DOCTORAL THESIS

Automated Replication of Tuple Spaces via Static Analysis and Transformation of Go Programs

PHD PROGRAM IN COMPUTER SCIENCE: XXXII CYCLE

Supervisors:

Author:

Prof. Rocco DE NICOLA
rocco.denicola@imtlucca.it

Aline UWIMBABAZI
aline.uwimbabazi@gssi.it

Dr. Omar INVERSO
omar.inverso@gssi.it

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy at the

GSSI Gran Sasso Science Institute
Viale Francesco Crispi, 7 - 67100 L'Aquila - Italy

Declaration of Authorship

I, Aline Uwimbabazi, declare that this thesis entitled «Automated Replication of Tuple Spaces via Static Analysis and Transformation of Go Programs» and the work presented in it are my own.

I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- This thesis has been composed by myself and the presented work is my own under the guidance of my supervisors Prof. Rocco De Nicola and Dr. Omar Inverso for a research degree at Gran Sasso Science Institute.
- Chapters 3, 4 and 5 are based on [1] Aline Uwimbabazi, Omar Inverso and Rocco De Nicola. *Automated Replication of Tuple Spaces via Static Analysis*. In Proc. of the International Conference on Fundamentals of Software Engineering, pages 21-36, 2021. This paper received the Best Research Paper Award at FSEN 2021.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed: ____________________

Date: July 2022

Abstract

Coordination languages for the tuple spaces can offer significant advantages in the specification and implementation of distributed systems. Replication can further improve the performance or robustness of these systems. A possible approach has been proposed to extend the coordination languages with replica-aware primitives. However, this approach relies on the ability of the programmer to specify and coordinate the replication of tuple spaces in order to preserve the desired consistency. Manual exploitation of the offered replication primitives is expensive and extremely hard. We present a technique that combines static analysis and program transformation for automated replication of tuple spaces.

As a starting point, we take goSpace coordination language, an implementation of Klaim in Go programming language, and extend it with the replica-aware tuple manipulation primitives to obtain what we call RepligoSpaces. We also present how a Go program that uses goSpace can be transformed into a program that uses RepligoSpaces while preserving the desired consistency level. This transformation is optimized by statically analysing the data access patterns. Custom static analyses may be plugged in relatively easily in our prototype implementation. We see this as a first step towards developing an integrated framework to experiment with data replication in distributed systems with tuple spaces. Thus, supporting designers for quick prototyping and evaluation of replication schemes. A series of experiments are performed on a case study to show how our approach can be used to design and test replication schemes.

Keywords: Coordination Languages, Tuple Spaces, Static Analysis, Automated Replication, Program Transformation, Golang.

Acknowledgements

The research work presented in this thesis would not have been possible without the guidance, patience, and support of my supervisors, Prof. Rocco De Nicola and Dr. Omar Inverso, who advised me throughout the course of my Ph.D. studies. Thank you.

There are people who live for helping in the most painful, desperate, and terrible situations, among them Dr. Catia Trubiani and F. Tryphon Kalimira. For that, I am grateful.

I would like to thank the dissertation committee members for having agreed to examine my research work. I am indebted to Prof. Willem Visser (Stellenbosch University, SA) for contributing immensely to my academic path. I am grateful to African Institute for Mathematical Sciences (AIMS-Rwanda) for giving me a chance to give back as a tutor.

Special thanks go to my family. My mother, father and siblings regularly encouraged me throughout my studies. I love you all. May God bless and protect you.

To all my friends, thanks for the support, for checking on me from time to time. Thank you for listening to me when things did not go according to plan. May God bless you.

My gratitude goes to Luca Di Stefano, Alessandro, Noah, Patrizia Pulsoni, and to the members of the Department of Computer Science and the administration at Gran Sasso Science Institute for their support and help. May God bless you.

None of what I accomplished matters without acknowledging Almighty God, who grants me all that I have. Without Him, life is nothing. All praise to you God.

Dedication

To my Almighty God for the gift of life and patience. All praise to you God.

To my lovely parents and person who will make a photo with me. May God bless you.

Contents

Abstract ii

Acknowledgements iii

Dedication iv

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Context and Approach . . . 1
- 1.2 Contributions . . . 4
- 1.3 Structure of the Thesis . . . 5

2 Background and Related Works 6
- 2.1 Introduction to Coordination . . . 6
- 2.2 The Linda Coordination Model . . . 7
  - 2.2.1 Operations . . . 8
  - 2.2.2 Example . . . 9
- 2.3 The Klaim Coordination Language . . . 10
  - 2.3.1 Klaim Syntax . . . 11
  - 2.3.2 Operational Semantics of Klaim . . . 14
  - 2.3.3 Example . . . 17
- 2.4 Data Replication . . . 18
- 2.5 The RepliKlaim Coordination Language . . . 20
  - 2.5.1 RepliKlaim Syntax . . . 20
  - 2.5.2 Operational Semantics of RepliKlaim . . . 23
  - 2.5.3 Example . . . 24
- 2.6 Related Work on Coordination Languages with Replication . . . 25
- 2.7 Programming with Spaces (pSpaces) . . . 28
  - 2.7.1 pSpaces Syntax . . . 29
  - 2.7.2 GoSpace Syntax . . . 30
- 2.8 The Go Programming Language . . . 32

2.8.1 Concurrency 32
2.8.2 Examples 33
2.9 Replication Models and Techniques 35
2.9.1 Replication models 35
2.9.2 Replication Techniques 36
2.10 Consistency Levels for Replicated Data 37
2.11 Concepts of Static Program Analysis 38
2.11.1 Static Analysis Techniques 40
2.11.2 Abstract Syntax Trees 42
2.11.3 Related Work on Static Analysis 45
2.12 Program Transformation 47
2.12.1 Related Work on Program Transformation 48

3 RepligoSpaces: goSpace with Replicas 50
3.1 Tuples and Templates 50
3.2 RepligoSpaces Syntax 51
3.3 Informal Semantics of RepligoSpaces 53
3.4 Prototype Implementation 54

4 Static Analysis and Program Transformation for Replication 60
4.1 Automated Replication of Tuple Spaces 61
4.1.1 Structures of Input and Output Programs 62
4.1.2 Overapproximating the Sets of Target Spaces 65
4.1.3 Program Transformation 67
4.2 Illustrative Example 73

5 Evaluation of Implementation 78
5.1 Case study 78
5.2 Experimental Setup 79
5.3 Experimental Results 80

6 Conclusions and Future Work 84
6.1 Conclusions 84
6.2 Future Work 85

List of Figures

1.1 Our overall approach to automated replication of tuple spaces 3

2.1 An example of tuple space 10
2.2 An example of tuple spaces expressed in Klaim 17
2.3 The phases of a functional replication model [2] 19
2.4 An example of tuple space expressed in RepliKlaim 25
2.5 An example of abstract syntax tree 44
2.6 An example of AST for the source code in Listing 2.7 45

4.1 Static analysis and source transformation for automated replication . . . 61
4.2 Example of transformation of global variables in Listing 4.4 72
4.3 No replication 76
4.4 Example replication strategies 76

5.1 Non-local read or write operations with and without replication 82
5.2 Execution times (ms) of non-local read or write operations with and without replication 83

6.1 Program transformation for the code shown in Listing 6.2 88

List of Tables

2.1 Syntax of Klaim. Adapted from [3] 13
2.2 Tuple Evaluation Function. Adapted from [3] 13
2.3 Pattern-matching predicates of Klaim [3, 4] 15
2.4 Structural operational semantics rules of Klaim. Adapted from [5] . . . 16
2.5 Syntax of RepliKlaim. Adapted from [6] 21
2.6 Basic operations on tuple spaces in Klaim and RepliKlaim 22
2.7 Operational semantics rules of RepliKlaim [6] 23
2.8 Syntax of goSpace [7] 31

3.1 Basic operations on tuple spaces in goSpace and RepligoSpaces 52

4.1 Transformation Rules from goSpace to RepligoSpaces 69

5.1 Configuration of the system for non-local read or write operations . . . 79
5.2 Configuration of the system for computing execution times of non-local read or write operations 80

Chapter 1

Introduction

1.1 Context and Approach

Coordination is essential in distributed systems and in managing the interactions and dependencies between the entities of systems. When designing a distributed system, adopting a suitable coordination model can be of fundamental importance. The use of coordination models started in the mid-1985s specifically with the use of the tuple-space model, also known as Linda [8]. The Linda coordination model was originally proposed for implementing parallel applications. Multiple, and distributed tuple spaces systems[1] have been introduced later [9–11] to improve performance and scalability. Known implementations of tuple-spaces based coordination models and languages can be seen in the industry (e.g., IBM's TSpaces [12] and Sun Microsystems' JavaSpaces [13]); and in several academic projects (e.g., Klaim [3], PageSpace [14], Lime [15, 16], MARS [17], TuCSoN [18], and TOTA [19]). Recent research works have extended Klaim [20] to practically program Klaim-based applications and to improve the efficiency of tuple spaces implementations [21]. Moreover, tuple spaces implementations have been used to develop the chemical-inspired models that can mimick chemical systems in terms of reactions and diffusion rules [22]. Thus, a distributed shared space for supporting pervasive service ecosystems has been implemented. The discussion in this thesis is specifically based on the concepts of coordination languages extended from Linda with the notion of location (e.g., Klaim) and enriched with the replica-aware coordination primitives (e.g., RepliKlaim [6]).

[1] Systems in which the tuples of the same space can be stored on different servers.

As we mentioned, we focused on tuple spaces, an associative shared memory implementing the Linda coordination model, which is extended with the notion of localities for accessing data and enriched with the replica-aware coordination primitives for replicating data. A tuple space is a collection (formally a multiset) of tuples or data items.

To facilitate the specification of inter-process communication patterns, some coordination models and languages provide explicit data access primitives. Those coordination models and languages which were made for coordinating and synchronizing agents (or processes) activities, have evolved into the designing and implementations of parallel and distributed systems. In Linda coordination model, processes can concurrently access an associative data store referred to as tuple space, where tuples, i.e., sequences of typed data atoms, can be stored or fetched from. Tuples can be picked up from tuple spaces by means of a pattern-matching mechanism. Processes synchronize and communicate in this way. Klaim [3] coordination language extends (Linda) this approach to multiple tuple spaces with explicit localities for greater flexibility.

On large data-intensive distributed systems, techniques to optimise data distribution and locality may significantly improve performance. One such technique, replication, fits very well within the coordination languages framework. The idea is quite simple: on a store operation, tuples are deployed to a set of target spaces rather than just to a single one. This increases locality, and thus reduces access latency, but brings along the problem of consistency: once a specific copy of a given tuple is modified, how are the remaining copies to be affected?

RepliKlaim [6] addresses such tension between performance and consistency by extending Klaim's operational semantics with replica-aware data manipulation primitives. The programmer can use these primitives to control the distribution of the data as well as the consistency levels. Yet, doing so requires programming ingenuity to specify and coordinate the replicas. Such manual reasoning can be particularly cumbersome because of process interleaving, and hardly feasible in the presence of a large number of complex processes. For the same reasons, evaluating different replication strategies with respect to the intended performance-reliability trade-off can be rather tricky.

In this thesis, we address the above shortcomings by proposing an experimental approach to support the design of replication policies in distributed systems that use tuple spaces for process coordination and data storage.

More concretely, we present an automated technique to transform the specifications of a given distributed system into an equivalent version where the tuples are replicated. The overall approach is sketched in Figure 1.1.

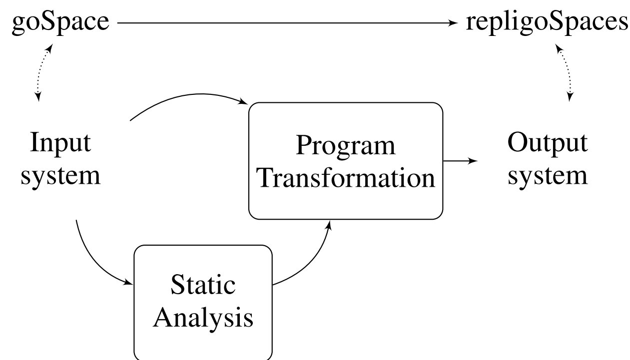


FIGURE 1.1: Our overall approach to automated replication of tuple spaces

The initial system of interest is modelled as a concurrent Go [23] program. The choice of using Go programming language has been motivated by the need to use its concurrency model developed based on the traditional process calculi, such as Milner's Calculus of Communicating Systems [24] and Hoare's Communicating Sequential Processes [25]. We only consider closed systems, i.e., systems where the number of processes is fixed upfront. The behavior of each system component of the system is defined by a separate thread of the program. Coordination takes place via the goSpace coordination language [26], a recent Go implementation of Klaim, specifically designed to program distributed applications with tuple spaces.

To attain automated replication, we work (1) at the programming interface level, by implementing extended primitives for replica-aware manipulation of tuples, and (2) at the source code level, by combining static analysis and program transformation to transform the initial model into replicated version.

We are specifically interested in exploiting replication to improve the performance of distributed systems. Taking inspiration from the way Klaim's operational semantics rules were extended in RepliKlaim coordination language, we have extended goSpace's coordination primitives with replica-aware coordination primitives to obtain what we call RepligoSpaces. The extended primitives make it possible to target multiple tuple spaces for a single store operation. In addition, an embedded tracking mechanism allows to consistently remove the replicated data as needed. At this point one could immediately obtain full consistency by naively using the extended primitives to replicate

every tuple to every shared space in the system. This could be automatically obtained via program transformation, by replacing the tuple manipulation operations with their replica-aware versions, but would likely result in unnecessary replication overhead.

For this reason, we work at the source code level; between the replica-aware data-handling layer and the program transformation part, we introduce a lightweight static analysis pass to refine the target spaces for each store operation. The key idea of our lightweight static analysis technique consists in determining the set of processes that can potentially perform a subsequent read operation on a stored tuple. We identify such processes by looking at the patterns used in the input operations occurring in the corresponding process specification functions, approximating the actual pattern matching mechanism of the normal tuple manipulation routines. In practice, given on the one hand an output operation and on the other hand an input operation, we check for a potential match between the tuple being stored and the given search pattern.

This simple workflow is easily extensible, given the modularity between the data-handling layer, the program transformation schema, and the static analysis procedure. Different static analysis techniques may be plugged in with a relatively limited effort. At the same time, alternative consistency models can be quickly prototyped by altering the existing replica-aware primitives. We see this as a first step towards developing an integrated framework to experiment with data replication in distributed systems with tuple spaces. This would allow, for instance, to evaluate under different consistency levels many interesting classes of systems, such as models of hardware cache or complex interaction models, where replication is heavily used and performance is particularly sensitive to variations in the data distribution policy. An experimental evaluation that shows how our approach can be used to design and test the replication mechanisms is performed on a scenario of a distributed lookup case study. In fact, our results include a number of experiments aimed at providing some performance related criteria for the improved latency, and exhibited good results that improved performance of distributed systems.

1.2 Contributions

The contributions of this research work are the following:

1. RepligoSpaces, a replica-aware extension to goSpace based on RepliKlaim's operational semantics rules.

2. A prototype framework that integrates static analysis and program transformation for automatically replicating tuple spaces. In particular, we have developed

   - a static analyzer that targets the data access patterns for precisely defining the sets of target spaces for replication;
   - a prototype replicator that transforms the initial model into the equivalent replicated version by using the target spaces computed via static analysis.

3. An experimental evaluation that shows how our approach can be used to design and test the replication mechanisms.

Our tool has been released as an open-source prototype, free for the community to use.

1.3 Structure of the Thesis

The rest of the thesis is organized as follows:

- Chapter 2 provides background knowledge related to the research pursued in this thesis, including a brief review of research work related to ours. We also provide some concepts of Go.
- Chapter 3 presents our first contribution, the tuple spaces-based coordination language, called RepligoSpaces, which extends goSpace with replica-aware coordination primitives. Specifically, it presents the syntax, operations, prototype implementations of the extended primitives, and informal semantics of RepligoSpaces.
- Chapter 4 presents our second contribution, our prototype framework for the automated replication of tuple spaces, that integrates the tuple spaces-based coordination languages, static analysis, and program transformation to improve the performance of a given distributed system that uses tuple spaces for data storage.
- Chapter 5 presents a usage scenario of our technique on a case study used for evaluating our implementations.
- Chapter 6 contains some concluding remarks and reflects on possible directions for future work.

Chapter 2

Background and Related Works

This chapter presents the concepts that are at the foundation of our research work and will serve in subsequent chapters. More specifically, it presents the main coordination languages used throughout the thesis, and a brief description of the Go programming language used to implement our prototypes. Furthermore, it describes data replication approaches and techniques as well as the consistency levels for replicated data. It proceeds with the description of some basic concepts of static analysis techniques and program transformation. It also presents a brief review of research work related to ours.

2.1 Introduction to Coordination

The term *«coordination»* has an instinctive meaning in our life, it can be defined as the process of managing activities and assuring that all of them progress in the good ways. Research in the coordination domain has 3 decades of history and from that period, several techniques and systems have been proposed and implemented. Omicini et al. [27] defines the term coordination in the context of distributed computing as:«the management of interactions and dependencies between the computational entities of a system.» Gelertner and Carriero [11] defines coordination in the context of coordination models and languages:«the coordination model is the glue that binds the separate activities into an ensemble.»

The history of coordination model and language started in the 1985′s when Gelernter and Carriero introduced *Linda* [8] as coordination language for implementing distributed applications. Since then, there has been growing interests in the implementations of several coordination languages. In this chapter, however, we only introduce the main coordination languages that are relevant to the work presented in this thesis.

2.2 The Linda Coordination Model

Linda [8, 9] is a coordination language for implementing the parallel and distributed systems. It provides a generative communication mechanism based on a logically shared memory space, called *tuple space*. Linda coordination model consists of: tuples, tuple space, agents (or processes) and communication primitives (or operations).

A tuple space is a collection of tuples that represent data items and are concurrently accessed from the processes. *A tuple* is a finite sequence of actual fields (e.g., expressions, values, processes) or formal fields (i.e.,variables). Tuples that contain variables are called *templates* (or *patterns)*. The formal fields define the patterns in the form of variables and their values are not known. The tuples to be added in a tuple space can only have actual fields, whereas templates are similar to a tuple, but can contain both actual and formal fields. `("Journalist","Sport",2018)` is an example of the tuple, while `("Journalist",!category,!year)` is a template with two variables `category` and `year`. The variables `category` and `year` are denoted by a leading symbol `!`.

In Linda and its variants, the processes are like the threads of execution and they can act as consumer or a producer of tuples. Communication primitives consist of the operations performed by such processes, such as writing, reading and removing tuples from a tuple space. Such primitives allow processes to share data by placing `(out)` tuples to a tuple space, reading `(rd)` and consuming them `(in)` from a tuple space. The tuples are accessed by their content rather than their address via pattern matching algorithm, by using templates or patterns.

The pattern matching algorithm matches the tuple `t` and template `P` if:

- `P` and `t` have the same number of fields and all the pairs of fields at the same position in `P` and `t` also do match;

- Corresponding fields of `P` and `t` have the same types;
- When comparing `t` and `P`, the values of the two fields are exactly the same.

Consider the following examples that illustrate the above pattern matching algorithm.

- `("Journalist",category,year)` matches `("Journalist","sports",2020)`. Because the two tuples have the same number of fields and all pairs of fields at the same positions also do match.
- `("Journalist",category,21)` does not match `("Journalist","sports",20)`. Although the two tuples have the same types and number of fields, their values are not exactly the same (i.e., `21` vs `20`).
- `("Journalist","status")` does not much `("Journalist",2020)`. Although the two tuples have the same number of fields, the types of their last fields are not the same (i.e., `string` vs `integer`).

