(empty),
call("assert",args(bool(true)))).

oppropCode:->StmtAST.
eqpropCode=call("test",args(empty)).
endm
```

The module defines an empty web service configuration config, code functions defining a function test(), which always asserts true, and code propCode that calls the test() function. Now we can use everything we defined in this section to build LTL formulas using the PLIXE's AST and run the model checker to find states that do not satisfy the formula.

```
redmodelCheck(statePair(config,functions),
assertions(propCode)).
```

In this case, the propCode always calls test() function, which asserts true. Therefore, the model checker returns result Bool: true.

Combined LTL Syntax

Although the defined syntax is ambiguity-free, it is too bulky and does not resemble a real programming language. The `propCode` operator that only calls the `test` function could be denoted as simple as only `test()`. Therefore, we want to define a syntax to shorten the PLIXE program in the LTL formulas. It should be similar to the syntax defined for the code component of web services. Specifically, we want the syntax to parse such expressions as `[] test()`, where `test()` is a function call defined in the `LANG-SYNTAX` module:

```
op_‘(‘):Token->Funccall.
```

We define the combined syntax of Maude's LTL formulas and lang's function calls in a module named `LTL-EXPR`. We do not need to define a new syntax but only allow the use of function calls in LTL formulas.

```
fmodLTL-EXPRis
includingLTL.
includingLANG-SYNTAX.

subsortFunccall<Formula.
endfm
```

The module above combines two modules. The `LTL` module defines LTL operators, such as logical *and* `/\`, *always* operator `[]`, and *implies* operator `->`. The `LANG-SYNTAX` was defined previously and it declares a JavaScript-like syntax for the PLIXE. The subsort declaration integrates the function call operator into LTL formulas so that it is possible to parse such formulas as `[] validState()`. Since the syntax of PLIXE is not parsable by Maude directly, we use the `metaParse` function together with processing functions to translate this syntax into valid Maude terms. The processing is defined in the `LTL-PROCESS` module.

```
modLTL-PROCESSis
...
opprocessLtl:Term >Formula.
```

The module defines the `processLtl` operator to map parsed meta terms into the LTL formula. The operator constructs an LTL formula with `assertions` properties defined in Section 3.5.3 out of the shortened syntax from the `LTL-EXPR` module.

Now, it is possible to parse strings with the syntax of PLIXE function calls combined with the Maude LTL operators. We define a shortcut function `parseLtl` to convert a list of Qid's, which is a tokenized string in Maude, into an LTL formula. The operator parses the input as a meta term for an LTL formula. Then, the meta term is passed to the `processLtl` function to move the formula down from the meta representation. Therefore, this function can be used to generate LTL formulas with assertion properties from their shortened syntax:

```
Maude>redparseLtl(tokenize("[]test()")).
reduceinLTL-PARSING:parseLtl(tokenize("[]test()")).
rewrites:209in10mscpu(14msreal)(20900rewrites/second)
resultFormula:FalseRassertions(call("test",args(empty)))
```

We passed a list of tokens from the short LTL formula `[] test()` into the `parseLtl` function and the result is a valid LTL formula with the call to the `test()` function as a PLIXE AST inside of the `assertions` operator. The formula can then be used in model checking. For example, given the previous example of the `PRED-TEST` module with predefined configuration and function definition, we can replace the `assertions` term by the `parseLtl` function with the shortened LTL syntax passed as an argument.

```
Maude>redmodelCheck(statePair(config,functions),
>parseLtl(tokenize("[]test()"))).
...
resultBool:true
```

The model checker returns the true boolean value, which means that the model always satisfies the assertions made in the `test()` function. The `test()` function indeed always asserts `true`, so this is an expected behaviour.

Results

This chapter presents results obtained from developing the WAFL framework. Section 4.1 describes the syntax of the framework and its usage from the Maude console. Section 4.3 provides a real-world example of the framework usage by describing the WAFL model of the example system and outputs of the framework after checking the model.

Syntax and Usage of WAFL

The WAFL framework provides an easy-to-understand syntax to define web service architectures. The syntax is parsed using the Maude predefined tools and transformed into the terms presented in Sections 3.4 and 3.5.3. Services are defined using the `service` block. An example definition of a simple service is provided in Listing 4.1

Listing 4.1: Example of a service definition.