2.2.1 Operations

Linda provides several operations, described as follows:

- `out(tuple)` - inserts a tuple into the tuple space.
- `in(template)` - selects and removes a tuple matching the given template from the tuple space. If a tuple cannot be found, the process that executes the operation blocks.
- `rd(template)` - reads a copy of a matching tuple from a tuple space, blocks if the tuple cannot be found.
- `eval(tuple)` - spawns a new process to evaluate the tuple and inserts the result of the evaluation into a tuple space.

Two of them `(in(template)` and `rd(template)`) consume the tuples and are blocking operations as they keep waiting for a tuple until it is found in a tuple space. The `out` and `eval` operations are non-blocking operations.

Note that operation `eval(tuple)` differs from `out(tuple)` operation because it first adds a tuple to a tuple space, then creates a new concurrent process for evaluating that tuple.

In addition to the above operations, Linda coordination model has been extended [28] to other versions that presented some forms of non-blocking versions of remove operations, called `inp` and `readp`. They behave like `in` and `read` operations, but, if no matching tuple is found, they directly return false (or null) if no matching tuples are present into a space instead of suspending the execution of a process. For example, the `readp` operation is used to check whether a tuple is stored in a tuple space or not.

Despite Linda has a desirable simplicity and is a powerful model for implementing parallel applications, it has some weaknesses. For example, the original Linda model only coordinated activities of processes on a single tuple space and does not scale well to the large systems [29]. Moreover, the original Linda coordination primitives were not completely adequate for programming distributed systems [3, 20]. Many researchers starting from the basic constructs of Linda have extended it and have implemented several models and languages based on it [15, 30–34].

2.2.2 Example

In this example, we consider a tuple space that contains different structured values and processes performing some operations. Tuples to be written in the tuple space can only have actual fields, whereas templates can contain both actual and formal fields. For example, tuples ⟨`"journalist","sport",2020`⟩ and ⟨`"journalist",id,27.3"`⟩ are produced by `out("journalist","sport",2020)` and `out("journalist",id, 27.3)` operations, and they can be read by `rd("journalist","category",year)` operation after pattern-matching. As shown in Figure 2.1, the given pattern contains one actual field and two formal fields.

The reading process can check all tuples in the tuple space consisting of three elements containing the strings “`journalist`” and “`category`” in the first two fields and assigns the third value of one of the matching tuples to variable `year` which is assigned to 2020 tuple’s field. However, the tuple `("journalist",id, 27.3)` does not match the given template, although they have the same number of fields, the types of their last fields are different (i.e., int vs string).

The withdrawing process performs `in("journalist","sport",2021)` operation to remove tuples that have the two fields of type string `("journalist","sport")`. In this case, the tuple `("journalist","sport",2021)` is not removed from the tuple space as the last fields are not exactly the same. A new process can be spawned with an operation `eval(...)`. Figure 2.1 presents an example of a tuple space.

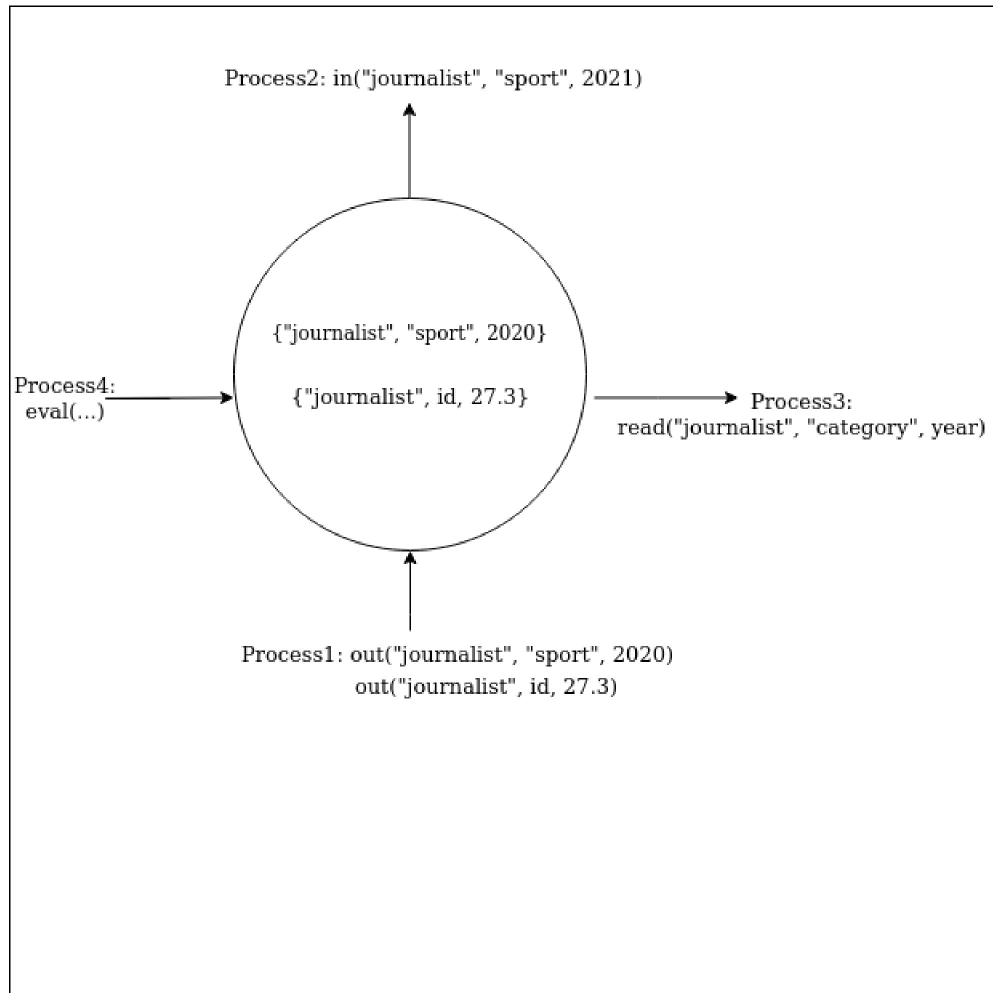


FIGURE 2.1: An example of tuple space

2.3 The Klaim Coordination Language

Klaim (*Kernel Language for Agents Interaction and Mobility*) [3, 20] is a coordination language that is used to describe distributed systems where both data and processes can be moved from one computing environment to another.

The coordination primitives and operational semantics rules of Klaim are designed and implemented based on how Linda's operations [8, 9] and the concepts of process algebra [24, 25] are designed and implemented. To this end, Klaim extended Linda by allowing the manipulation of tuples on multiple tuple spaces and by offering communication primitives with explicit localities. In Klaim, the processes and tuple spaces can be located on different *nodes*, and localities represent unique identifiers for such nodes. Klaim processes can run concurrently, at the same node or at different nodes.

Explicit localities allow to distribute and retrieve data to and from the localities, and to structure the tuple space. In fact, the data manipulation operations of Klaim are based on the standard Linda primitives for tuple spaces but, in addition, they explicitly indicate the target tuple space as a reference $(@\ell)$ to the intended locality. In Klaim, processes like any other data items, can be transferred from one locality to another and can be executed at any locality.

Motivated by the success and popularity that Klaim coordination language gained, many researchers starting from the constructs of Klaim coordination language have extended it with enriched coordination primitives and control flow constructs to implement X-Klaim (eXtended Klaim) [20, 35, 36]. X-Klaim is a coordination language developed to program distributed applications which consist of software components interacting through multiple distributed tuple spaces. Other models and languages based on Klaim are developed [4, 37–44].

2.3.1 Klaim Syntax

Klaim consists of a core Linda with multiple tuple spaces and operators for building processes. Klaim's specification is a network *N*, i.e., a possibly empty set of nodes or components. In Klaim, a component `L::[K, P]` consists of a locality name denoted by ℓ, a data repository *K* (i.e., set of data items) and multiple processes `P`. *A locality* is a computational entity (sometimes referred to as *sites* or *nodes* of a network) where components or processes reside or are being executed. In Klaim, each node has both the physical and logical locality, which are linked by the *allocation environment*. It enables a (partial) mapping from logical to physical localities, i.e., by enabling processes from one node to access only the nodes included in its allocation environment. In this case, each node in a network has a single and unique locality that is known to all other nodes. Processes are built by means of operators that are borrowed from Milner's calculus

of communicating systems [24]. The syntax of Klaim is reported in Table 2.1 and described as follows [3]:

- *nil* represents a process that cannot perform any action.
- `A.P` represents a process that first executes the action a and then behaves like `P`.
- $P_1 + P_2$ represents the non-deterministic choice of processes P_1 and P_2.
- $P_1 \mid P_2$ represents the parallel composition of two processes.
- `X` is a process variable.

In the definition of process invocation $A \langle \tilde{P}, \tilde{\ell}, \tilde{e} \rangle$, A denotes a process name; $\tilde{P}$, $\tilde{\ell}$ and $\tilde{e}$ represent a list of processes, a list of localities and a list of expressions which are passed when invoking the process (in the same way that parameters are used when calling a function). Process variables play an important role during the communication. More specifically, when a process is moved from one locality to another for the purpose of execution, the variables can dynamically be assigned to the processes. Assume that a process is added to a tuple space when it is selected; the process variable is also assigned to this process for execution. The arguments associated to the tuples and templates are defined as follows.

- The set of localities L, is ranged over by $L = \{\ell_1, \ell_2, \ell_3, \ldots\}$,
- The set of basic values $\mu = \{u, v, w, \ldots\}$,
- The set of value variables $!\nu = \{!x, !y, !z, \ldots\}$,
- $\tau \subseteq (\mu \cup \nu)^*$ is a set of tuples ranged over by $\tau = \{t, t', \ldots\}$,
- `Exp(e)` is the category of value expressions, which are built from the values and value variables by using a set of operators.

$N ::= \mathtt{0} \mid \ell::[K,P] \mid N \parallel N$	(networks)
$K ::= \mathtt{0} \mid et \mid K,K$	(repositories)
$P ::= \mathtt{nil}$	(null process)
$\mid A.P$	(action prefixing)
$\mid P_1 + P_2$	(choice)
$\mid P_1 \mid P_2$	(parallel composition)
$\mid X$	(process variable)
$\mid A \langle \tilde{P}, \tilde{\ell}, \tilde{e} \rangle$	(process invocation)
$t ::= e \mid P \mid \ell \mid !u \mid t_1, t_2$	(tuples)
$T ::= e \mid P \mid \ell \mid !x \mid !X \mid T_1, T_2$	(templates)
$A ::= \mathtt{out(t)@}\ell \mid \mathtt{in(T)@}\ell$	(actions)
$\mid \mathtt{read(T)@}\ell \mid \mathtt{eval(P)@}\ell$	

TABLE 2.1: Syntax of Klaim. Adapted from [3]

A data repository K is a collection of evaluated tuples according to the tuple evaluation function mechanism $\tau[[\ ]]_\rho$ reported in Table 2.2. For example, if t is a tuple and ρ an environment, the evaluation of t within ρ is defined as $\tau[[t]]_\rho$, the $\varepsilon[[e]]$ is applied to the expressions, where $e \in \varepsilon$ (i.e., ε is the set of expressions) as shown in Table 2.2.

$$\tau\ [[e]]_\rho = \varepsilon\ [[e]] \qquad \tau\ [[P]]_\rho = P\{\rho\}$$

$$\tau\ [[\ell]]_\rho = \rho\{\ell\} \qquad \tau\ [[!x]]_\rho = !x$$

$$\tau\ [[t,t^{'}]]_\rho = \tau[[t]]_\rho,\ \tau[[t^{'}]]_\rho$$

TABLE 2.2: Tuple Evaluation Function. Adapted from [3]

In Klaim, the processes interact directly with tuple spaces to store tuples to or retrieve tuples from the localities. Klaim provides four basic primitives to manipulate tuple spaces. Two (non-blocking) operations `out(t)`@ℓ and `eval(t)`@ℓ permit tuples to be

added to a tuple space. Two (blocking) operations `in(`T`)`$@\ell$ and `read(`T`)`$@\ell$ are used to access and modify a tuple space. These operations are described as follows [3]:

- `out(`t`)`$@\ell$: non-blocking operation that adds a tuple t to the tuple space located at location ℓ.
- `read(`T`)`$@\ell$: selects via the pattern matching one of the tuples at locality ℓ that matches template `T`; this operation blocks until a tuple matching T is found.
- `in(`T`)`$@\ell$: removes a tuple matching the pattern T from the location ℓ. The process executing the `in(`T`)` operation blocks until a tuple matching the pattern T is found.
- `eval(`P`)`$@\ell$: offers the possibility of code mobility by spawning a process `P` at locality ℓ where it will be evaluated.

Note that the argument `T` is often called a pattern or template, and its fields contain either actual or formal fields. In the rest of this work, we will use `T` to indicate a pattern (or a template) and `t` to indicate a tuple.

2.3.2 Operational Semantics of Klaim

The operational semantics of Klaim provides the guidelines of its implementation. They explicitly associate the pattern matching predicates (see Table 2.3) with each piece of semantics in a formal way. More specifically, they define the computational steps to be performed: the evaluation of tuples, the pattern matching rules for retrieving a tuple from a tuple space, the operations useful for processes interactions.

The pattern matching rules of Klaim presented in Table 2.3 are used to select a tuple that matches any given pattern from a tuple space. These rules are also exploited in the operational semantics when the processes perform `in` and `rd` operations. The pattern matching rules work if: an evaluated tuple matches against an evaluated pattern if both have the same number of fields and their corresponding fields also do match; the corresponding processes, two values (localities) are identical; the locations or processes have the same types, while the formal field $!x$ and the corresponding values (names) have also the same types.

The rules for defining the pattern matching function are given in Table 2.3 and the tuple evaluation function is reported in Table 2.2.

A pattern matching function, `match(.,.)`, is used to verify a compliance of a tuple with respect a template and to associate values (i.e. names and processes) to variables bound by the template. Intuitively, an evaluated tuple matches against a template if both have the same number of fields, and corresponding fields match (where a bound name matches any value, while two names match only if they are identical).

We assume that "$\circ$" denotes a substitution composition and "ε" denotes an empty substitution. A successful matching returns a substitution function associating the variables that are contained in the formal fields of the given template with the values that are contained in the corresponding actual fields of the located tuple.

We used a notation P_σ where $\sigma = match(T, et)$, to indicate the substitution of `T` for `et` in `P`. Moreover, we assume that P_σ yields a process written according to the operational semantic rules reported in Table 2.4. Klaim's pattern matching predicates differs from Linda's original one in that it does not allow the tuples to contain formal fields [45]. The definition of function match is given by the following laws:

$$match(v, v) = \varepsilon \qquad match(!x, v) = [v/x]$$

$$match(\ell, \ell) = \varepsilon \qquad match(!u, \ell) = [\ell/u]$$

$$match(!X, P) = [P/X] \qquad \frac{match(T_i, et_i) = \sigma_i \quad (i = 1, 2)}{match(T_1, T_2,\ \ et_1, et_2) = \sigma_1 \circ \sigma_2}$$

$$\frac{match(T_1, et_1) = \sigma_1 \quad match(T_2, et_2) = \sigma_2}{match(T_1, T_2,\ \ et_1, et_2) = \sigma_1 \circ \sigma_2}$$

TABLE 2.3: Pattern-matching predicates of Klaim [3, 4]

The structural operational semantics rules of Klaim are reported in Table 2.4.

$$(ACTP): \quad \frac{}{A.P \xrightarrow{A} P}$$

$$(PARALLEL): \quad \frac{P \xrightarrow{A} P'}{P|Q \xrightarrow{A} P'|Q}$$

$$(CHOICE): \quad \frac{P \xrightarrow{A} P'}{P + Q \xrightarrow{A} P'}$$

$$(OUT): \quad \frac{P \xrightarrow{out(t)@\ell'} P' \qquad et = \tau[[t]]}{N||\ell :: [K, P]||\ell' :: [K_{\ell'}, P_{\ell'}] \quad \longrightarrow N||\ell :: [K, P']||\ell' :: [(K_{\ell'}, et,)P_{\ell'}]}$$

$$(IN): \quad \frac{P \xrightarrow{in(T)@\ell'} P' \qquad \sigma = match(T, et)}{N||\ell :: [K, P]||\ell' :: [(K_{\ell'}, et), P_{\ell'}] \quad \longrightarrow N||\ell :: [K, P'_{\sigma}]||\ell' :: [(K_{\ell'}, P_{\ell'}]}$$

$$(READ): \quad \frac{P \xrightarrow{read(T)@\ell'} P' \qquad \sigma = match(T, et)}{N||\ell :: [K, P]||\ell' :: [(K_{\ell'}, et), P_{\ell'}] \quad \longrightarrow N||\ell :: [K, P'_{\sigma}]||\ell' :: [(K_{\ell'}, et), P_{\ell'}]}$$

$$(EVAL): \quad \frac{P \xrightarrow{eval(Q)@\ell'} P'}{N||\ell :: [K, P]||\ell' :: [(K_{\ell'}, P_{\ell'}] \quad \longrightarrow N||\ell :: [K, P']||\ell' :: [K_{\ell'}, P_{\ell'}|Q]}$$

Table 2.4: Structural operational semantics rules of Klaim. Adapted from [5]

We describe the structural operational semantics rules of Klaim as follows:

The rule *(ACTP)* expresses a prefixing action. Here a process performs an A action and becomes P.

The rule *(PARALLEL)* expresses that if the process P does an action and becomes P'. Then nothing changed when parallel processes are involved.

The rule *(CHOICE)* expresses that if the process P does an action and becomes P'. Then nothing changed when another process is involved.

The rule *(OUT)* expresses that a process performing the output operation, can insert a tuple resulting from the evaluation of t to the tuple space at a component ℓ'.

The rule *(IN)* expresses that a process is willing to perform an input operation if `T` is evaluated and the pattern matching succeeds, the evaluated tuple `et` is therefore removed from the tuple space and a substitution that is returned is used to the continuation of a process executing the operation.

The similar rule *(READ)* is used to model a process that performs an input operation, but differs from the *IN* rule because the accessed tuple remains in the tuple space, thus the tuple space is not modified.

The rule *(EVAL)* expresses that if there is process *P* that can spawn new process `Q` at location ℓ' and becomes `P`$'$. Therefore, there is a node that contains locations and evolves the new created process `Q` at a locality ℓ' which has process `P`.

An implementation of Klaim coordination language is available online at the link: https://github.com/LorenzoBettini/xklaim.

2.3.3 Example

In this example, we consider a network of nodes, where processes and tuple spaces are allocated.