```
(service S {
    listen {
        "channel" -> queueHandler
    }
    routes {
        "/url" -> httpHandler
    }
    persistent var = 0 ;
    code {
        function queueHandler(message) {
            lock("resource");
            setPersistent("var", 5);
            unlock("resource");
        }
        function httpHandler(request) {
            respond("ok");
```

```
        }
    }
})
```

In the listing, the service `S` is defined using the `service` keyword. The service listens to the channel in the `listen` block and handles messages received using the `queueHandler` function. HTTP requests sent to the `"/url"` endpoint are handled using the `httpHandler` function. The service contains the persistent resource named `var` and it is initially set to 0. The `code` section of the service contains the code base written in PLIXE. The code defines the handler functions for the `listen` and `routes` blocks. The PLIXE program syntax resembles JavaScript code with similar function definitions and control flow structures. The `queueHandler` function locks a resource preventing parallel executions until the resource is unlocked and sets the `var` resource to 5. The `httpHandler` function responds to the request with an `"ok"` string. The input is wrapped in parentheses to instruct the Maude console to pass the string into the loop mode.

The system initialization is performed using the `init` block with a PLIXE program that runs once when the system starts. The example of an `init` blocks is given in Listing 4.2.

Listing 4.2: Example of an init block definition.

```
(init {
    message("channel", 1);
    request("S", "/url", 5);
})
```

The program sends a value of 1 to the `channel` queue and a value of 5 to the `"/url"` endpoint of the service `S`. The assertion functions are defined in the `props` block shown in Listing 4.3.

Listing 4.3: Example of an assertion block definition.

```
(props {
    function prop() {
        assert(services . S . persistents . var == 5);
    }
})
```

The block defines a function `prop()` that makes an assertion that the `var` resource is equal to five. The defined model can be checked using the LTL model checker. The checking is invoked using the `ltl check` command that accepts an LTL formula as an argument. The atomic propositions of the WAFL model are calls to assertion functions defined in the `props` block. We can check that the `prop()` assertion holds for every reachable state of the model using the command in the Listing 4.4.

Listing 4.4: Example of a model check command.