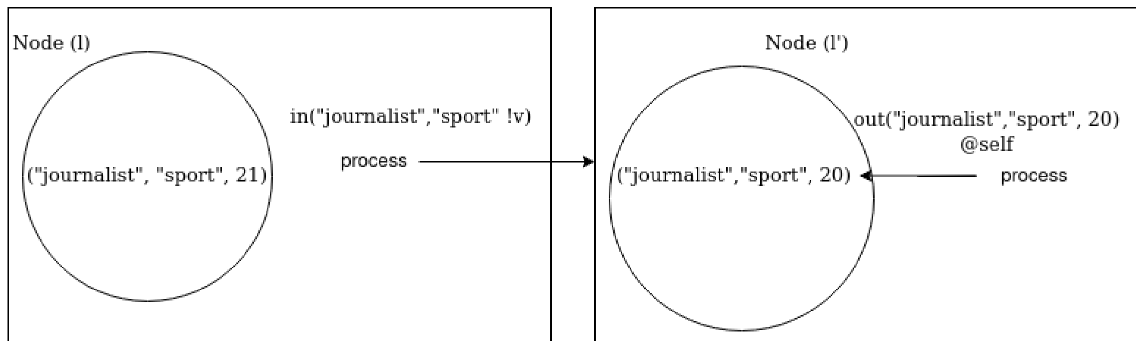


FIGURE 2.2: An example of tuple spaces expressed in Klaim

It illustrates a network of two nodes expressed in Klaim, characterized by the localities ℓ and ℓ', used by processes to manipulate the tuple spaces. Processes use `self` to refer to their execution nodes. In other words, `self` is a distinguished locality which points to the local tuple space. The distribution of tuples in the tuple spaces is performed via the out operation, e.g. `out("journalist","sport",20)@`ℓ puts the tuple to the node with location ℓ. The removal operation of tuples is performed via the in operation, e.g. `in("journalist","sport",21)@`ℓ' after pattern-matching. In

this case, the template, `("journalist","sport",!v)` consists of two defined fields (`"journalist","sport"`) which will be used to find a matching tuple and a variable `v` denoted by a leading symbol `!`, used to indicate variables. The third field can have any value of type integer. As a result, both tuples `("journalist","sport",21)` and `("journalist","sport",20)` matches the template `("journalist","sport",!v)`. However, only one of them will be nodeterministically selected.

Figure 2.2 shows a network of tuple spaces and processes communication expressed in Klaim coordination language.

2.4 Data Replication

Replication [46] is a technique that creates or keeps multiple copies of the same data in an application, called *replicas*, and stores them at several locations (e.g., servers, sites, etc), so that it can be used even when some of the copies are not available due to sites failures. This technique has been widely applied to implement systems consisting of different components including the replicas over which operations are performed. Two operations on the data, write (update) and read are supported by a replication control mechanism. The communication between different system components (clients and replicas) takes place by exchanging messages [2].

A replication model executes a sequence of events when a client requests operations to be performed, and they can be executed in the following five stages [2, 47], (see Figure 2.3):

- *Request*: an operation to one or more replicas is sent by the user.
- *Coordination*: coordination is achieved by servers coordinating with each other to consistently synchronise the execution of the different operations or the requests by clients.
- *Execution*: the replica server performs the requests and stores the results and operations are executed on the replica servers.
- *Agreement*: the replica server agrees on the results obtained from the execution.
- *Response*: the results from the performed operation are sent back to a client. In some cases, the response is sent to a user by one replica server, while in other

cases, the front end gets the results from the collection of replica servers and chooses a single result and sends it back to the user.

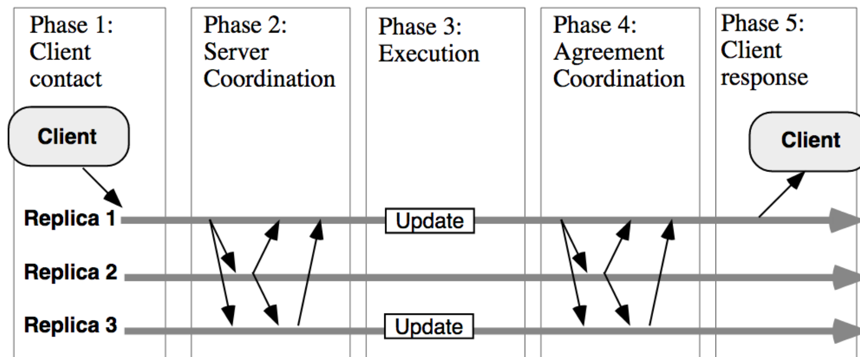


FIGURE 2.3: The phases of a functional replication model [2]

Replication can be classified into two main strategies according to the way in which the replicas are determined.

- *Static strategy* in which the number of replicas and host servers or locations for replication are constant and fixed, it means that they are pre-established during the design phase.
- *Dynamic strategy* in which replicas are determined and removed based on users' access pattern algorithms or on storage capacity available at different locations. In this case the locations for replication can be selected at run-time.

Looking at the ways replicas can be stored, static and dynamic replication strategies can be exploited in the following systems:

- Centralized systems in which all data are stored on the same server.
- Distributed systems in which the data can be stored on different servers.

This thesis focuses on the strategies and issues of replication in distributed systems within the tuple space paradigm. A distributed system is a software system in which the components or programs located on a network communicate and coordinate their activities by exchanging messages.

Replication is widely used and to enhance performances while reducing access latency and to guarantee fault tolerance and high data availability.

***Enhancing performance and reducing access latency*:** When data are stored on a single location or server and too many requests from various users need to be served at the same time, the whole system is slowed down and access latency increases. Replication helps in improving the performance by increasing data locality for replication. When data are stored at multiple locations or sites, users can get the data closest to their sites or servers, response time is thus reduced.

***Guaranteeing data availability and fault tolerance*:** When data are stored on a single location or server, and this location or server fails to respond due to crashes or other sorts of failures, the stored data cannot be accessed. With the use of replications and by storing data at multiple servers or locations if one of the servers or locations fails the system still continues to operate using the replicated data.

One of the key challenges when implementing replication is how to guarantee the consistency of replicas. As replica is not just an ordinary copy of the original data item, when one copy of replica is changed, the remaining copies of data items are also affected, thus a fully equivalent copy should be created for keeping identical replicas. Consistency is guaranteed by using synchronization mechanisms and ensuring that all replicas have the same data values.

2.5 The RepliKlaim Coordination Language

This section describes the tuple-based coordination language, called RepliKlaim, which enriches Klaim with primitives for replica-aware coordination. We start the description with the syntax followed by the communication primitives and operational semantics.

RepliKlaim is a coordination language that adds to Klaim (Section 2.3) new features for dealing with replication. It relies on the programmer's ability to decide where to store data and RepliKlaim is responsible for maintaining the tuple spaces consistent. Two levels of consistency are considered: *weak* and *strong*.

2.5.1 RepliKlaim Syntax

The syntax of RepliKlaim is similar to Klaim but, its communication primitives are extended for replica-aware programming. The main differences from Klaim in practice

are its features to deal with multiple localities instead of a single one, as well as the absence of features to deal with the process mobility (i.e., `eval` action). The syntax of RepliKlaim consists of the descriptions of the networks, repositories, processes, operations, localities, tuples and templates. It specifies the networks of nodes or components. Each component is identified by a unique locality ℓ, a data repository K and parallel processes P. Table 2.5 presents the syntax of RepliKlaim.

$N ::= \ell::[K,P] \mid N \parallel N$	(networks)
$K ::= \emptyset \mid \langle et_i, L\rangle \mid K,K$	(repositories)
$P ::= nil \mid A.P \mid P{+}P \mid P \mid P$	(processes)
$L ::= \epsilon \mid \ell \mid L \bullet L$	(locations)
$T ::= e \mid P \mid \ell \mid !x \mid T,T'$	(templates)
$t ::= e \mid P \mid \ell \mid t,t'$	(tuples)
$A ::= out_s(t_i)@L \mid in_s(T_\tau)@\ell \mid read(T_\tau)@\ell$	(strong actions)
$\mid out_w(t_i)@L \mid in_w(T_\tau)@\ell$	(weak actions)

TABLE 2.5: Syntax of RepliKlaim. Adapted from [6]

A repository is a set of data items specified by a pair of $(\langle et_i, L\rangle)$, where et_i stands for an indexed tuple and L represents the localities in which replicas of tuple et_i are stored. We let `i` denote a unique identifier of the tuple. To better represent data items such as the variables that can be produced and updated, identifier-indexed tuples are used instead of ordinary anonymous tuples.

Processes are created from a *nil* process that does not do any action, the constructs for the parallel processes P | P, the non-deterministic choice P + P and from the action prefixing A.P where a process performs the action A and becomes P. Moreover, they can be executed either at different localities or at the same locality.

The syntax of RepliKlaim is also defined by a set of localities $L = \{\ell_1, \ell_2, \dots, \ell_n\}$, a set of basic values $\mu = \{u, v, w, \dots\}$, a set of variables $\nu = \{!x, !y, !z, \dots\}$, a set of tuple identifiers $I = \{i, i', \dots\}$, a set of indexed evaluated tuples $\epsilon\tau = \{et_i, et_i', \dots\}$, a set of indexed tuples $t = \{t_i, t_i', \dots\}$, and a set of templates $T = \{T_\iota, T_\iota', \dots\}$.

It is worth noting that in Repliklaim, a locality ℓ in L can appear as ℓ or as $\underline{\ell}$ (For example, see the rules OUT_w and IN_w in Table 2.7). Notation $\underline{\ell}$ is used to indicate ownership of the tuple. In fact, each replica should exactly have one owner, i.e., every occurrence of L has at most one owner location $\underline{\ell}$. This approach is used to prevent data inconsistencies due to concurrent weak operations, i.e, read or removal of a replicated tuple. Note that, we used the subscripts letters w and s in the names of operations to indicate whether operations preserve weak or strong consistency.

Similarly to Klaim, RepliKlaim provides a set of blocking and non-blocking operations that add, search and remove the tuples from or to the tuple spaces. The tuples in RepliKlaim, i.e., *replicated tuples*, have the same format as Klaim's tuples.

Possible operations to manipulate tuple spaces in RepliKlaim coordination language along with the corresponding Klaim versions are shown in Table 2.6. Blocking operations are marked with the symbol (*).

Description	**Klaim**	**RepliKlaim**
Put a tuple	node.out(t)@ℓ	node.out$_\alpha$(t,ℓ_1...ℓ_n)
Read a tuple*	node.read(T)@ℓ	node.read(T,ℓ)
Search and remove a tuple*	node.in(T)@ℓ	node.in$_s$(T,ℓ)
Search and remove a tuple	n/a	node.in$_w$(T,ℓ)

TABLE 2.6: Basic operations on tuple spaces in Klaim and RepliKlaim

The operations of RepliKlaim are described as follows.

The non-blocking output operation out$_\alpha$(t,$\ell_1 \ldots \ell_n$) permits to add the tuple t to the data repositories located at all localities $\ell \epsilon L$ $(L = \ell_1 \ldots \ell_n)$ atomically (when strong consistency is required) or asynchronously (in case of weak consistency) (respectively for α = s or α = w). Thus, the shared tuple is replicated to every locality in L.

The input operation read(T,ℓ) reads from a tuple space. It uses the pattern T to find a matching tuple (if any) from locality ℓ, but it does not remove the matching tuple. In case, no matching tuple is found, the operation blocks until a matching tuple becomes available.

The blocking operation in$_s$(T,ℓ) searches for a tuple matching the pattern T at ℓ and atomically removes all replicas of that tuple, thus preserving strong consistency.

Operation $\texttt{in}_w(T,\ell)$ can also be performed on tuple spaces asynchronously in order to remove all replicas of a tuple that match T. This operation only preserves weak consistency. In short, given a pattern T, $\texttt{in}_w$ removes a matching tuple and all its copies, but if there is no tuple matching a given pattern, it returns false instead of blocking.

The forms of code mobility based on sending the actual code through the operation $\texttt{eval(-)@}\ell$ is not implemented in RepliKlaim coordination language.

2.5.2 Operational Semantics of RepliKlaim

The structural operational semantics of RepliKlaim coordination language are useful for designing its implementation. They are presented in Table 2.7 and described below.

$$(ACTP): \quad \frac{}{A.P \xrightarrow{A} P}$$

$$(PAR): \quad \frac{P \xrightarrow{A} P'}{P|Q \xrightarrow{A} P'|Q}$$

$$(CHOICE): \quad \frac{P \xrightarrow{A} P'}{P + Q \xrightarrow{A} P'}$$

$$(OUT_s): \quad \frac{P \xrightarrow{out_s(t_i)@L} P' \qquad \forall \ell' \in L. \nexists et', L'. \langle et'_i, L' \rangle \in K_{\ell'} \qquad et_i = \tau[[t_i]]}{N \| \ell :: [K, P] \| \Pi_{\ell' \in L} \ell' :: [(K_{\ell'}, P_{\ell'}] \quad \longrightarrow N \| \ell :: [K, P'] \| \Pi_{\ell' \in L} \ell' :: [K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}]}$$

$$(IN_s): \quad \frac{P \xrightarrow{in_s(T_\iota)@\ell''} P' \qquad \ell'' \in L \qquad \sigma = match(T_\iota, et_i)}{N \| \ell :: [K, P] \| \Pi_{\ell' \in L} \ell' :: [(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}] \quad \longrightarrow N \| \ell :: [K, P'\sigma] \| \Pi_{\ell' \in L} \ell' :: [K_{\ell'}, P_{\ell'}]}$$

$$(READ): \quad \frac{P \xrightarrow{read(T_\iota)@\ell'} P' \qquad \sigma = match(T_\iota, et_i)}{N \| \ell :: [K, P] \| \ell' :: [(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}] \longrightarrow N \| \ell :: [K, P'\sigma] \| \ell' :: [(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}]}$$

$$(OUT_w): \quad \frac{P \xrightarrow{out_w(t_i)@L} P' \qquad \underline{\ell''} \in L \qquad \nexists et', L'. \langle et'_i, L' \rangle \in K_{\ell''} \qquad et_i = \tau[[t_i]]}{N \| \ell :: [K, P] \| \ell'' :: [K_{\ell''}, P_{\ell''}] \quad \longrightarrow N \| \ell :: [K, P'] \| \ell'' :: [(K_{\ell''}, \langle et_i, L \rangle), P_{\ell''} | \Pi_{\ell' \in (L \setminus \ell'')} eval(out_u(et_i, L)@\ell']}$$

$$(IN_w): \quad \frac{P \xrightarrow{in_w(T_\iota)@\ell''} P' \qquad \ell'' \in L \qquad \underline{\ell''} \in L \qquad \sigma = match(T_\iota, et_i)}{N \| \ell :: [K, P] \| \ell' :: [(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}] \quad \longrightarrow N \| \ell :: [K, P'\sigma] \| \ell' :: [(K_{\ell'}, P_{\ell'} | \Pi_{\ell''' \in (L \setminus \ell')} eval(in_u(et_i, L)@\ell''']}$$

TABLE 2.7: Operational semantics rules of RepliKlaim [6]

The rule *ACTP* expresses a process that is willing to perform action A and behaves like P. The rule *CHOICE* expresses a process that executes an action and behaves like $\mathtt{P}'$ and nothing changes when another process is involved. The rule *PAR* models the process P that executes an action and behaves like $\mathtt{P}'$, and nothing changes when parallel processes are involved.

The rule *(OUT$_S$)* models the behaviour of a process P executing a strong output operation $out_s(t_i)@L$ that places an evaluated tuple et_i in all locations in L and behaves like P'. Therefore, a network N replicates the tuple et_i in all locations within L. However, this can be done only if a version of the same data item et' that is going to be added, does not exist in the repository of the owner of et'(i.e., ℓ'').

The rule (OUT_W) models the behaviour of a process that executes a weak output action of the form $out_w t_i@L$ by enforcing the absence of a version of data item i. The difference with respect to the strong output action is in the creation of the process that is responsible of placing the evaluated tuple $< et_i, L >$ at all locations within L except in itself (i.e., ℓ'').

The rule (IN_S) models the behaviour of a process executing an input operation that deals with the removal of a copy of tuple that matches a given pattern from location ℓ and from all locations in the set L.

The rule (IN_W) removes an evaluated tuple et_i from an owner ℓ' of a tuple that has a replica in the target location ℓ.

The rule *(READ)* supports process to read tuples from location ℓ' and behaves like $\mathtt{P}'$ but stays at its initial locality ℓ'.

2.5.3 Example

The example shown in Figure 2.4 consists of a network of localities `loc1` and `loc2`, used by processes to manipulate the tuple spaces. It illustrates concurrent read and strong output operations. The idea is that component `loc1` can put a tuple with replicas in `loc1` in a strong manner, while `loc2` is reading the tuple from `loc1`. The read operation can happen only after all replicas are created.

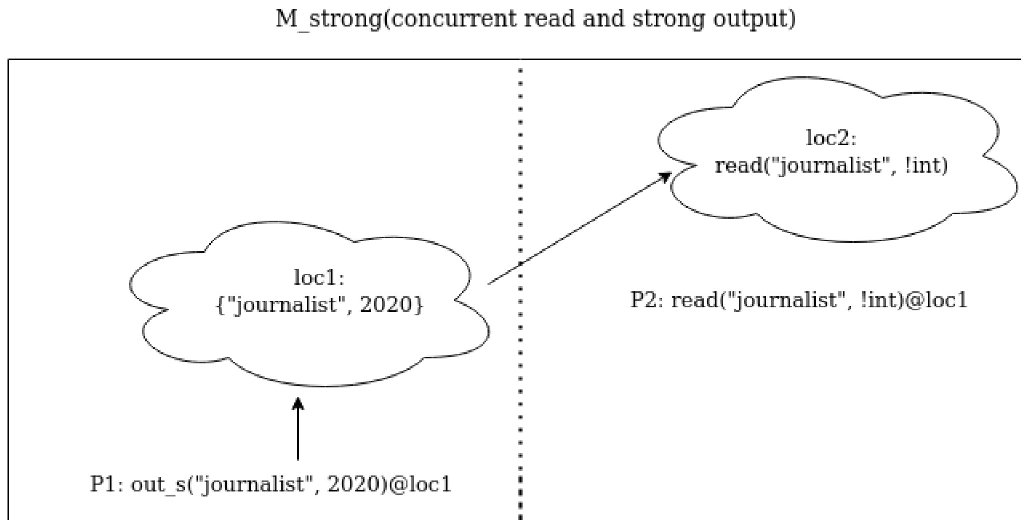


FIGURE 2.4: An example of tuple space expressed in RepliKlaim

2.6 Related Work on Coordination Languages with Replication

In this section, we relate our work to the state of the art techniques and systems for tuple spaces based coordination languages, which exploited replication techniques. Replication and consistency are studied in distributed systems. In particular, they are applied in the tuple spaces coordination languages, and have maintained the consistency levels.

In addition to the pSpaces family of implementations, tuple space systems have been proposed in different programming languages. Java implementations include Klava [48] for Klaim, and jRESP[1] for SCEL [49] (Java Run-time Environment for SCEL Programs), a Java framework that provides programmers with a set of APIs that simplify design, development and coordination of distributed systems. jSpace, the implementation of pSpaces in Java, was initially based on a fork of jRESP. We chose to work on top of pSpaces because it is actively maintained.

RepliKlaim [6] is a tuple-based coordination language that builds on Klaim language and enriches it with replica-aware coordination primitives. In particular, it relies on the programmers to decide themselves where to place specific copies of tuples and is responsible for preserving the strong and weak consistency views of tuple spaces.

[1] http://jresp.sourceforge.net/

Beside RepliKlaim coordination language, tuple replication is also implemented in X10, a general-purpose and object-oriented programming language for large-scale distributed systems [50, 51]. However, the proposed approach introduces primitives for data sharing based on a centralized data location which is avoided in our approach, and the responsibility to control replication of tuples across the places is left to the programmer.

Lime (Linda In Mobile Environment) [52] is a distributed tuple space for ad-hoc mobile networks. As far as we know, it did not consider tuple replication aspects. A version of Lime, proposed by Murphy et al. [16], is a replication based tuple space model that improves data availability. Our research work, however, focuses on improving the performance of distributed systems that use tuple spaces, and can additionally guarantee data availability. It uses replication profiles to define which tuples and how copies of tuples should be replicated among the gathered devices. In other words, it exploits the predefined replication policies to determine how replicas should be created and updated.

A replication-based tuple distribution is proposed in [53] to improve the performance of tuple access. Russello et al. [54] proposed an approach which relies on replication of tuples and self adaptation for optimizing replica placement and improving performance and data availability. The self adaptation paradigm relies on the numbers of reads and writes performed by certain nodes. To this end, they have applied different replication strategies such as full replication that replicates tuples at all available nodes, the replication policies that replicate tuples to a certain number of nodes (a group of n nodes) and other policies which replicate tuples to the nodes that produce and consume tuples of the same data types. The mechanisms that preserve consistency have to be considered.

In [55], various replica placement strategies are integrated into the Linda coordination model for sharing data, guaranteeing data availability, and decreasing the access latency. Consistency levels (strong and weak) are considered but at the level of data items instead of the operations as in our approach.

All the above-mentioned approaches are similar to RepliKlaim coordination language in that the responsibility to control replication is left to programmers.

Alternative distribution mechanisms for tuple spaces based on the concept of ghost tuples have been proposed in [56, 57], where it is the system that may decide not to eliminate tuples for using them later. In the attempt to limit the activity on the tuple space and increase scalability, a hierarchical tuple space model that supports partial replication has been proposed in [58]. Our approach, however, is a fully-automated replication, and improves the performance of distributed systems.

Spatial distribution of tuples is a rather different approach to ours where tuples contain both content and replication rules [59]; in this model the propagation of the tuples is asynchronous and thus strong consistency has to be explicitly considered and programmed. Consistency models for replicated data are covered in [60]. TOTAM [61] is another tuple space based coordination model. It uses replication approach and consistent framework to support the implementation of mobile context-aware applications. However, it suffers from the limitations of relying on the programmer to control over which tuples present in the tuple space should be accessed by the applications and which one should be deleted.

Harnie et al. [62] introduced a tuple based model that use replication and distributed removal of tuples approaches to support the programming of urban area applications where public transportation can be exploited [62]. In particular, the replication has been used to increase data availability in a highly disconnected environment as it enables intermediary nodes to carry out information to and from their eventual destination. The distributed removal concept relies on the programmers to delete the tuple from its tuple space manually or based on the expiration dates.

Bakken et al. [63] propose FT-Linda, a variant of Linda that exploits replication to address fault tolerance, but they did not consider any consistency level of tuple space.
A replica tuple-space based coordination models that focus on fault tolerance and security have been proposed [64, 65]. The authors have applied Byzantine fault-tolerant state machine replication and Byzantine quorum systems replication techniques in order to control the access mechanisms.

Casadei et al. [66] introduced a dynamic matching approach that is based on only grouping the tuples of the same type in specific spaces so that they can be quickly located. However, the authors did not consider any consistent view of tuple space, when a tuple is removed from its space among other spaces. Russello et al. [67] propose a technique that dynamically uses replication policies to handle availability in the shared data space systems and to improve performance. The proposed technique differs from our approach because it relies on the programmers abilities to coordinate the replication of tuples. Dynamic replication has also been considered in [68].

2.7 Programming with Spaces (pSpaces)

pSpaces[2] is a coordination model that supports the development of distributed applications with tuple spaces in different modern development platforms. Existing implementations of pSpaces are available in Go, Java, and Swift, and are based on the formal operational semantic rules of Klaim (Section 2.3.2).

The difference between Klaim and goSpace is in the way the reading operations: in Klaim, when a matching tuple is found, a substitution is generated and applied to the process executing the action, while in goSpace (according to the descriptions of goSpace's operations given in Section 2.7.2) the found tuple is returned. Another difference between them is the lack of a mechanism to support process and code mobility. In Klaim, mobility provides a suitable primitive (i.e., `eval` action) that spawns processes, which can be moved from one locality to another and be remotely executed. Note that in this thesis, we do not consider process spawning as it is not currently implemented in goSpace, and it is not relevant for our analysis.

Similarly to Klaim and Linda coordination languages, pSpaces handles data in forms of tuples. We recall that a tuple is a finite ordered lists of elements that can be used to represent the messages or data items and can consist of several structured values. In pSpaces, a space is a collection of tuples. Spaces can be either local or remote, in the sense that they can be located on another device. A remote space supports the same operations as for local spaces, but it needs slightly different operations to be created and connected with. Every space is associated with a unique uniform resource identifier (URI) encoded as a string, i.e., the *space identifier*.

In this thesis, we make no explicit distinction between local and remote spaces: each component manipulating the tuple spaces is also associated its own URI, which makes it possible to figure out whether a space is local or not.

Note that pSpaces and Klaim are the same in the sense of supplying the same coordination primitives, such as write, read and remove (with the exception of code mobility in Klaim that enables processes to exchange pieces of code through the communication primitive `eval`) with the main differences of being syntactically different and supporting the implementation of distributed applications in different platforms (for pSpaces).

[2]//github.com/pSpaces/

2.7.1 pSpaces Syntax

pSpace allows multiple applications. Each application is implemented by using a specific programming language (e.g., Go, Java, etc). Moreover, it contains a memory M and concurrent processes P executing operations. Memory M represents a map of the variables into values. Among those values we can mention Spaces. Parallel composition of applications use the operator ||.

A process can be either an inactive process or nil, a prefixing process $A.P$, or a parallel composition of two components $C||C$. Hence, behaviours of a component are modelled by a process `P` executing actions. The actions of a process mainly consist of creating a local space and remote space. Tuples are indicated by non-empty lists of expressions `e`, while a template is shown by non-empty lists of expressions `e` or types τ.

pSpace provides operations for creating local and remote spaces, described as follows:

- `Station:= NewSpace("tcp://localhost:31419/space")` - creates a space named Station,
- `Server:= NewRemoteSpace("tcp://serverhost:3115/room12")` - creates a remote space named `Server` at the given `uri`.
  `Server` is the name of space, `tcp` is the communication protocol, `serverhost` is the address of a server, `3115` is the port number and `room12` is the space identifier.

pSpace relies on communication primitives similar to those of Klaim, essentially a set of blocking and non-blocking actions to add, search, and remove tuples to or from a space. Moreover, tuples are content-addressable based on the pattern matching mechanisms. Non-deterministic mechanism is implemented in the pSpaces communication primitives. When one needs to retrieve a tuple with a pattern `T`, and there is more than one tuple that match the given pattern.

pSpaces provides input operations (`Get` and `GetP`) for removing the tuple `t` that matches the given pattern `T`. The input operations (`Query` and `QueryP`) are used for searching and reading tuples that match the given pattern `T`, and output operation `(Put)` for adding a tuple to a space. The pattern matching mechanism of pSpaces is similar to Klaim's one described in Section 2.2 and in Table 2.3.

In most pSpaces implementations, the formal fields used to select a tuple that matches a given template are specified by associating data types to names or value lists. The

reader interested in programming with Spaces (pSpaces) is referred to [7] for additional details.

The software package *goSpace*[3] is an implementation of pSpaces in Go that supports the development of concurrent and distributed applications with Spaces. Similar to Klaim, goSpace is a tuple space coordination model in which tuples are shared on a network-based memory, called *Space*. The space serves as both tuples storage and their exchange location, where the tuples can be concurrently accessed from the processes through the distributed communication primitives.

2.7.2 GoSpace Syntax

The syntax of goSpace is defined by the grammar shown in Table 2.8. It consists of the system S that represents a multiset of applications. The process P represents the concurrent processes executing operations. Processes can be created from the nil process that does not perform any action, a process prefixed with an action to be executed, the parallel composition (P_1 | P_2, which enables the computation in processes P_1 and P_2 to proceed simultaneously), and choice between actions (P_1 + P_2, either proceed with P_1 or P_2). The communication primitives of goSpace enable a process to write `(Put)` the tuples to a space, read and remove `(Query/QueryP/Get/GetP)` the tuples from a space based on the pattern-matching rules.

The pattern-matching rules of goSpace are similar to the ones of Klaim (see Section 2.3.2). However, in the implementation of goSpace, the formal fields are specified with pointer values (e.g., &v, where `v` is the name of a variable). For example, the pattern for retrieving the number of packages of tea would be `("tea",&numberpackages)`.

We assume a tuple is denoted by the non-empty lists of expressions e or a sequence of the tuples. Templates are represented either by a template of the type τ or types τ or the non-empty lists of expressions e.

[3]https://github.com/pSpaces/goSpace

$S ::= \emptyset \mid \texttt{S} \parallel \texttt{S} \mid \texttt{App(host, M, P)}$ (system)

$P ::= \texttt{0}$ (null process)

$\mid \texttt{A.P}$ (action prefixing)

$\mid \texttt{P}_1 \texttt{ + P}_2$ (choice)

$\mid \texttt{P}_1 \texttt{ | P}_2$ (parallel composition)

$t ::= \texttt{e} \mid \texttt{e,t}$ (tuples)

$T ::= \texttt{e} \mid \tau \mid \texttt{e,T} \mid \tau\texttt{,T}$ (templates)

$A ::= \texttt{space.Put(t)}$ (operations)

$\mid \texttt{space.Query(T)} \mid \texttt{space.QueryP(T)}$

$\mid \texttt{space.Get(T)} \mid \texttt{space.GetP(T)}$

TABLE 2.8: Syntax of goSpace [7]

The implementation of goSpace relies on the communication primitives similar to those of Klaim. GoSpace provides a set of blocking and non-blocking operations that add, retrieve and remove tuples from or to spaces. The pattern matching rules for retrieving the tuples work in similar way to the one of Klaim. More specifically, the templates (formal fields) and tuples (actual fields) must have the same number of elements, the same type for each corresponding element of both tuples and templates, and the values of fields are same. The possible operations of the goSpace are described as follows:

- `s.Put(t)` - is a non-blocking operation that places the tuple `t` to the space `s`.
- `s.Query(T)` - scans the space `s` to look for a tuple that satisfies the pattern `T`, blocks until a tuple is found. It returns the matched tuple, if any, but does not remove it from the space.
- `s.QueryP(T)` - is a non-blocking version of `Query` operation. It looks for a tuple matching the given pattern in the space and returns the found tuple (if any). In addition, it returns a boolean value indicating whether the operation was successful or not.
- `s.Get(T)` - removes from the space `s`, a tuple matching the given pattern while keeping note of the matched values. If there is no matching tuple, the process that executes the operation blocks until such a tuple is found.

- `s.GetP(T)` - is a non-blocking version of `Get` operation. It removes from the space `s`, a tuple matching the pattern `T`, and eventually returns a boolean value to indicate whether the operation was successful.

2.8 The Go Programming Language

Go, also known as *Golang* [23, 69] is an open source, object-oriented and concurrent programming language developed at Google by Griesemer, Pike and Thompson. The first stable version *Go1* has been released in March 2012 and the current version *Go1.17* has been released in August 2021 [70]. It has gained rapid attraction over the past few years by several research works published, and is now being adopted in the implementation of software and real world applications. In contrast to other object-oriented programming languages like C++ and Java, Go language does not support the inheritance and does not have some concepts related to object-oriented programming language, such as polymorphism and function overloading. Instead, it provides the interface types and functions as the means of representing collection of methods and perform the polymorphism. Instead of having the header files like in C++ programming language, it carries with it the packages systems and builds in tools for implementing the Go code.

2.8.1 Concurrency

Go is mainly designed with the aim of improving the traditional multi-threaded programming languages by strengthening and providing the easiest concurrent programming languages. To this end, its strength lies on its high-level concurrency mechanisms such as, the thread model and its synchronization mechanisms. Go's multi-threads models are designed using two principles [71]: 1) make threads (goroutines) lightweight and easy to create and 2) use explicit messaging (channel) to communicate among threads.

Go supports concurrency through so-called *goroutines*. They are lightweight functions which can execute concurrently with other goroutines in the same addresses space (obtained from the Go programming language specification) and communicate through the channels. More concretely, goroutines permit users to execute programs simultaneously because a program can be asynchronous, and can exhibit different behaviors. A goroutine behaves similarly to a thread in other programming languages. It is created by adding the keyword `"go"` before a function call.

Go supports different traditional synchronization primitives such as lock and unlock (for mutexes). The native inter-thread synchronisation mechanisms in Go differ from more traditional synchronisation mechanisms over shared memory by promoting the slogan:"don't communicate by sharing memory; instead, share memory by communicating [69]", and encouraging the communication via channels [72]. Therefore, information is mainly exchanged via channels rather than shared memory.

Channels in Go allow goroutines to communicate. The use of channel ensures that only one goroutine has an access to the data at a time, thus helping to develop simpler and higher quality concurrent applications. Go supports two types of channels [71]: *buffered* and *unbuffered*.

Unbuffered channel is a channel that has initially no capacity of storing the messages into it. It therefore requires the user to insert a message in order to have the goroutine process unblocked by a channel. It basically blocks a goroutine whenever the channel is empty and waits to be filled up. Sending data to (or receiving data from) an *unbuffered channel* will block a goroutine until another one removes data from or sends data to the channel.

Buffered channel offers the possibility of storing messages inside it. It could be filled up to its defined capacity, not with just one message. Sending data to a *buffered channel* will block only when the buffer is full, while like before receiving will block when the buffer is empty.

2.8.2 Examples

Go program consists of packages (i.e., folders) which contain *.go files*. Each program begins with a package declaration, a list of imports, several functions, type declarations and variables. A program starts running in the package `main`, and using the package with import path `"fmt"` as well as other necessary ones. Consider the simple examples in Listings 2.1 and 2.3 that present how to spawn threads (goroutines) and how to use channels. Listing 2.1 shows how to spawn a goroutine. In the line 12, `go say()` starts a new goroutine. Now `say()` function runs concurrently along with the `main()` function. We used a `Sleep` of the time package at line 13 to make the `main()` wait for the `say()` goroutine to terminate.

```
1 package main
2 import (
3   "fmt"
4   "time"
5 )
6 func say(s string) {
7   for i := 0; i < 2; i++ {
8     fmt.Println(s)
9   }
10 }
11 func main() {
12   go say("Greetings")
13   time.Sleep(50 * time.Millisecond)
14   say("Hello world!")
15 }
```

LISTING 2.1: A concurrent Go program spawning a goroutine

```
1 Greetings
2 Greetings
3 Hello world!
4 Hello world!
```

LISTING 2.2: Output for the program shown in Listing 2.1

(Listing 2.3) presents two goroutines (writer and reader) communicating and synchronizing their activities through the channels.

```
1 package main
2 import "fmt"
3
4 func writer(msg chan string, done chan bool) {
5   fmt.Println("Writing a message...")
6     //sending message to a channel msg.
7     msg <- "acknowledge the message"
8     //sign of termination.
9     done <- true
10 }
11
12 func reader(msg chan string, done chan bool) {
13     // read from channel msg.
14     msgvar := <-msg
15     fmt.Printf("Reading message.: %s\n", msgvar)
16     done <- true
17 }
18
19 func main() {
20     // create channels.
21     done := make(chan bool)
22     a := make(chan string)
23     //start to asynchronize goroutines.
24       go writer(a, done)
25       go reader(a, done)
26     //wait for receiving the message.
27       <-done
28       <-done
29     fmt.Println(" All goroutines are successfully done!")
30 }
```

LISTING 2.3: Example of communication of goroutines and channels

```
1 Writing a message...
2 Reading message.: acknowledge the message
3 All goroutines are successfully done!
```

LISTING 2.4: Output for the program shown in Listing 2.3

2.9 Replication Models and Techniques

2.9.1 Replication models

In distributed system, there exist various ways where a single data replica can be updated. It requires some update synchronization mechanisms to guarantee a consistent level whenever a read or write operation is executed. The replica updates synchronizations can be done in two ways: synchronous and asynchronous models [73].

A synchronous replication (also known as *eager replication*) requires the update (write) operation on all replicas of the same data items to be done immediately whenever some

changes are made. More specifically, all replicas are locked such that a write action can be executed on all data replicas simultaneously, and these replicas are synchronized. Consequently, all replicas have the same data values, and it can only be completed if replicas have successfully acknowledged the update (write) operation.

An asynchronous replication (also known as *lazy replication*) involves an update of only one replica and propagates the changes to other replicas after the update (write) operations are completed on that single replica.

2.9.2 Replication Techniques

Based on the synchronization replication models described (Section 2.9), we describe some important techniques for data replication as follows [2, 47, 73].

- *Primary-copy*: this technique only relies on the primary copy to execute the update. At any time, there is only one single primary replica manager and one or more secondary replica managers (or slaves). A primary copy of a data refers to as an original data that is created first. In this technique, a write request is submitted to a secondary replica manager, the same request is forwarded to a primary replica manager that will update (or write) and propagate the changes to the rest of all secondary replica managers. All secondary replicas managers are therefore used for read-only queries. The most important thing to consider is that, there should be only one primary copy in the whole system executing the transaction.

- *Update anywhere* (also known as *Group*): Any site containing a replica of the data can update it. It is the most used replication method and it uses the locking mechanisms in which the clients are allowed to update the replica anywhere in distributed systems. Note that if an efficient design is not taken into consideration, this method may affect the performance more than the primary copy.

- *Read-one write-all (ROWA)*: requires a read operation to be executed from any replica of data item, while a write operation has to be performed on all replicas. It uses synchronization mechanism of the write operations and in this manner they are written simultaneously. As all replicas are expected to remain in the synchronized mode all the time, it maintains the consistency by reading one copy of data item and writing all copies. One of the issues of this technique is that the write operations are expensive and sensitive due to a single replica failure.

2.10 Consistency Levels for Replicated Data

Consistency is a formal description that presents how the copies from the same data vary through the write and read operations within the same replicated systems. It basically describes the access rules to any copy of data, and thus ordering distributed memory updates. The data consistency issue deals with the implementation of either a synchronous or an asynchronous models for replicated system that assist its exploration.

In the synchronous model, all replicas are synchronously updated at once, any process can use a local replica and change it. Once an update is completed, all replicas are synchronized. In the asynchronous model, one replica is updated immediately while and the others are asynchronously updated at a certain moment in time.

Possible approaches for guaranteeing consistency level when implementing data replication are the following [74] based on whether the system sends the data update to:

1. all replicas at the same time: all of them updated at once;
2. an agreed-upon master node first that resolves all requests to update the data item: the order chosen to perform these updates determines the order in which all replicas perform the updates. After the master node resolves updates, it replicates them to all replica locations;
3. a single (arbitrary) node first: the system performs the updates at that location and then propagates them to the other replicas.

The degree of consistency of replicas relies on the chosen approach and can have different forms described as follows [60, 75]:

***Strong consistency*:** Strong consistency is a property to define consistency models and requires all replicas to receive the information in the same order. After the update is completed, any successive access action can return the updated values. Therefore, users cannot notice that the data are replicated. This approach is identified as easier to reason about and as guaranteeing software and applications to be more trustworthy. It is the most expensive in terms of synchronization but increases the level of data availability. For example, distributed services such as booking a flight and withdrawing money from a bank account demand a strong consistency level.

***Weak consistency*:** is a another type of consistency that relies on some conditions or period of times for synchronizing the updates. The system does not guarantee that the subsequent accesses will return the updated values. The weak consistency model is not always sufficient for applications because it might lead to data inconsistency but it has the advantage of reducing the cost of synchronization and of machine latency. For example, weak consistency level may be required when reading temperatures from several sensors at different locations.

***Eventual consistency*:** This is a variant of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. Simply, if there are no new write operations invoked on the object, eventually all reads will return the same value. This model is easier to implement and simplifies the design and operation of distributed services. Moreover, it prioritizes high availability over strong consistency and improves performances. For example, the domain name system (DNS) implements eventual consistency, where an update to a name is distributed based on the configured pattern and in combination with the time controlled caches; eventually all clients can see the updates [76].