```
(ltl check [] prop())
```

Since the property does not hold at the initialization step of the system where `var` equals 0, the framework outputs the counterexample with the full trace of the system state.

WAFL Applications for Common Web Architecture Patterns

Contemporary web architectures usually employ widely accepted practices in terms of their communication. The study highlights two main web architecture design principles - orchestration and choreography.

Choreography design is based on sending messages to message queues that distribute them between previously subscribed web services . Microservices deployed over a wide area network may use such design to alleviate the temporary unavailability of web services. However, choreographed architectures are hard to manage when the business processes rely on long communication chains between many microservices. The main way of sending information between services is asynchronous message queues, and a service sending the message does not know when the receiving service will finish handling the message. Therefore, the receiving service must signal the next one in the chain after finishing its execution, constituting an uncentralized system with each service unable to construct a complete workflow on its own.

Orchestration, on the other hand, relies on HTTP requests between web services. The main advantage of such an approach is that a web service pauses its execution until it receives an HTTP response . The response indicates that the called web service finished its execution, and the caller may send a new request to the next service in the chain. Such control allows for defining whole workflows in one function of a web service. The main concept of orchestration is collecting such workflow functions in one web service that 'orchestrates' other servers responsible for small subtasks.

Although choreography and orchestration solve some infrastructure problems, both of them are prone to concurrency-related bugs in different cases. In this section, we will present a common bug caused by the concurrency named lost update. The definition of lost update is well studied in the field of database design and most contemporary database management systems deal with the lost update problem by default. However, it is easy to overlook such mistakes on some occasions. For example, it is widely accepted practice to execute 'background jobs' to handle computationally expensive algorithms. These background jobs are expected to execute only one instance of an algorithm and therefore such algorithms do not account for the lost update. Now, let us say someone wants to start the algorithm manually, adding a button to do so. At this point, the lost update can be introduced because the button can be pressed too fast and two instances of the algorithm will run concurrently.

We will describe how one can locate the lost update problem in orchestrated and choreographed web service architectures using WAFL. We will further describe the lost update on the examples provided in the following sections.

Verification of Web Service Orchestration

Let us consider an orchestrated web service architecture for an online banking application. It consists of a bank website that receives input from users and calls the balance service that keeps track of user balance. The balance service can update the balance value by accessing it from the storage, changing the value, and putting it back. This process is illustrated in Figure 4.1.

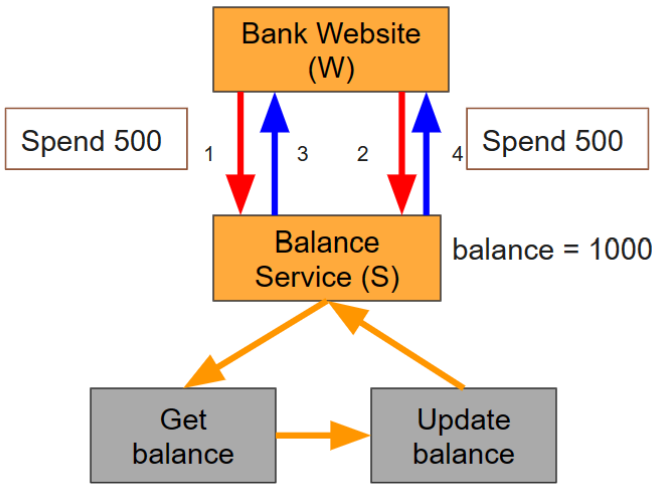

Figure 4-1: Orchestrated Bank Website Architecture Example.

We can check if the lost update happens in such architecture by sending two concurrent requests (1 and 2) to spend 500. If the initial value of the balance is 1000, the correct behaviour will lead the balance service to have 0 on the balance at the end. Let us first define the assertion function that checks this property:

```
props {
    function total(amount) {
        assert(
            services . balanceService . persistents . balance == amount
        );
    }
}
```

We define the `total` function that asserts whether the balance value of the balance service is equal to some amount. Then, we can construct the LTL formula that utilizes the `total` function to check whether the balance is equal to zero:

$$\Diamond \blacksquare total(0)$$

This formula checks that the balance will *eventually* be equal to zero and *always* stay like this afterwards. Running the WAFL command (`ltl check <> [] total(0)`) outputs a counterexample with the balance value ending up equal to 500, indicating that one update to spend 500 was lost. The counterexample output contains a state where both processes

handling each own request access the balance value equal to 1000. Therefore, each process decreases the value by 500 and puts 500 back into the storage. This is a trivial case of the lost update problem caused by a process reading a value that is about to change. The update made by the first process will be then rewritten by the second process that has accessed the old value.

Verification of Web Service Choreography

Although web service architectures following orchestration design have different ways of communication, they can have the same problem of lost updates. Let us consider a similar architecture of the banking website but with choreographed architecture. Figure 4.2 illustrates such architecture that may introduce a lost update.

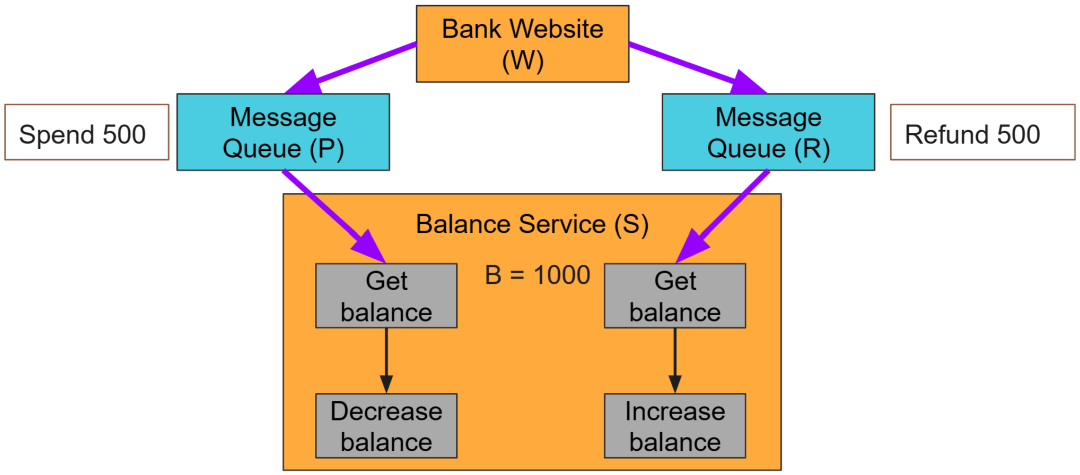

Figure 4-2: Choreographed Bank Website Architecture Example.

Here, the website sends two messages to concurrent message queues (P and R). One of the messages instructs the balance service to spend 500. After receiving a message from this queue, the balance service accesses the balance value, decreases it, and puts the new value back. The second message queue contains a message to refund 500, reading which will instruct the balance service to increase the current balance. As before, the initial value of the balance is 1000. Hence, it must be equal to 1000 after spending and refunding the same amount of 500. We can reuse the assertion function named `total` from Section 4.2.1 and construct the following LTL formula:

$$\Diamond \, \blacksquare \, total(1000)$$

As before, the formula checks that the balance will be equal to 1000 at some point and will not change afterwards. The result of model checking this formula gives us a counterexample where the processes handling message queues P and R read the same value of the balance and the update of one of the processes is lost.

Example - Electronic Document Exchange

The developed framework was tested on a real-world example of a web service architecture. The example models a system of electronic document exchange that is rapidly gaining popularity in digitalized government structures. One of the critical properties in such document exchange systems is moving two-sided contracts through the pipeline and ensuring that communication between contract sides is not compromised because of delays in the network. The WAFL specification code is presented in Listing A.1. The example model describes a web service architecture consisting of three services. System A is a document management system of company A, and it is about to receive a document from an external system of company B. The `init` block describes a high-level business process from an end-user standpoint. First, they send a request to create a two-sided contract, then send it to another party, and retract it afterwards. From the user's view, this is a consequential process without any concurrency involved. However, the architecture is set up in a way that the creation of documents and their retraction is done concurrently via the service named `centre`. After sending the document to company A and retracting it, system B emits messages to two concurrent message queues named `newDocument` and `receipt`. The order of their consumption is non-deterministic and can change depending on the current load on each queue. Therefore, system A must make sure that it does not ignore retraction receipts if they are consumed before receiving the document creation. Therefore, it saves the receipts in the `retracted` persistent list for such cases. However, when a retraction and creation are handled at the same time, two handlers `extCreate` and `extReceipt` might access an incomplete state of the `agreements` variable. To avoid this, both functions introduce locks.

The example introduces property functions as well. The `retractedAny` function checks that any of the systems has an agreement with the given ID retracted. Another function `retractedAll` asserts that all systems have the agreement retracted. Using these functions, we can build an LTL formula that includes assertion-based properties:

```
[](retractedAny("doc1")-><>retractedAll("doc1"))
```

The formula specifies that every time the document `doc1` is retracted in any of the systems it will eventually be retracted in all other systems as well. Running the model checker using this property in the WAFL framework produces the following output:

```
Maude>(ltlcheck[](retractedAny("doc1")-><>retractedAll("doc1")))
Valid
```

This indicates that the architecture fulfils the specified LTL formula. On the other hand, if we comment out the `lock` and `unlock` functions in system A, and run the same command, we get the output shown in Listing 4.5. It contains all necessary information about the configuration of the loop states, as well as all states that led to the invalid loop. The information includes all acquired locks, persistent variables, and executing processes to infer the failing scenario. The full output is too bulky to include, so some details in the listing were omitted and replaced by dots (...).

Listing 4.5: Output of LTL checking with counterexample.