2.11 Concepts of Static Program Analysis

Static program analysis aims at automatically analyzing a given program for determining whether that program satisfies some particular properties. This is a significant research topic because it addresses an important problem that is recognized to be undecidable (Rice's theorem). Informally, Rice's theorem states that any non-trivial semantic property of a program is undecidable [77]: in other words, it is not possible to design a static analysis approach that can prove any non-trivial property on a given program both automatically and exactly. A semantic property can be concerned with the program's behavior (for example, does the given program terminate for all inputs), unlike a syntactic property (for example, does the given program contain an if-then-else statement). Some undecidable properties such as a program that never divides by zero, or a program never dereferences a null pointer have motivated several researchers to design and implement some approximations analyses. For example, Model-checking [78] is among the proposed techniques, and it shows sequences of states occurring during a program's execution and decides whether these states satisfy a safety property. Hence,

it is deemed semi-decidable and precise of an abstract model of a program whereas, data flow analysis is considered terminating [79].

Static analysis procedure supports computing over-approximations and under approximations analyses of program behaviors. We here provide a brief overview of static analysis. We will not go into details but, we want to introduce some basic concepts and static analysis techniques [80–82].

Static program analysis (also known as source code analysis) [81] is a technique that automatically reasons about the behaviors of a computer program without having to execute the programs. It is mostly conducted on the source code and by using effective approximations, it can check all the possible executions of a given program, generate results after inspecting the structure of its source code, the sequences of statements, and the way the variable values are refined over several function calls. In static analysis, the programs are not executed but are analyzed by a static analyzer to generate useful information for the program development process in order provide good quality of software. A static analyzer is a program that can take other programs as input and indicate whether they satisfy the properties of interest or not.

Static program analysis usually requires soundness. Soundness analysis provides guarantees that the information obtained from the analysis holds on all possible program executions, whereas unsound analysis makes no such guarantees. The main advantage of static analysis is that all the code is analyzed. This differs from dynamic program analysis where portions of source code will be considered only under some specific conditions that could never be met during the analysis phase [83].

We distinguish between over-approximation and under-approximation analyses:

- *Over-approximation* designates techniques that explore a set of all possible executions of the input program. It estimates the program behaviors that may occur along all the execution paths of the same program. Over-approximation can spot all bugs in the analyzed program and may produce false negatives. Abstract Interpretation [84–87] is usually presented for over-approximation analysis. It aims at computing an over-approximation of the set of states that is reachable in a program. More concretely, it detects invariants of a specific shape (called an abstract domain) at each location of such a program and enables one to check the properties of interest. Abstract interpretation has been used for designing Polyspace

Bug Finder[4], a static analyzer whose purpose was to prove that a program under analysis satisfies some properties such as, a program never divides by zero, in order to prove absence of run-time errors under all possible control and data flows in the source code for the C and C++ programming languages.

- *Under-approximation* analysis assists in capturing a subset of all possible behaviors of a given program. It basically estimates the program behaviors that must occur along all its execution paths. Under-approximation analysis may miss some bugs and result in false positives, i.e., a non-terminating program is considered to unconditionally terminate, or an unsafe program may be considered safe. A program unconditionally terminates if all of its executions are of finite length. Bounded Model-Checking (BMC) [88–90] is designed for under-approximation analysis. It is a symbolic technique for program analysis where only subsets of feasible program behaviors are explored. Given a program, a property and a bound k, it checks whether the property (that typically represents the negated form of some error condition) can be violated within k execution steps [91]. For example, Bounded Model Checking is used to find errors in concurrent software [92].

Static program analysis is used for many purposes such as optimizing compilers and semantic based program manipulation tools (e.g., error detection, program understanding tool). It provides approximations formalized via specific techniques (e.g., abstract interpretation) and can offer the best trade-off between the precision information extracted from a given input program and the efficiency of algorithms used to extract information from the program text [93].

Several techniques can be used to perform static program analysis. They explore the program's behaviors for all possible inputs and the possible states that a given program can reach: Data Flow Analysis and Control Flow Analysis [81], Syntactic Pattern Matching [94], Constraint Based Analysis [81], and Abstract Interpretation [84] are among the most popular. We now briefly describe some of the static analysis techniques.

2.11.1 Static Analysis Techniques

Syntactic Pattern Matching [94] is a technique that inspects a program by relying on the parser to analyze its syntax. The parser takes as input the source code of a

[4] http://www.mathworks.fr/products/polyspace-bug-finder/

given program and produces as output a data structure in the form of an abstract syntax tree. *An abstract syntax tree* is a tree representation of a syntactic structure of a source code written in a programming language. The nodes of such a tree show the constructs (e.g., statements, expressions, etc.) that occur in the source code of a program. Additional comments are reported in section 2.11.2.

Data Flow Analysis [79, 83, 95–97] is a technique that predicts *the flow of information* through the locations of a given program. In fact, it can be used to provide information about the program's behavior without running it. For example, data flow analysis can be used to predict the set of possible arithmetic expressions that have been evaluated at a program location, or the set of variables that have constant values in the program. There are two classes of flow of information analysis [97]: They can be computed either backward or forward.

For the first class problem (i.e., Forward flow), given a point P in the program, it shows what happens before control reaches P (i.e., which definitions can affect computations at that point). For example, in the reaching definition problem, one wants to know which definitions (e.g., statements such as int x = 3) is reachable at the point P. The variable is said to be defined at the point P in the flow graph, if it appears at the left side of an assignment (i.e., if its value is changed).

For the second class problem (i.e., Backward flow), given a point in the program, it shows what happens after control leaves that point (i.e., what can be affected by computations from that point). For example, in the live variables problem, one wants to know the live variables, with a variable a being considered live at a point P in the flow graph, if the value of a at P will be used afterward in some path starting in p.

One of the uses of the data flow analysis technique has been on verifying a property of a concurrent program as a pattern of selected program events and asks the analysis to verify if all program executions satisfy the given property or not [98].

Control Flow Analysis [99] is a technique that inspects the given program and presents how the flow of control are hierarchically sequenced. The key observation is that it makes all possible execution paths of a given program to be analyzed. Usually, the sensitive and flow-insensitive analyses take place; a flow-sensitive analysis respects the order of statements, whereas a flow-insensitive analysis does not respect it. The sequences of control are expressed as a control-flow graph where nodes represent the basic blocks of code (e.g., statements, expressions), while the directed edges indicate the possible flow of control between the nodes.

2.11.2 Abstract Syntax Trees

An abstract syntax tree is a data structure that consists of program terms (e.g., expressions and statements) that represent a computation. It is used in the construction of compilers. It has a form of a tree and consists of program terms (e.g., expressions and statements) that represent its syntactic structure or computation.

An AST consists of two main components namely, the source code and the tokens. The construction of an AST in GO programming language is performed into three phases:

1. *Lexical Analysis or Tokenization* - the compiler scans the given input program file or expressions and generates the tokens. In Go, this happens by using the packages `go/scanner` and `go/token`. Each of the generated tokens correspond to a construct in the input program such as statements, identifiers, literals, keywords, etc. Hence, the tree will be composed of tokens generated from statements and expressions of a program. For example, consider the code fragment that creates the space `"Store"` like this: `var Store = NewSpace("store")` The tokens that represent the above statement (from left to right) are: *token.Var, token.Ident, token.Assign, token.Ident, token.Lparen, token.String and token.Rparen*.

2. *Parsing* - the tokens generated from an input program are fed to a parser package in the Go's standard library, namely `go/parser`. This parser package checks the syntactic structure of the parsed input program file and presents it in a form of a tree, which contains all information of the input program.

3. *Presentation* - the parsed input program file is printed in the form of an abstract syntax tree by using the go packages, namely: `go/ast` and `go/printer`.

We consider a simple example of Go program that increments the given integer value by 5; it is shown in Listing 2.5.

```
1  package main
2  import "fmt"
3
4  func main() {
5    a := 10
6    incr := func(x int) int {
7      fmt.Printf("increments %d by five : ", x)
8      return x + 5
9    }
10   b := incr(a)
11   fmt.Println(b)
12 }
```

LISTING 2.5: An example Go program

The first step is to parse the program file into an AST. For this purpose, the `go/parser` package is used (see Listing 2.6). The `fset` variable holds tokenisation information about the files being parsed. The file variable is the root node of the AST. The final line of the code prints an AST to the standard output.

```
1 fset := token.NewFileSet()
2 node, err := parser.ParseFile(fset, "program1.go", nil, parser.ParseComments)
3 if err != nil {
4     log.Fatal(err)
5     ast.Fprint(os.Stdout, fset, node, nil)
6 }
```

LISTING 2.6: Parsing a given Go program

Line 1 uses the package `go/token` to create a new `FileSet` that represents the file of a source code parsed to generate an abstract syntax tree. The `parser.ParseFile` with the parameter `parser.ParseComments` at line 2 parses a given program.
`program1.go` is the file name which is opened when the src is nil. If there is no error in the parsed source file, then an `*ast.File` is returned.

The second step is to print the AST nodes from the parsed input program file, which is performed at line 5. The AST of the code fragment in Listing 2.5 is shown in Figure 2.5.

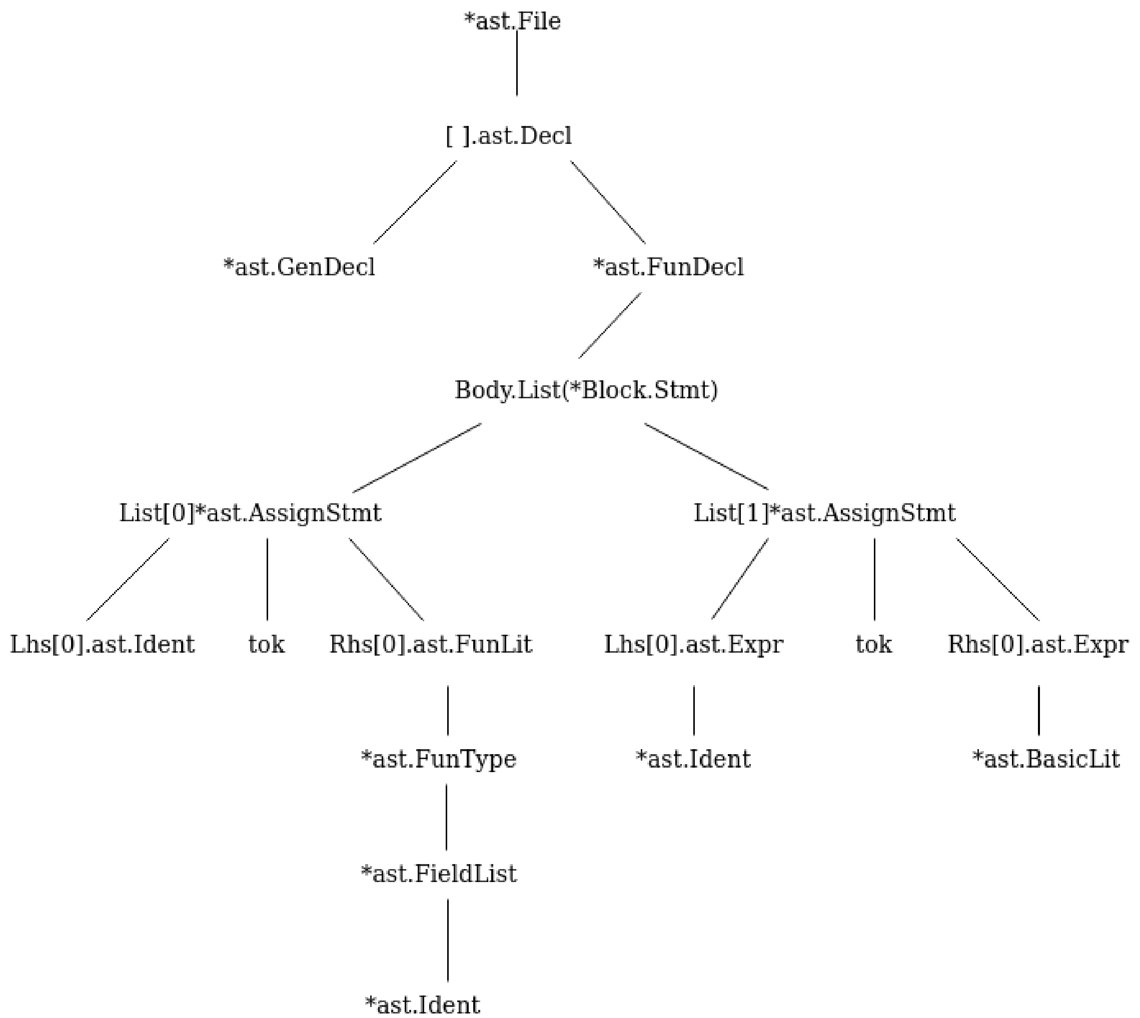


FIGURE 2.5: An example of abstract syntax tree

The goSpace program presented in Listing 2.7 gives the AST shown in Figure 2.6.

```
1  package main
2  import (
3    . "github.com/pspaces/gospace"
4  )
5  func main() {
6    store := NewSpace("tcp://host:123/store")
7    go process(&store)
8  }
9  func process(store *Space) {
10   store.Put("bicycle", 200)
11 }
```

LISTING 2.7: A Go Program using goSpace

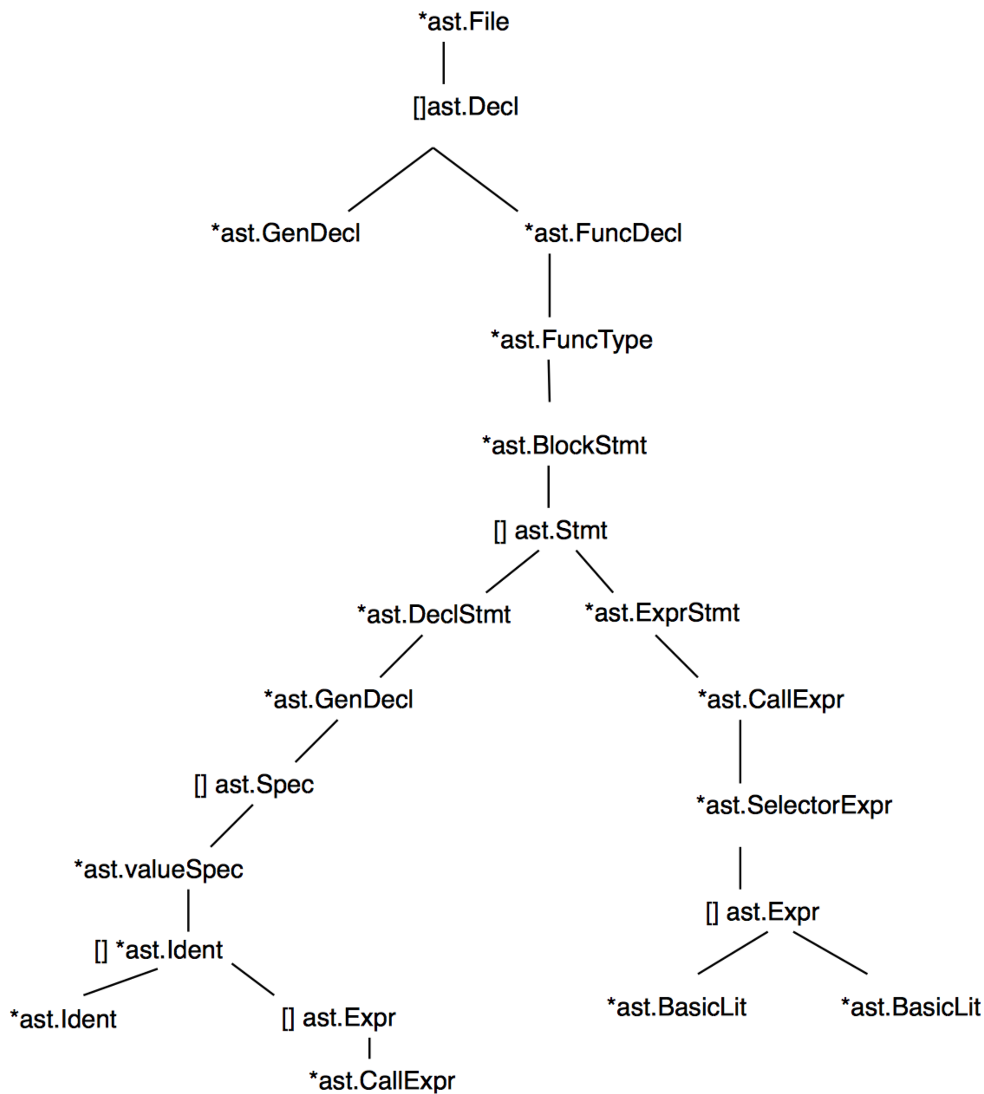


FIGURE 2.6: An example of AST for the source code in Listing 2.7

2.11.3 Related Work on Static Analysis

In this section, we shall discuss the research works integrating static analysis techniques with the coordination languages and highlight other research area. Some of the following research works directly inspired ours. Static analysis is used in many coordination contexts. In [100], Obreiter et al. exploit the static analysis and propose a tuple-space based coordination model that achieves scalability. They also suggest an approach based on mapping the tuples to frequency; they are used in the write and read operations in order to distribute the tuples on several servers.

Static analysis techniques are used in Klaim based coordination languages [101–103]. Our static analyzer is implemented in Go and is used to analyze Go programs written in goSpace coordination language. The purpose was to control the kind of operations that processes can perform at different localities, to preserve secure access to tuple spaces and safe process migration. Our focus, however, is on improving the performance of distributed systems that use tuple spaces, and reducing access latency.

Static analysis approach for Klaim language is also proposed to solve the licence conformance problem [104], and to validate that the client code conforms to the security policy. Bodei et al. [105] proposed a control flow analysis technique that over-approximates the behaviors of Klaim processes to track the propagation of tuples and identify their possible trajectories within a Klaim network. More specifically, the technique makes it possible to detect a priori, how the tuples can move in the Klaim network and when they can safely traverse a path or their move to a specific node may be dangerous. Furthermore, it supported reasoning about the transformations applied to a selected datum along those paths. In our work, we focused on computing the over-approximated locations for replication of tuples by checking the matches of write operations with the read and remove operations. In [106, 107], the static control and data-flow analyses have been exploited in Klaim language for enforcing security policies (e.g., prevents processes running on another node from sending a fake message) in distributed tuple space systems. However, they rely on the users ability to enforce the security policies, by specifying the access controls that static analysis cannot accomplish. Therefore, by applying the dual static-dynamic checks, type checking can provide guarantees about the processes allowed to proceed.

IoT-LySa [108] is a language based on a process calculus with a control flow analysis to show which data items a node can receive and from which nodes. The underlying ideas are to track and predict possible trajectories of data communicated in an IoT system, and check whether sensitive data can move via possibly unsafe nodes. Our static analyser, however, checks tuples operations, the number of its arguments and data types to track which tuples to replicate and where to replicate them.

Diwan et al. [109] proposed and evaluated three versions of alias analyses based on programming language types to disambiguate memory references. Two of these versions determined the aliases by relying on the declarations of types and field names, thus supporting the refinement of the types of objects that an access path may refer to access the memory locations. The other version has improved the analyses by using the

flow insensitive algorithm to include the effects of variable assignments and references. They statically and dynamically evaluated the proposed alias analyses with respect to redundant load elimination. Our work mainly differs from their work with respect to the target programming languages, redundancy elimination, dynamic analysis metrics and optimization of read-write access for replication.

Another area of related work involves verification to augment the techniques or the tools. Static verification of concurrent Go programs for bounded liveness and safety has been considered in [110–112]. Static verification approach that uses some form of regular expressions is proposed [113] for detecting deadlocks in Go programs. Like our research work, they considered a fixed number of processes and synchronous communications. Bounded analysis of concurrent programs for safe replication has been proposed [114], and maintained various forms of weak consistency levels, but did not guarantee all replicas to have a strong consistency level.

2.12 Program Transformation

This section briefly describes program transformation as it plays a key role in the research work presented in this thesis.

Source-to-source transformation, sometimes also referred to as code refactoring, has been introduced four decades ago (i.e., in 1977) for improving the maintenance of source code in a given input program [115].

Program transformation is the action of changing one program into another program. This action involves designing and implementing a set of modification rules that a computer performs on one or more constructs in the source code of an input program into one or more constructs of a target program. The source and target languages, respectively, are the languages in which the transformed program and its resulting output language are written. The transformation systems can be divided into three classes [116]:

1. *Manual*: The user is responsible for each and every stage of the transformation process. More specifically, the user should select a transformation rule that he wants to apply to a system from a set of available ones.

2. *Fully automatic*: aims at fully automating the transformation systems completely without relying on the user's ingenuity. It is possible to fully automate the transformation of a specification into another one that can be effectively executed on a computer once it is written and formalized.

3. *Semi-automatic*: aims at simplifying as much of the transformation process as possible while still allowing programs to be transformed within the paradigm.

Program transformation can be classified into two categories: rephrasing and translation. In the first category, that is the rephrasing, an input program is changed into a different output program in the same language, i.e., source and target programs or languages are the same. For the second category, that is the translation, an input program is changed into an output program in different target language, i.e., source and target programs are different. Their use in software development activities and static analysis that revolve around changing the source codes and analyzing their behaviors are justified by works that have shown that software performances and quality can be achieved [117–119]. In this thesis, we only apply translation as our research work fits into that category. A reader interested in the strategies of program transformation is referred to [120, 121] for more details.

2.12.1 Related Work on Program Transformation

There have been research works aiming at applying source code transformation approaches in the area of coordination languages and other several contexts.

The combination of tuple space-based computing and transformation is explored and is at the core of the Linda coordination model. It has effectively improved the performance of concurrent systems [122]. GrGen.NET [123] is a tool for the graph transformations, it allows users to work with declarative pattern matching and rewriting rules, and facilitates the modification of graph-based representation. Our focus, however, is on automated transformations of Abstract Syntax Tree, and optimized by statically analysing the data access patterns of the given programs.

Different source code transformation techniques have been proposed in several contexts, including example-based [124] or rule-based [125, 126] transformations. Implementations of rule-based transformations are available in Haskell [127] and Stratego/XT [128]. Our research work, however, introduces a pattern-based transformations approach restricted to Go.

Several source transformation frameworks for different languages are available. A popular source framework for C and C++ is ROSE [129]. Transformation tools developed within the ROSE framework are based on directly modifying the syntax tree and then un-parsing it to obtain the modified program. Another transformation framework for C and C++ programming languages, widely adopted for software verification, is the Clang compiler framework [130]. Source transformation in Clang is obtained by directly altering the relevant fragments of the initial source code, because modifying the abstract syntax tree of the program is not allowed. In addition, Clang provides libraries with the features that support the source code rewriting transformation (like refactoring) [131].

Other authors proposed transformation tools [132, 133] which build an abstract syntax tree of the source code of programs and use the predefined rewrite rules to manipulate the Abstract syntax tree. The Proteus system [134] uses an abstract syntax tree approach to perform transformations on the large C/C++ systems. The rule based and phasing based mechanisms for program transformation are proposed [135]. The transformation process uses abstract syntax trees organized in a set of phases and each phase consist of a set of rules that are executed by the transformation engine to perform a certain task.

Various transformation tools applicable to multiple programming languages have been proposed. TXL [136] relies on the user's ability to specify the grammar and transformation rules in the TXL programming language. In [137], Van Tonder and Le Goues propose Comby, a rule based transformation framework applicable to multiple languages. It requires users to write the transformation templates and changes the code in multiple languages. Our approach differs from these approaches in terms of relying on users ability to specify and write the transformation rules. In other words, the pattern matching mechanism used in the source code differs from ours; it does not consider data access patterns but it relies on user-specified match templates.

In [138], Koppel et al. propose Cubix, a source code transformations tool applicable to multiple languages. The proposed approach relies on the ability of users to define and write the transformation patterns once, which can be applied to programs written in many programming languages. Our approach, however, does not require users to define and write transformations patterns. We focus on the transformation of Go programs written in goSpace coordination language.

Chapter 3

RepligoSpaces: goSpace with Replicas

In this chapter, we present our tuple-based coordination language *RepligoSpaces*, which enriches goSpace with the primitives for replica-aware coordination. The extended coordination primitives can be used to automatically replicate tuple spaces and guarantee strong consistency level. It borrows the concepts of spaces (i.e., *pSpaces*) or localities like in Klaim styles.

This chapter is structured as follows. Section 3.1 describes some concepts of tuples and templates. Section 3.2 presents the syntax of RepligoSpaces. Section 3.3 presents the informal semantics of RepligoSpaces. Section 3.4 presents how the replica-aware coordination primitives of RepligoSpaces are implemented.

3.1 Tuples and Templates

Tuples and Templates (or patterns) are fundamental elements adopted by many tuple spaces-based coordination languages. The definitions of tuples and templates in RepligoSpaces are similar to the ones of goSpace (Klaim) coordination language. We recall that the concept of spaces used in RepligoSpaces is similar to the one of locations used in RepliKlaim. Both spaces and locations are used to store tuples.

The components of tuples are expressions. *The tuples* are finite sequences of information items of (only) actual fields that are used to elaborate data items and exchange the

information. Tuples can also contain the variables and are called *templates* (or *patterns*). In fact, pattern `T` can contain both actual and formal fields, and are used to select tuples from the spaces.

An example of tuple is `("Hello",2021)`, a sequence of two actual fields of types string and integer, whereas examples of patters are `(!string,!int)` and `(!int,value)`.

Templates are used to select the tuples, and this process is done via the pattern matching mechanism. This mechanism is defined by the pattern-matching predicates similar to the one of the Klaim coordination language presented in Table 2.3 of Chapter 2.

3.2 RepligoSpaces Syntax

This section focuses on the syntactic features of RepligoSpaces. RepligoSpaces is based on the notion of *space* and relies on the Klaim-like coordination language. The syntax of RepligoSpaces provides replica-aware programming routines.

The space can be `S`, a set of target spaces (ranged over $s_1, s_2, \ldots, s_n$), where replicas of tuples are stored, or `s`, one of the elements of the spaces, where tuples can also be stored. In other words, `S` is used to denote the set of spaces and $(s_1, s_2, \ldots, s_n)$ to denote its elements. The elements of the set of spaces ($S = s_1, s_2, ..., s_n$) may represent the set of target spaces for replicating tuples. An illustrative example of replication strategies of the tuples over the spaces and additional details will be provided in the next chapter.

Actions performed by processes permit to write tuples to the set of target spaces `S` via `MPut` action, read the tuple matching the given patterns via the `MQueryP` and `MQuery` operations, and remove the tuples and its replicas from the spaces via `MGetP` operation. In concordance, expressions denoted by `e` represents the constant `c`, the basic program variables `x` and values `v`. Tuples `t` are the sequences of actual fields, i.e., values or expressions. *Templates* (*patterns*) are the formal fields, which means that certain values of these fields are not known. Hence, the formal fields are used to build templates. Tuples to be placed in the tuple spaces can only have actual fields, whereas templates can contain both actual fields and formal fields.

RepligoSpaces consists of operations that extend goSpace's operations (Section 2.7.2) with replica-aware tuple manipulation primitives. In fact, these operations are implemented to support the replication of tuples across multiple spaces. RepligoSpaces actions can be used to add tuples to the spaces, and to retrieve and remove tuples from them. The functionality of each RepligoSpaces primitives is described as follows:

- The `MPut`$(t, s_1 \ldots s_n)$ is the non-blocking operation that adds a tuple to a set of spaces. When multiple spaces are provided as the target destinations, the tuple is replicated in all of them.
- The `MQuery`(T, s) operation searches from the space s, a tuple that matches the given pattern T. If the matching tuple does not exist, the operation blocks until a tuple is found.
- The `MQueryP`(T, s) operation queries a specific space s for tuples matching the pattern T. It returns the found tuple, if any and an empty tuple when none is found.
- The `MGetP`(T, s) operation searches for a tuple that matches pattern T and removes it from its space and from any other space where it was previously replicated.

RepligoSpaces operations and their goSpace counterparts are shown in Table 3.1. Blocking operations are marked with the symbol (*).

Description	**goSpace**	**RepligoSpaces**
Add a tuple	`s.Put`(t)	`MPut`$(t, s_1 \ldots s_n)$
Search and read a tuple*	`s.Query`(T)	`MQuery`(T, s)
Search and read a tuple	`s.QueryP`(T)	`MQueryP`(T, s)
Search and remove a tuple	`s.GetP`(T)	`MGetP`(T, s)

TABLE 3.1: Basic operations on tuple spaces in goSpace and RepligoSpaces

Note that we have not considered `Get` primitive (i.e., a blocking operation to remove the tuples from the spaces) in RepligoSpaces like in goSpace (see Table 2.8); this design choice was considered in order to avoid deadlocks. Assume that a process gets a lock; in case there is no tuple matching the given template, the process will not release that

lock. Hence, other processes are blocked and cannot do any action. Therefore, we have not considered the `Get` primitive to avoid deadlocks. However, the lack of `Get` primitive does not affect the expressiveness of the language.

3.3 Informal Semantics of RepligoSpaces

This section describes the semantics of the coordination primitives of RepligoSpaces in an informal way.

In RepligoSpaces, the system is formed by an ordered sequence of processes and spaces. Every space has an index or an identifier for each process. We chose to have an ordered sequence of spaces because in this way every space can have a specific identification index. Processes can be executed in an interleaving fashion. RepligoSpaces provides coordination primitives that enable processes to add tuples (`MPut`) to the spaces, read (`MQuery` and `MQueryP`) and remove (`MGetP`) tuples from the spaces using the templates.

Tuple and templates have to be evaluated before they are used to remove or read the tuple from the spaces. The read and remove operations are based on a mechanism that requires matching tuples with template. A template matches a tuple if:

1. Both the tuple and the template have the same number of fields, and all pairs of fields at the corresponding positions do match;
2. The fields of tuple and the corresponding pattern have the same types;

Note that, the read and remove operations only succeed if the tuples match the given templates in the given spaces. Otherwise, if the matching of tuple with the template does not succeed, a *nil* tuple is returned to the process executing an action.

MPut deals with a strong output action `MPut(t)@S` and is used to add a tuple in every space of the set of all target spaces. As a result, every space in the set of spaces will contain a replica of the tuple.

MGetP deals with the strong input action `MGetP(T,s)` and is used to remove a matching tuple from its own space, and from all spaces containing its replica. If a copy of a matched tuple is found in one local space, it is removed from that space and from all spaces that contain its replica. However, if the tuple is not found in any of the considered spaces, a nil tuple is returned to the process willing to remove the evaluated tuple.

MQueryP deals with the non-destructive read action `MQueryP(T,s)`. It reads a tuple that matches the given template from its own space within the set of all spaces. If a process does not find any tuple matching a given pattern in any space (i.e., the matching of tuple with pattern does not succeed). The process that is willing to read the evaluated tuple gets a *nil* (or *empty)* tuple. When the matched tuple is accessed and retrieved, it is left in its own local space (differently from the operation *MGetP*).

Finally, *MQuery* is used to read the tuple that matches the given pattern from its own space within the set of all spaces. Once the tuple is found and accessed, it is not removed from its own space. Note that the space may not react to the searching of tuple (i.e., when the tuple is not found); in this case the process that is willing to read the tuple from the space would be blocked.

3.4 Prototype Implementation

In this section, we present RepligoSpaces implementation, our extension of goSpace [26] to replica-aware programming. The choice of using goSpace is motivated by the rich set of features of the host programming language *Go*.

Both pSpaces and goSpace (Section 2.7) allow one to manipulate tuples within a single space. With RepligoSpaces, we instead are able to manipulate tuples across multiple spaces transparently. Our extension follows the same approach of RepliKlaim with Klaim (see Sections 2.3 and 2.5, respectively). The differences between RepliKlaim and RepligoSpaces are that, in RepliKlaim, we can specify the level of consistency for the operations, while in RepligoSpaces, we have opted for a specific consistency model (strong consistency). Moreover in/get operations take a list of locations as parameters, not just one. Indeed, we considered Repliklaim's approach with respect to Klaim for automatically replicating tuples by using RepligoSpaces, but our linguistic approach relies on an implementation based on goSpace differently from that of RepliKlaim, implemented in Java. Moreover we have used a lightweight static analysis technique, and the two coordination languages are slightly syntactically different. We can only say that Repliklaim relates to Klaim just like our RepligoSpaces relates to goSpace.

In what follows, we will provide a detailed description of the way RepligoSpaces operations are implemented. They are instrumental to support the manipulation of tuples on

multiple spaces. More concretely, goSpace as described in Section 2.7 of Chapter 2, allows one to manipulate tuples within a single space. Here we extended it to manipulate tuples within multiple spaces while preserving the strong consistency level.

The `MPut` action shown in Listing 1 writes a tuple to a set of spaces. It takes as input a tuple `t` and a set `S` of space identifiers, in the form of strings that encode their URIs.

```
1 func MPut(t Tuple, Sp Replispace, S []string) Tuple {
2     Sp.mux.Lock()
3
4     // create tuple t' = {t,S}
5     var data []interface{}
6     data = append(data, t.Fields...)
7     data = append(data, S)
8     var t1 Tuple = CreateTuple(data...)
9
10    // add t' to each space in S
11    for i := 0; i < len(S); i++ {
12        Sp.Sp[S[i]].Put(t1.Fields...)
13    }
14
15    Sp.mux.Unlock()
16    return CreateTuple(t1)
17 }
```

Listing 1: The `MPut` operation replicates a tuple over a set of spaces

The idea is then to simply perform a normal goSpace `Put` operation for every space in `S` (lines 10–13). To do so, we need a reference to the space object identified by the *URI* at any given position of the set `S`. To this purpose, we use a global map `Sp` from URIs to references to `space` objects. Note that this is not a limiting factor as our source transformation procedure will automatically populate `Sp` for us (Section 4.1.3). Please notice also that the actual tuple being stored is not `t`, but an extended tuple obtained by appending `S` to `t` (lines 4–7). This avoids centralized tracking of the storage locations [6] and simplifies the implementation. We are interested in strong consistency, thus the sequence of `Put` operations is enclosed in a critical section (lines 2 and 15) to enforce atomicity.

Note that the data items or tuples to be used in the given spaces, may have different data types, repligoSpaces *struct* is defined with two fields: `Mux` and `Sp[...]`. In accordance with this, all fields of different types are combined and associated to a specific data type (e.g., Space, etc). Therefore, these fields can be accessed along the created goroutines. For the same reason, we used `Sp.Sp[...]` (see line 12 of Listing 1) to ensure that all elements (or fields) are selected, including the spaces list added for indicating where tuples are stored (as the tuple is extended by appending the spaces list to the original tuple).

```
1 type Replispace struct {
2     mux sync.Mutex
3     Sp  map[string]*Space
4 }
```

The MQueryP operation illustrated in Listing 3.2 searches the given space for tuples matching the given pattern. It takes as input a tuple `p` (i.e., a pattern) and a space identifier `s`, and returns as output a tuple, if any. As a result of the previous `MPut` operation as described above, every stored tuple is extended with an extra field that contains the set of target spaces. Therefore, our search pattern `p` will need to be adapted accordingly by appending to `p` an extra field to be used as a placeholder to match the set of targeted spaces in the last field of any stored tuple (line 5 and lines 6–9). The modified pattern `p1` so obtained is used instead of `p` to retrieve matching tuples at space `s` (line 12). On a successful search (lines 14–19), the last field of the returned tuple is removed as no longer relevant (line 16), and the tuple originally stored is returned. Otherwise, an *empty* tuple is returned (line 22).

```
1 func MQueryP(p Tuple, Sp Replispace, s Space) Tuple {
2     Sp.mux.Lock()
3
4     // create template p' = {t,S}
5     var y []string // <--- extra field to match the space list S
6     var data []interface{}
7     data = append(data, p.Fields...)
8     data = append(data, &y)
9     var p1 Tuple = CreateTuple(data...)
10
11    // query a tuple via a pattern matching from a specific space
12    t1, e := s.QueryP(p1.Fields...)
13
14    if e == nil {
15        // no error: return the matching tuple without the last field
16        var u = CreateTuple(t1.Fields[:len(t1.Fields)-1]...)
17        Sp.mux.Unlock()
18        return u
19    }
20
21    Sp.mux.Unlock()
22    return CreateTuple()   // returns an empty tuple when no tuple is available
23 }
```

Listing 3.2: The `MQueryP` operation to search for a replicated tuple

The MQuery operation illustrated in Listing 3.3 is similar to the implementation of `MQueryP` operation, except that it blocks until a tuple is found. More specifically, it searches the given space for tuples that match the given pattern, and blocks if the tuple is not found (line 22). As a result of the previous `MPut` operation as described above, every stored tuple is extended with an extra field that contains the set of target spaces. Therefore, our search pattern `p` will need to be adapted accordingly by appending to `p` an extra field to

be used as a placeholder to match the set of targeted spaces in the last field of any stored tuple (line 5 and lines 6–9). The modified pattern `p1` so obtained is used instead of `p` to retrieve matching tuples at space `s` (line 12). On a successful search (lines 14–19), the last field of the returned tuple is removed as no longer relevant (line 16), and the tuple originally stored is returned. Otherwise, it blocks until the tuple is found (line 22).