```
--- Previous states...
-->
END
With loop:
--- connections and queues...
service "centre" {
    ...
}
service "systemA" {
        code {...}
        routes {...}
        locks(empty)
        persistent("agreements",map(
            "doc1" |-> map(
                ("correspondents" |-> list(strVal(
                "companyA")),
                "id" |-> strVal("doc1"),
                "sender" |-> strVal("newCompany"),
                "state" |->
                strVal("sent"))))
            )
        persistent("retracted",list(strVal("doc1")))
}
service "systemB" {
        ...
}
-->
END
```

Conclusion

Formal verification of web service architecture is not a widely accepted practice among software engineers. One reason is that formal verification requires knowledge and experience in formal methods. To make formal verification more accessible to developers we introduce WAFL - a formal verification framework that allows to model web services using an imperative programming language. The source code is accessible at a GitHub webpage[1]. The framework was developed as an extension to the Maude language using provided tools to add new commands with custom syntax to Maude. The framework supports the definition of web service architecture by specifying separate servers with isolated environments and executable code. We also introduce PLIXE - the programming language the code is written in. The communication between services is either a synchronous HTTP request or an asynchronous message queueing. The verification of the architecture is performed using assertion-based properties. The properties call a PLIXE

[1] https://github.com/zholzhas/wafl

program that makes assertions about architecture configuration to define whether the property holds.

Limitations and Future Work

Currently, the WAFL framework has several limitations restricting its usage.

Firstly, the developed framework does not construct an optimal state space of web service architectures. Due to the limitations of rewrite logic, messages instantiated inside a process are rewritten in several steps. The state space blows up exponentially in the number of excessive steps and reduces the model checking time. Therefore, verification time for middle to large models is unreasonably high. Thus, there is a perspective for optimizing the framework. The most important step is to develop such rewrite rules that will process messages from PLIXE in one step. It is not a trivial task because normally, when we have several environments like PLIXE and WAFL and we need to transfer messages between them, it takes at least three steps:

- generate a message in PLIXE,
- move it from PLIXE environment to WAFL, and
- process the message in WAFL.

The reverse process then sends information to a PLIXE program. Therefore, one should use more optimized data structures for such communication. Another option is to transfer the framework to the *K*-framework since it was designed to define programming languages.

It is also difficult to navigate the model checking result with a counterexample. The output is too bulky and has a lot of information unrelated to the current trace. This can be solved by identifying the rewrite rule that caused a transition in the trace and displaying only objects participating in the rewrite rule.

Another limitation of WAFL is the potential to write PLIXE programs with infinite execution. Although this thesis did not study the computational power of PLIXE, the initial aim of PLIXE implies that the language should be Turing-complete to model orchestration engines. Hence, possibly infinite programs can break the behaviour of the framework, and we can not identify such programs. One solution is to introduce a user-defined limit on the number of executed lines of code. When this limit is broken, a user will see a warning that the program could have entered an infinite loop.

Appendix A. Example WAFL Specification

Listing A.1: Example Specification for Document Exchange.