Note that we have relied as much as possible on the functions of goSpace (Section 3.2). The difference between the blocking and non-blocking operations is at the level of replica-aware routines. Our RepligoSpaces blocking operation (like `MQuery)` will invoke the blocking version (like `Query`) of the corresponding function in goSpace. Therefore, we didn't have to implement the notification of a waiting mechanism for the `Query` operation.

```
1 func Query(p Tuple, Sp Replispace, s Space) Tuple {
2     Sp.mux.Lock()
3
4     // create template p' = {t,S}
5     var y []string // <--- extra field to match the space list S
6     var data []interface{}
7     data = append(data, p.Fields...)
8     data = append(data, &y)
9     var p1 Tuple = CreateTuple(data...)
10
11    // query a tuple via a pattern matching from a specific space
12    t1, e := s.Query(p1.Fields...)
13
14    if e == nil {
15        // no error: return the matching tuple without the last field
16        var t2 = CreateTuple(t1.Fields[:len(t1.Fields)-1]...)
17        Sp.mux.Unlock()
18        return t2
19    }
20    // blocks until the tuple is found
21    Sp.mux.Unlock()
22    return CreateTuple()
23 }
```

LISTING 3.3: The `MQuery` blocking operation for searching a replicated tuple

The `MGetP` operation illustrated in Listing 3.4 uses a pattern `p` to search and remove a matching tuple from space `s` and any other space where it was replicated. It returns as output the tuple, if any. As for the other operations, the pattern `p` needs to be adapted with an extra placeholder to match the set of target spaces appended to the stored tuples by the `MPut` operation. We can then use the modified pattern `p1` to scan space `s` for matching tuples (line 12). On a successful search (lines 14–35), we extract from the matched tuple, the set `S` of spaces holding a replica of the tuple (line 16). To perform a standard goSpace `MGetP` operation on every space in `S`, we use the map `Sp` to retrieve

a reference to the relevant space object identified by the URI in `S`, similarly to the procedure used to implement the `MPut` operation. Thus, upon searching for the matching tuples from space `s` (line 12), the list `S` of all spaces containing a replica of the matching tuple is extracted and transformed in the form of strings of spaces identifiers (lines 16–19). The loop (lines 22–34) performs a `GetP` operation for every space in the set `S` of space identifiers (line 24) using the map `Sp` and the modified pattern `p1`. On a successful search (lines 26–34), at the last iteration, the tuple is stripped from the extra field containing the target `URIs` and returned. Note that, since we are assuming only strong operations, it should not be possible for the search to be unsuccessful after passing the first check (line 14). Eventually the operation returns an *empty* tuple in case none is found (line 38).

```
1  func MGetP(p Tuple, Sp Replispace, s Space) Tuple {
2      Sp.mux.Lock()
3
4      // create template p' = {t,S}
5      var y []string // <--- extra field to match the space list S
6      var data []interface{}
7      data = append(data, p.Fields...)
8      data = append(data, &y)
9      var p1 Tuple = CreateTuple(data...)
10
11     //  search the tuple from space s
12     t1, e := s.QueryP(p1.Fields...)
13
14     if e == nil {
15         // extract the list of all spaces
16         var S = (t1.Fields[len(t1.Fields)-1])
17         // transform the interface type of spaces into the string type
18         var v []string
19         v = S.([]string)
20
21         // for each space in the set S of space identifiers
22         for s := range v {
23             // remove the tuple from the relevant spaces
24             u, e1 := Sp.Sp[v[s]].GetP(p1.Fields...)
25
26             if e1 == nil {
27                 if s == len(v)-1 {
28                     // no error: tuple successfully removed from the space
29                     u = CreateTuple(u.Fields[:len(u.Fields)-1]...)
30                     Sp.mux.Unlock()
31                     return u
32                 }
33             }
34         }
35     }
36
37     Sp.mux.Unlock()
38     return CreateTuple()   // returns an empty tuple when no tuple is available
39 }
```

Listing 3.4: The `MGetP` operation for removing a replicated tuple

Note that, to ensure that the `MGetP` operation (Listing 3.4) deletes the right tuple and its replicas, we do not first perform the removal of tuples. In fact, we first search for a tuple (via `MQueryP`) matching a given pattern (line 12) and, when it is found, we use the returned tuple as a search key (as a pattern) to find other spaces where it is placed (line 16). In fact, when we store the tuple and its replicas, we also keep track (as part of the tuple) of the spaces where they are copied. At the end of this process, we remove the tuple and the replicas from their spaces (lines 22–34). In this way, we ensure that when there are multiple matching patterns, our prototype tool still removes the right tuples from different spaces and avoid compromising consistency.

Since, we do not know exactly how many elements or data items can be in a tuple in a space, to avoid loosing all fields of a tuple, we use a function `t.Fields` (or `p.Fields` for a pattern). For an example, see line 6 of Listing 1 or line 7 of Listing 3.2) where we accommodate any number of elements (or fields) of a tuple `t` that is being parsed. In fact, in Go it is possible to create a function with a variable number of parameters. In our case, the created function allows us to include a parameter (see line 7) which is in essence a tuple that can have many fields within a space (see lines (lines 5–9) in Listings 3.2, 3.3 and 3.4). For this reason, instead of parsing a tuple `t` as a tuple in all RepligoSpaces operations, we used the one (i.e., t_1) that can also contains many fields of tuples (see line 12 of Listings 1, 3.2, 3.3, and 3.4).

Furthermore, the communication primitives in RepligoSpaces only support replications that require a global lock because we wanted to achieve strong consistency level.

Chapter 4

Static Analysis and Program Transformation for Replication

Existing coordination languages for tuple spaces require manual programming effort to specify and coordinate replicas in order to enhance the performances of distributed systems while ensuring the desired consistency properties. These tasks are very expensive and time-consuming, motivating research to find the appropriate solutions to handle automated data replication. In Chapter 3, we presented our first contribution, a linguistic approach, that is, the replica-aware implementation of Klaim coordination language in the Go programming language, built on top of goSpace that we call *RepligoSpaces*.

In this chapter, we present the automated replication technique based on static analysis and program transformation and discuss how RepligoSpaces, described in Chapter 3, fits within a fully-mechanisable procedure for automated replication of programs over tuple spaces that relies on combining static analysis and program transformation. We rely on a lightweight static analysis pass on the initial program to compute the sets of target spaces for replication, in order to replace the standard tuple manipulation operations with their replica-aware versions. The combined approach preserves strong consistency, thanks to a tracking mechanism embedded in the tuple manipulation operations and to the fact that set of target spaces is safely over-approximated.

4.1 Automated Replication of Tuple Spaces

In this section, we describe in details the overall procedure that leads to automatically replicating the tuple spaces and the way we have to developed the source-to source code transformations. Figure 4.1 shows the overall workflow of our approach.

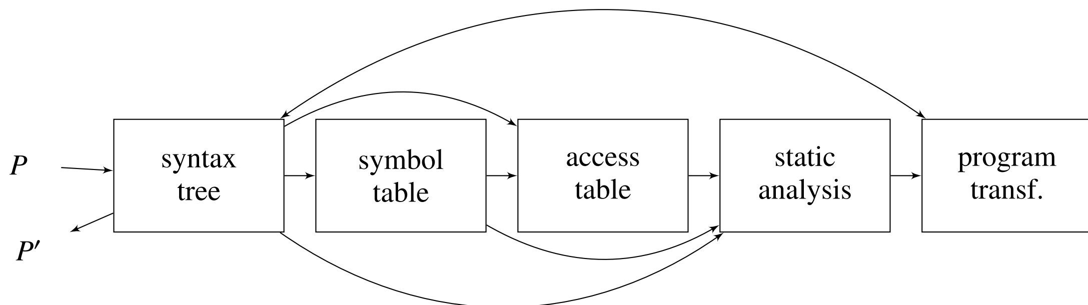


FIGURE 4.1: Static analysis and source transformation for automated replication

Our technique automatically transforms an initial (or input) Go program (P) that uses goSpace for data manipulation into an equivalent output program ($P^{'}$) (or Replicated version) where the communication primitives are transformed into the RepligoSpaces primitives. We consider two programs as equivalent when they compute the same functions even though they are syntactically different.

The first step deals with parsing the initial (or input) program P to generate an *Abstract Syntax Tree* (*AST*, for short). The syntax tree is recursively visited to generate the *symbol table*. Symbol tables maintain information about the identifiers of a program. This information is inserted when declarations of variables are analyzed. During the process of collecting information in the symbol table, we start visiting the body of the process definition functions, and then recursively nested blocks. As we go along, we assign unique identifiers to blocks so that as soon as a new variable declaration occurs in the syntax tree, that variable is added to the set of symbols for the current block; types of variables are also extracted from the syntax tree and stored in the symbol table.

The second step consists in building an *access table* by extracting information from the syntax tree and the symbol table. In particular, we visit the syntax tree again to detect all the operations on tuples (see Table 3.1 of Chapter 3). At the same time, we perform the symbol table lookups to figure out a type of fields for each tuple occurring as an argument for any of such operations. At the end of this process, we obtain for each tuple operation an actual tuple along with the type of each field of the tuple.

The third step consists in performing a *static analysis* by visiting the syntax tree a third time and combining information from the symbol table and the access table in order to over-approximate the set of target spaces for replication. At the end of the process, we obtain a modified program $\mathtt{P}'$ by un-parsing a modified syntax tree.

The final step consists in performing a *program transformation*. It takes as input, the initial program and data-access tables built via the static analysis passes described above, and generates a program where each tuple is replicated as indicated by the corresponding access lists. This can be done by parsing an input program into an abstract syntax tree, and then performing a series of pattern-based transformations on (parts of) this tree. It is worth observing that the input program may represent an *abstract model* of a more complex system whose computations that do not directly involve tuples are simply abstracted away.

In the processes definition functions $p_1, \ldots, p_n$ every call to a goSpace routine is transformed into a call to the corresponding extended primitive (Section 3.2) to achieve replication accordingly. For `MPut` operations, the set of target spaces for replication is added as an argument (e.g., cf. line 21 of Listing 4.1 and Listing 4.2). Each such set is over-approximated by the procedure described in the following section. Any other access operation, such as `GetP`, `Query` or `QueryP` (lines 31 and 40 of Listing 4.1) is instead changed to always refer to the local space.

4.1.1 Structures of Input and Output Programs

We provide a description of structures for the input and output programs.

Structure of an input program. Our automated replication approach starts with a step of defining an initial (input) program P. The input program is a Go program used in goSpace and represents the *model* for a distributed system. Each function except the main corresponds to a process. The output program is divided into four blocks of source code: declarations of import packages, declarations of global variables, main( ) function, and function definition. Specifically, we restrict our attention to the Go program used in goSpace and satisfies the following assumptions:

1. Any input program should contain the main function.
2. Any thread is spawned from the main function.
3. The number of spaces and threads are the same.

4. All spaces are created inside the main function.

Note that the above assumptions do not imply any loss of generality of the proposed replication approach. We have used them to simplify the implementation of our prototypes and to be able to perform program transformation.

The skeleton of an input program P is shown in Listing 4.1.

```
1  import (
2    . "github.com/pspaces/gospace"
3    ...
4  )
5
6
7
8
9
10 func main() {
11   s1 := NewSpace("tcp://localhost:123/s1")
12   go Process1(&s1)
13   ...
14 }
15
16
17
18 func process1() {
19   ...
20   s1.Put("Sport",2022)
21   ...
22
23 }
24
25
26 func process2() {
27   ...
28   var choice bool
29   var description string
30   s1.GetP(&description,&choice)
31   ...
32 }
33
34 .....
```

LISTING 4.1: Input program

An input program is thus composed of a set P of n parallel processes performing concurrent computations over a set S of n shared tuple spaces.

We assume that each process is defined by a separate and a unique *process definition function*, and that all such functions are collected into the input program.

We denote the process definition functions with $P = p_1, \ldots, p_n$. We also assume that the input program additionally contains a *main section* where all the shared tuple spaces

are created beforehand and associated to unique space identifiers, and that all processes are spawned as separate threads.

We denote with $S = s_1, \ldots, s_n$, the set of spaces shared among the processes, and associate to each process p_i a local tuple space s_i. We consider every tuple manipulation operation performed by a process to be a *local* operation if it refers to that space, and a *remote* operation otherwise.

Structure of an output program. The output program contains the declarations of global variables, the main function containing the spaces created, and the function declarations modelling the actions of processes. In addition, the output program contains the auxiliary data structures required by the RepligoSpaces replica-aware routines of Chapter 3, and in particular:

- `var uri = make(map[space]string)`: tracks the space created with a uniform resource identifier (uri), maps the `uri` from the spaces objects to their identifiers, and the space created is encoded as a string;
- `var sp = make(map[string]*Space)`: maps the space created and encoded as string to the relevant space object having as data type, a `Space`.

The output program (see Listing 4.2) retains the same structure as the input program (see Listing 4.1). The global section of the initial (input) program is extended with auxiliary data structures, such as the map `sp` from space identifiers to concrete references to space objects (line 12) and the map `uri` from space objects to space identifiers (line 13) (used for example in Listing 4.2). An additional package with the definitions of the extended (or replica-aware) tuple manipulation routines (Listings 1, 3.2, etc.) is added to the import section at the beginning of the output program (line 3) shown in Listing 4.2.

In the process definition functions $p_1, \ldots, p_n$ every call to a goSpace routine is transformed into a call to the corresponding extended primitive (Section 3.2) to achieve replication accordingly, and obtain an output program P' like the one shown in Listing 4.2.

```
1 import (
2   . "github.com/pspaces/gospace"
3   . "repligospaces"
4   ...
5 )
6
7 var uri = make(map[space]string)
8 var sp = make(map[string]*Space)
9
10 func main() {
11   s1 := NewSpace("tcp://localhost:123/s1")
12   sp["tcp://localhost:123/s1"] = &s1
13   uri[s1] = "tcp://localhost:123/s1"
14   go Process1()
15   ...
16 }
17
18 func process1() {
19   var choice bool
20   ...
21   MPut("Sport",2022,targets0)
22   ...
23
24 }
25
26 func process2() {
27   ...
28   var choice bool
29   var description string
30   MGetP(&description,&choice,uri[s3])
31   ...
32 }
33
34 ...
```

LISTING 4.2: Output program

4.1.2 Overapproximating the Sets of Target Spaces

In this section, we describe our static analysis technique, which over-approximates the sets of target spaces for replication of tuple spaces by checking the possible matches with the read and remove operations in the program. The static analyzer is entirely implemented in Go.

We aim at reducing unnecessary overhead by automatically inspecting the program to refine the set of target spaces. To that end, static analysis is used to extract from the initial program that uses goSpace coordination model, the data access patterns, and then use this information during a program transformation phase.

It is worth noticing that the extended tuple manipulation routines (Section 3.2) are independent from the specific technique used for reducing the set of target spaces for

data replication. In the following, we simply describe a lightweight static analysis technique for over-approximating such sets of target spaces. The goal of our static analysis procedure is to work out a refined set of target spaces, i.e., the *data-access tables*, for replicating the tuples while preserving strong consistency.

Let us consider a tuple `t` and a process p_i performing an output operation of `t` into a specific space s_j. The key idea of our approach consists in determining the set of processes $P' \subseteq P$ that can potentially perform a subsequent read operation on that tuple. We identify such processes by looking at the patterns used in the input operations within the corresponding definition functions, approximating the actual pattern matching mechanism of the normal tuple manipulation routines. In practice, when given on one hand an output operation and on the other hand an input operation, we check for a potential match between the tuple being stored and the given search tuple or template. We repeat this for every process except p_i and for every input operation in the corresponding process definition function, and we obtain P' by progressively excluding from `P` any process that is *definitely* not involved in an input operation that matches the tuple `t`. Eventually, the data-access table for replicating t will be the set $S' \subseteq S$ induced by P' on S.

For simplicity, let us assume that a field of a tuple `t` given as input to an `MPut` operation can be either a constant or a variable identifier, while a field of a pattern `p` taken by `MGetP` or `MQueryP` can be either a constant value or a formal field, namely the typed variable reference. We can provide an informal description of the used procedure.

The matching mechanism initially compares the number of fields of `t` and `p`: if they are different, then certainly there is no match; otherwise, there is still a possibility for `t` and `p` to match. The matching is then refined based on the actual fields of the tuple and the pattern, ignoring any formal fields or placeholders. A difference of any actual field at the same position of `t` and `p` indicates a mismatch. The matching is eventually refined again by taking into account the type of the formal fields of `p`. A type mismatch between the actual field of `t` (either a constant or a variable) and the corresponding formal field of `p` again means no match. It is worth noticing that combining the matching mechanism described above with the replica-aware routines from Section 3.2 preserves consistency, because

1. the matching algorithm only avoids replication for those spaces where a tuple is definitely never going to be accessed (i.e., no matching input operation for that tuple exists in the whole process definition function corresponding to that space),

and therefore the algorithm safely over-approximates the set of target spaces for replication;

2. the tracking mechanism embedded within the replica-aware tuple manipulation routines guarantees that when one copy of a tuple is removed all its replicas are atomically removed as well.

4.1.3 Program Transformation

In this section, we explain in detail our program transformation approach. More specifically, we present the transformation rules and the functionality of each of them, and some examples of how the proposed program transformation rules work. We released the open-source code of our replicator prototype, which is available online at the link: `https://github.com/Uwimbabazi/Replication/releases/tag/v1`.

The use of our program transformation approach can be summarised in three main steps:

1. Building an Abstract Syntax Tree: a given input program (i.e., non-replicated) is parsed to generate its abstract syntax tree (AST).

2. Transforming the AST: based on our transformation rules, the series of pattern-based transformations on the (part of) generated AST are performed. Therefore, the AST of a given non-replicated input program is transformed into its corresponding replicated version, where typically coordination primitives of goSpace are transformed into RepligoSpaces coordination primitives.

3. Providing the outcome: the transformed output program (RepligoSpaces program) is provided.

The *replicator* prototype tool that we propose in this section consists in splitting the above steps into simple ones implemented as a single *module*.

In general, every program transformation process is composed of either one or a combination of two important actions like removing the terms from or adding the terms to the programs. In the context of this research work, we focus on the additive program transformation instead of the removal ones. More concretely, we consider program transformation primarily as a targeted mechanism for replicating tuples while relying

on the target spaces refined by the static analysis.

Below we explain in detail our program transformation approach.

Suppose we have a grammar parser for the Go programming language that produces the types used to represent abstract syntax trees for the Go programs[1]. Our program transformation approach consists of automatically parsing an initial (input) program that uses goSpace to build its AST, and then visiting the generated AST to un-parse it back and generate the transformed input program `P`, which is the output program $\mathtt{P}'$ (i.e., RepligoSpaces program). In fact, the transformed program is obtained by modifying the behaviours of AST visiting process in a such way to produce the output program.

In the first step, the input Go program that uses goSpace `P` is specified. We used the `go/parser`[2] and `go/token`[3] packages to automatically parse the given input program and generate its abstract syntax tree.

In the second step, the generated AST of the given input program is visited, and a series of pattern-based transformations are applied to this tree, then un-parsed it back to generate its transformed output program. This mechanism is performed by overriding the Go parser's AST-based pretty-printer.

In the final step, the transformed output program (i.e., RepligoSpaces program) is provided. This is accomplished by using the `go/printer`[4] package.

In the following, we describe our program transformation rules that allow us to build a complete replicator prototype, thus supporting an automated replication of tuple spaces.

The program transformation rules are shown in Table 4.1, and have guided in the implementation of our replicator prototype that replicates tuples spaces by transforming goSpace programs into RepligoSpaces programs.

[1]https://golang.org/pkg/go/ast/
[2]https://golang.org/pkg/go/parser/
[3]https://golang.org/pkg/go/token/
[4]https://pkg.go.dev/go/printer

goSpace	**RepligoSpaces**
(1) `space.Put(Tuple)`	`MPut(Tuple, space`$_1$`..., space`$_n$`)`
(2) `space.Query(T)`	`MQuery(Template, uri[space])`
(3) `space.QueryP(Template)`	`MQueryP(Template, uri[space])`
(4) `space.GetP(Template)`	`MGetP(Template, uri[space])`
(5) `import (."gospace" )`	`import (."gospace", ."repligospaces")`
(6) `var space Space`	`var uri = make(map[Space]string` `var sp = make(map[string]*Space`

(7) `space := NewSpace("tcp://localhost:123/space")`	`space:= NewSpace("tcp://localhost:123/space")`
	`sp["tcp://localhost:123/space"] = &space`
	`uri[space] = "tcp://localhost:123/space"`

TABLE 4.1: Transformation Rules from goSpace to RepligoSpaces

Rule (1) `space.Put(Tuple)` → `MPut(Tuple, space`$_1$`..., space`$_n$`)` is used to model the behaviour of processes that replicate the tuples to the set of all target spaces.

Rule (2) `space.Query(T)` → `MQuery(Template, uri[space])` is applicable when transforming the read operation, where `MQuery` is a method name, `T` is a pattern and `uri[space]` is the specific space identifier that contains the matching tuple.