```
(service systemA {
    persistent agreements = {};
    persistent retracted = [] ;
    routes {
        "/ext/create" -> extCreate
        "/ext/receipt" -> extReceipt
    }
    code {
        function extCreate(request) {
            lock(request . id);
            agreements = getPersistent("agreements") ;
            agreement = request ;
            correspondents = [] ;
            i = 0 ;
            ret = getPersistent("retracted") ;
            while (i < len(ret)) {
                if (ret [ i ] == agreement . id) {
                    agreement . state = "retracted" ;
                }
                i = i + 1 ;
            }
            agreements[ agreement . id ] = agreement ;
            setPersistent("agreements", agreements) ;
            unlock(request . id);
            respond(agreement) ;
        }
        function extReceipt(request) {
            lock(request . id);
            agreements = getPersistent("agreements") ;
            id = request . id ;
            if (keyExists(agreements, id)) {
                agreement = agreements[id] ;
                agreement . state = "retracted" ;
                agreements[agreement . id] = agreement ;
                setPersistent("agreements", agreements) ;
            }
            else {
                ret = getPersistent("retracted");
                ret = append(ret, id);
                setPersistent("retracted", ret);
            }
            unlock(request . id);
            respond("ok");
        }
    }
}
service centre {
    persistent correspondents = {
        companyA : "systemA",
```

```
        newCompany : "systemB"
    };
    listen {
        "newDocument" -> newDocument
        "receipt" -> receipt
    }
    code {
        function newDocument(request) {
            systems = getPersistent("correspondents");
            i = 0 ;
            while ( i < len(request . correspondents) ) {
                request(systems[ request . correspondents [ i ] ],
                    "/ext/create", request ) ;
                i = i + 1 ;
            }
        }
        function receipt(request) {
            systems = getPersistent("correspondents");
            i = 0 ;
            while ( i < len(request . correspondents) ) {
                request(systems[ request . correspondents [ i ] ],
                    "/ext/receipt", request ) ;
                i = i + 1 ;
            }
        }
    }
}
service systemB {
    persistent agreements = {};
    routes {
        "/create" -> create
        "/send" -> send
        "/retract" -> retract
    }
    code {
        function create(request) {
            agreements = getPersistent("agreements") ;
            agreement = request . agreement ;
            agreement . state = "created" ;
            agreements[agreement . id] = agreement ;
            setPersistent("agreements", agreements) ;
            respond(agreement) ;
        }
        function send(request) {
            agreements = getPersistent("agreements") ;
            id = request . id ;
            agreement = agreements[id] ;
            agreement . state = "sent" ;
            agreements[agreement . id] = agreement ;
            setPersistent("agreements", agreements) ;
```

```
            agreement . sender = "newCompany" ;
            message("newDocument", agreement);
            respond("ok");
        }
        function retract(request) {
            agreements = getPersistent("agreements") ;
            id = request . id ;
            agreement = agreements[id] ;
            agreement . state = "retracted" ;
            agreements[agreement . id] = agreement ;
            setPersistent("agreements", agreements) ;
            message("receipt", { type : "retract", id : id,
                correspondents : agreement . correspondents });
            respond("ok");
        }
    }
}
init {
    request("systemB", "/create", {
        agreement : {
            id : "doc1",
            correspondents : [ "companyA" ]
        }
    });
    request("systemB", "/send", {
        id : "doc1"
    }) ;
    request("systemB", "/retract", {
        id : "doc1"
    }) ;
}
props {
    function retractedAny(id) {
        serviceNames = [ "systemB", "systemA" ];
        i = 0 ;
        retracted = false ;
        while (i < len(serviceNames)) {
            agreements = services[serviceNames[i]]
                . persistents . agreements ;
            if (keyExists(agreements, id)) {
                retracted = retracted ||
                    agreements[id] . state == "retracted" ;
            }
            i = i + 1 ;
        }
        assert(retracted) ;
    }
    function retractedAll(id) {
        serviceNames = [ "systemB", "systemA" ];
        i = 0 ;
```

```
        retracted = true ;
        while (i < len(serviceNames)) {
            agreements = services[serviceNames[i]]
                . persistents . agreements ;
            agreement = agreements[id] ;
            retracted = retracted
                && agreement . state == "retracted" ;
            i = i + 1 ;
        }
        assert(retracted) ;
    }
})
```