Rule (3) `space.QueryP(Template)` → `MQueryP(Template, uri[space])` is quite similar to the previous one (rule 2), but the actions of processes that read the tuples that match the given pattern `T` are non-blocking. It says that a non-blocking action `QueryP` is transformed into the new non-blocking `MQueryP` action. `MQueryP` is a method name which contains arguments: `T` as a pattern and `uri[space]` as the specific space identifier that stores the tuple found via pattern matching.

Rule (4) `space.GetP(Template)` → `MGetP(Template, uri[space])` specifies the transformation of a non-blocking removal of a tuple from its local space, where `GetP` is translated into the new non-blocking removal action `MGetP`.

Rule (5) uses the keyword "`import`" for importing a Go package, the command `import(."gospace")` → `import(."gospace",."repligospaces")` has a single import declaration for gospace package which is transformed into another import with two *importspecs* (i.e., gospace and repligospaces packages). As we built our approach on top of the RepligoSpaces, "repligospaces" package is imported to be used with the gospace's package. This transformation rule ensures that a combination of gospace and repligospaces packages can be used.

Rule (6) is applicable when transforming global declarations for auxiliary data structures of the created spaces. The built in `make` function is used to initialize a map. The space object identified by a unique uniform resource identifier `(uri)` is encoded as a string and associates the spaces to their specific URIs, and it maps `uri` to references to space object. The transformation process is performed by first preparing the required AST for the global variables (i.e., `uri` and `sp`) as written above, then iterated over the `FuncDecl` node that represents types of function declaration for the global variables. Finally, it appends the prepared sub-ast of the declarations of global variables to the list of existing one.

Rule (7) is applicable when transforming the `NewSpace` invocation into auxiliary data structures (see Section 4.1.1). A space can be created by using the method `NewSpace`. Each space is characterized by a logical address identified by a unique uniform resource

identifier (`URI`) encoded as a string and the name of the created space represents its logical address.

A simple example of program transformation

We now present a small example of source code transformation that replaces statements in the function declarations or the global variable declaration part of the program (shown in Listing 4.2). The program transformation is implemented by overriding the visit of function declarations that generates the source code from the AST-subtree representing declarations. In short, after building an AST-subtree for the given declarations of variables (lines 2–16), the body of the function declaration is visited in order to append the AST-sub tree for the variable to the list of declared variables. This transformation process for the source code in Listings 4.3 and 4.4 is shown in the source code fragments presented in Figure 4.2.

```
1 var s Space
```

LISTING 4.3: Before transformation

```
1 var uri = make(map[space]string)
2 var sp = make(map[string]*Space)
```

LISTING 4.4: After transformation

```
1 // AST for: var uri = make(map[space]string)
2   exmaptype2 := &ast.MapType{Key: identspace, Value: identstring}
3   expr := &ast.CallExpr{Fun: identmake, Args: []ast.Expr{exmaptype2}}
4   dec := &ast.ValueSpec{Names: []*ast.Ident{identurl}, Values: []ast.Expr{expr}}
5   gendecl = &ast.GenDecl{Tok: token.VAR, Specs: []ast.Spec{dec}}
6   // Sub-AST for: var uri = make(map[string]*Space)
7   value2 := &ast.StarExpr{X: identspace1}
8   // map[string]*Space
9   maptype := &ast.MapType{Key: identstring, Value: value2}
10  // make(map[string]*Space)
11  excallexpr := &ast.CallExpr{Fun: identmake, Args: []ast.Expr{maptype}}
12  // Sp = make(map[string]*Space)
13  dec21 := &ast.ValueSpec{Names: []*ast.Ident{identm}, Values: []ast.Expr{
      excallexpr}}
14  // var Sp = make(map[string]*Space)
15  gendecl1 = &ast.GenDecl{Tok: token.VAR, Specs: []ast.Spec{dec21}}
16 for _, f := range node.Decls {
17    list1 = append(list, f)
18    _, ok := f.(*ast.FuncDecl)
19    if ok {
20      continue
21    }
22    if !done {
23      list = append(list, gendecl,gendecl1)
24      done = true
25    }
26  }
27  node.Decls = list
```

FIGURE 4.2: Example of transformation of global variables in Listing 4.4

In the first step, we describe how to build the AST for the declarations of global variables presented in Listing 4.4. More specifically, in Figure 4.2, lines 2–5 specify how to build and transform the AST for the statement (line 1) in Listing 4.4.

In line 2, the `MapType` node represents a map type. It says that Map of the key `space` is associated to its value of type `string`, and generates as a result "`[space]string`".

Line 3 presents the node of type `CallExpr` (represents an expression followed by a list of arguments) that starts with the function expression `make`, followed by arguments having the map type with a key `space` and a value of `string` type (line 2), and it gives as results the expression "`make([space]string)`".

Line 4 presents the `ValueSpec` node that represents a variable declaration. It starts with identifier name `uri` followed by its values of type `CallExpr` (line 3) and then generates as results "`uri = make(map[space]string)`".

Line 5 shows the `GenDecl`(generic declaration) node that can either represent a constant, a type or variable declaration. It combines statements (lines 3–4) and starts with the token `var` followed by `Specs` which contains variable declarations from line 4, and then generates as results the declaration "`var uri = make (map[space]string)`".

Line 7 represents an expression `Space` of a pointer type and generates as results the sub-ast with the statement "`*Space`".

Line 9 shows an expression `Space` of map type which is associated to the string and generates as results the sub-ast with the statement "`map[string]*Space`".

Line 11 combines the results from lines 7–9. It starts with the function expression `make`, followed by the function arguments of a map type (line 9), and generates the sub-ast with the statement "`make(map[string]*Space)`".

Line 13 specifies a node of type `ValueSpec` and starts with *Names* that contains identifier `Sp`, followed by all expressions extracted from line 11 and provides as result, the sub-ast with the statement "`Sp = make(map[string]*Space)`"

Line 15 presents a node of type `GenDecl` (generic declaration) and starts with the token `var`, followed by a `Specs` that contains the `ValueSpec` (representing variable declaration or VarSpec) from line 13 and provides the final declaration of variable like "`var Sp = make(map[string]*Space)`"

Lastly, the loop (lines 16–27) arranges and appends all variable declarations (lines 5, 13 and 15) to a list of declarations in order to get the transformed output source code (lines 1–2) presented in Listing 4.4.

4.2 Illustrative Example

Listings 4.5 and 4.6 present the example of goSpace (i.e., non-replicated) and RepligoSpaces programs snippets which will serve to illustrate how some of the program transformation rules work. In fact, we transform an initial program to automatically achieve replication of tuple spaces, by converting all the operations of goSpace into calls to the new RepligoSpaces routines introduced in Section 3.2 of Chapter 3.

To see how pattern-based syntax tree transformations work, let us now consider a simple example with the function call at line 21 of Listing 4.5, where `process1` performs a `Put`

operation of the tuple `("A",10)` into the local tuple space `s1`. This fragment of code will trigger transformation because the referenced object (`s1`) is a tuple space (which is detected via a symbol table lookup) and the `Put` method is among the relevant ones (see Table 3.1). In the syntax tree, the corresponding subtree for the whole expression is therefore changed into a call to `MPut` (see Listing 1 of Section 3.4); new child nodes are appended to the function call node in the syntax tree for the extra parameters as shown in Listing 4.6. Un-parsing the syntax tree modified in this way will produce the transformed program.

Note that in our illustrative examples, we used `&desc` instead of `&string` because patterns can contain both the actual and formal fields. In fact, `&desc` is a template and `&string` is its type. In order to simplify the presentation and ensure that our approach works efficiently, we assume that the number of processes is equal to the one of spaces. Furthermore, all considered processes run in parallel.

As an example, we show how the procedure described above leads from the goSpace program of Listing 4.5 to the RepligoSpaces one shown in Listing 4.6.

The graphs that represent the data distribution for the initial program (see Listing 4.5) is shown in Figure 4.3 and the transformed program (see Listing 4.6) are shown in Figures 4.4a and 4.4b, respectively. Figure 4.4a represents universal replication and is included for comparison. In the figures, arrows from left to right indicate write operations; arrows from right to left indicate read operations.

Let us consider the tuple `("A",10)` stored by `process1` at line 21. The `GetP` operation at line 31 `process2` uses as the pattern a string constant and a formal field of integer type. Therefore the local tuple space `s2` is included in the set of spaces for replication of `("A",10)`. Note that the analysis is control-flow insensitive because the branch condition at line 29 is ignored.

Now let us consider `process3`. The size of the pattern given at line 40 and that of the tuple `("A",10)` under consideration match. The types of the last fields respectively of the tuple and of the pattern do not match (`bool` vs `integer`). Therefore, the tuple is not replicated to `s3`.

We now focus on the second tuple `(choice,10)` stored by `process1` at line 23. The type of the first field of the tuple is known, but its value depends on previous computations. The pattern used in the input operation in `process2` at line 31 does not match this type. The tuple is thus not replicated at `s2` or at `s3`.

Indeed, in the transformed program (Figure 4.4b), the only replicated tuple is `("A",10)` which is replicated to `s2` as it can potentially be accessed by `process2`. Note that there is no need to store this tuple to `s1` as no subsequent matching read operation within `process1` occurs. Note that, in general, this program transformation does not depend on the specific static analysis technique used to work out the set of target locations (i.e., shown as `targets0` and `targets1` in Figure 4.4b). The example of replication strategies is shown in Figures 4.4a–4.4b. Listing 4.5 presents the source code of an input program.

```
1  import (
2    . "github.com/pspaces/gospace"
3    ...
4  )
5
6
7
8
9
10 func main() {
11   s1 := NewSpace("tcp://localhost:123/s1")
12   go Process1(&s1)
13   ...
14 }
15
16
17
18 func process1() {
19   var choice bool
20   ...
21   s1.Put("A",10)
22   ...
23   s1.Put(choice,10)
24   ...
25 }
26
27 func process2() {
28   ...
29   if check {
30     var key int
31     s1.GetP("A",&key)
32   }
33   ...
34 }
35
36 func process3() {
37   ...
38   var choice bool
39   var desc string
40   s1.GetP(&desc,&choice)
41   ...
42 }
43 ...
```

Listing 4.5: Input program (Non-replicated)

The graph that represents data distribution in the input program (i.e., non-replicated program) is shown in Figure 4.3.

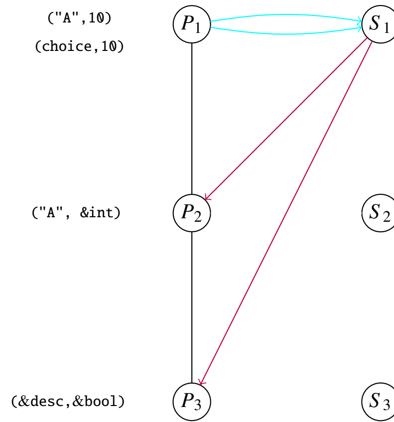
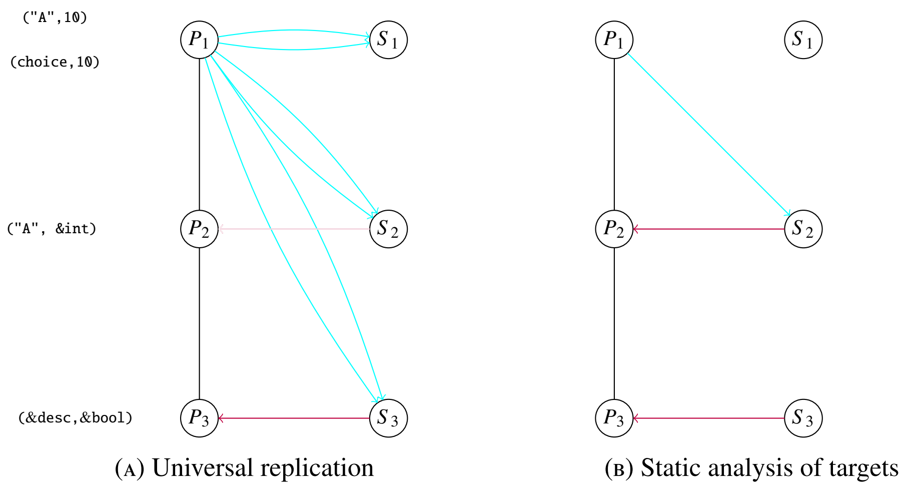


FIGURE 4.3: No replication

The graphs that represent the data distribution for replication strategies are shown in Figures 4.4a and 4.4b.



(A) Universal replication

(B) Static analysis of targets

FIGURE 4.4: Example replication strategies

The transformed program for the graph shown in Figure 4.4b is presented in Listing 4.6.

```
1  import (
2    . "github.com/pspaces/gospace"
3    . "repligospaces"
4    ...
5  )
6
7  var uri = make(map[space]string)
8  var sp = make(map[string]*Space)
9
10 func main() {
11   s1 := NewSpace("tcp://localhost:123/s1")
12   sp["tcp://localhost:123/s1"] = &s1
13   uri[s1] = "tcp://localhost:123/s1"
14   go Process1()
15   ...
16 }
17
18 func process1() {
19   var choice bool
20   ...
21   MPut("A",10,targets0)
22   ...
23   MPut(choice,10,targets1)
24   ...
25 }
26
27 func process2() {
28   ...
29   if check {
30     var key int
31     MGetP("A",&key,uri[s2])
32   }
33   ...
34 }
35
36 func process3() {
37   ...
38   var choice bool
39   var desc string
40   MGetP(&desc,&choice,uri[s3])
41   ...
42 }
43 ...
```

LISTING 4.6: Output program

Chapter 5

Evaluation of Implementation

This chapter presents an evaluation of our prototype implementations. More specifically, we describe the considered case study and how we conducted the experiments. We also present the experimental results obtained from conducting the experiments using the replication approach described in the previous chapters. The considered case study should help in understanding the effect of replication on distributed systems.

5.1 Case study

We considered a distributed system composed of n computational nodes, each executing a separate program, and interacting through a decentralized data store containing up to m elements. Following a similar schema to those used in distributed lookup protocols (e.g., Chord [139]), memory entries are represented as key-value pairs, with a partitioned address space among the nodes. Each node is responsible for storing m/n memory entries. A node reads from and writes to either its own local memory, or to that of another node, depending on the source or target memory address. Each node performs o operations, with p denoting the expected percentage of write operations.

For such a system, one might consider adopting a replication schema in the attempt to reduce non-local access (at the cost of additional local storage, plus some overhead for replication to non-local storage). To experiment with this idea, we model the nodes as separate processes, and the local memory of a node as the local tuple space of the corresponding process, with tuples ($address$, $value$) representing values held at different memory addresses.

5.2 Experimental Setup

- **Computing environment**. We conducted the experiments on a standard computer with macOS High Sierra 10.13, Intel Core i5 with 2 cores, running at 2.7 GHz with 8 GB.
- **Parameters used in the case study**. To evaluate the effect of replication on the system, we conducted different experiments with respect to the values shown in Tables 5.1 and 5.2. We denote by n, m, o and p the number of processes for each system, the overall memory size, the total number of (read and write) operations per each process and the expected percentage of write operations, respectively.

System Identification	Scenario	Parameters			
		n	m	o	p
1	1	4	32	16	{10, 20, ..., 90}
	2	4	64	16	{10, 20, ..., 90}
2	3	16	128	16	{10, 20, ..., 90}
	4	16	256	16	{10, 20, ..., 90}
3	5	32	256	16	{10, 20, ..., 90}
	6	32	512	16	{10, 20, ..., 90}
4	7	64	512	16	{10, 20, ..., 90}
	8	64	1024	16	{10, 20, ..., 90}

TABLE 5.1: Configuration of the system for non-local read or write operations

For each scenario of a system, we have considered 9 different configurations. For each combination of the chosen values for n, m, and p, we have generated 10 test cases (i.e., programs) with random data access patterns. We did run each test case 10 times. This led to a round of 100 runs for each configuration. We repeated each such round twice: once on the initial program, and once on the replicated program obtained with our tool, for an overall number of 1800 runs. We computed and compared the average values of remote accesses (read/write) of processes. Moreover, we considered all read operations to be `QueryP` and write operations to be `Put`.

The second test we have conducted on our implementation is calculating its latency (or execution time). More specifically, we focused on computing execution times of non-local read or write operations of processes. Similar to what we have described before, we again considered 9 configurations for each system, and all experiments were parameterized with respect to the values shown in Table 5.2.

System Identification	Scenario	Parameters			
		n	m	o	p
1	1	4	32	16	{10, 20, ..., 90}
	2	4	64	16	{10, 20, ..., 90}
	3	4	128	16	{10, 20, ..., 90}
2	4	32	256	16	{10, 20, ..., 90}
	5	32	512	16	{10, 20, ..., 90}
	6	32	1024	16	{10, 20, ..., 90}
3	7	64	1024	16	{10, 20, ..., 90}

Table 5.2: Configuration of the system for computing execution times of non-local read or write operations

We conducted additional experiments on our prototype implementation where we computed execution times (in seconds) of non-local accesses (read/remove) of processes with and without replication. We considered all write operations to be `Put`, read operations to be `QueryP` and remove operations to be `GetP`.

For this set of experiments, we have again considered 9 different configurations. For each configuration, we generated 10 test cases (or programs) with random data access patterns, then ran each test case 10 times that led to rounds of 100 runs each. We repeated each round twice: once on the initial program, and once on the replicated program obtained with our prototype implementation, for an overall number of 1800 runs for each of the four considered systems.

We fixed the number of processes where $n \in \{4, 16, 32, 64\}$, but varied overall memory size where $m \in \{64, 128, 256, 512\}$. For all these systems, we set the number of operations per process, $o = 16$ while varying the expected percentage of `Put` operations, where $p = \{10, 20, \ldots, 90\}$.

5.3 Experimental Results

In this section, we investigate the effectiveness of our replication approach by analyzing, and commenting our experiments, aiming at demonstrating the conditions under which the use of replicas can provide significant performance advantages.

The experimental results are summarised in Figures 5.1a–5.1h where we compare the average count of non-local memory accesses with and without replication, for each configuration. Without replication, both read and write access can be non-local, depending

on the address being accessed. With replication, read operations are always local, because tuples are always replicated where they can be potentially accessed. However, this comes at the cost of extra non-local write access to replicate the data. If the system tends to read from the shared memory more often than writing to it, our approach can be beneficial. In Figures 5.1a–5.1h, the number of non-local accesses with replication is maximised when the read and write operations occur with the same probability. Replication seems to be more beneficial with larger memory size (from 32 to 64, or from 128 to 256, or from 256 to 512, or from 512 to 1024).

The `x-axes` of Figures 5.1a–5.1h represent the ratio of expected percentage of `Put` operations, the `y-axes` of Figures 5.1a–5.1h represent the average counts of non-local accesses (read/write) with and without replication, for each configuration.

The experimental results where we computed execution times of non-local accesses (read or write operations) of processes with and without replication, for all configurations are presented in Figures 5.2a, 5.2b, 5.2c, 5.2d, 5.2e 5.2f, and 5.2g.

We found that memory latency is reduced when the system tends to read from the shared memory more often than writing to it, and when such a system has a large memory size, our replication approach seems to be beneficial. By latency, we mean the time required to accomplish a tuple space operation. In other words, by considering the small system with small size memory, the execution times for replication tend to be increased instead of being reduced; whereas by using the large system with large size memory, the execution times for replication are instead reduced. This is an interesting feature of our replication approach, and it shows how much we can learn when choosing a system with a memory of a large size with respect to the number of processes.

It is worth remarking that the fine-grained use of locks in the replicated system as opposed to the non-replicated version (where concurrent reads are instrumented through a global lock) may have facilitated the improved latency observed in the replicated system.

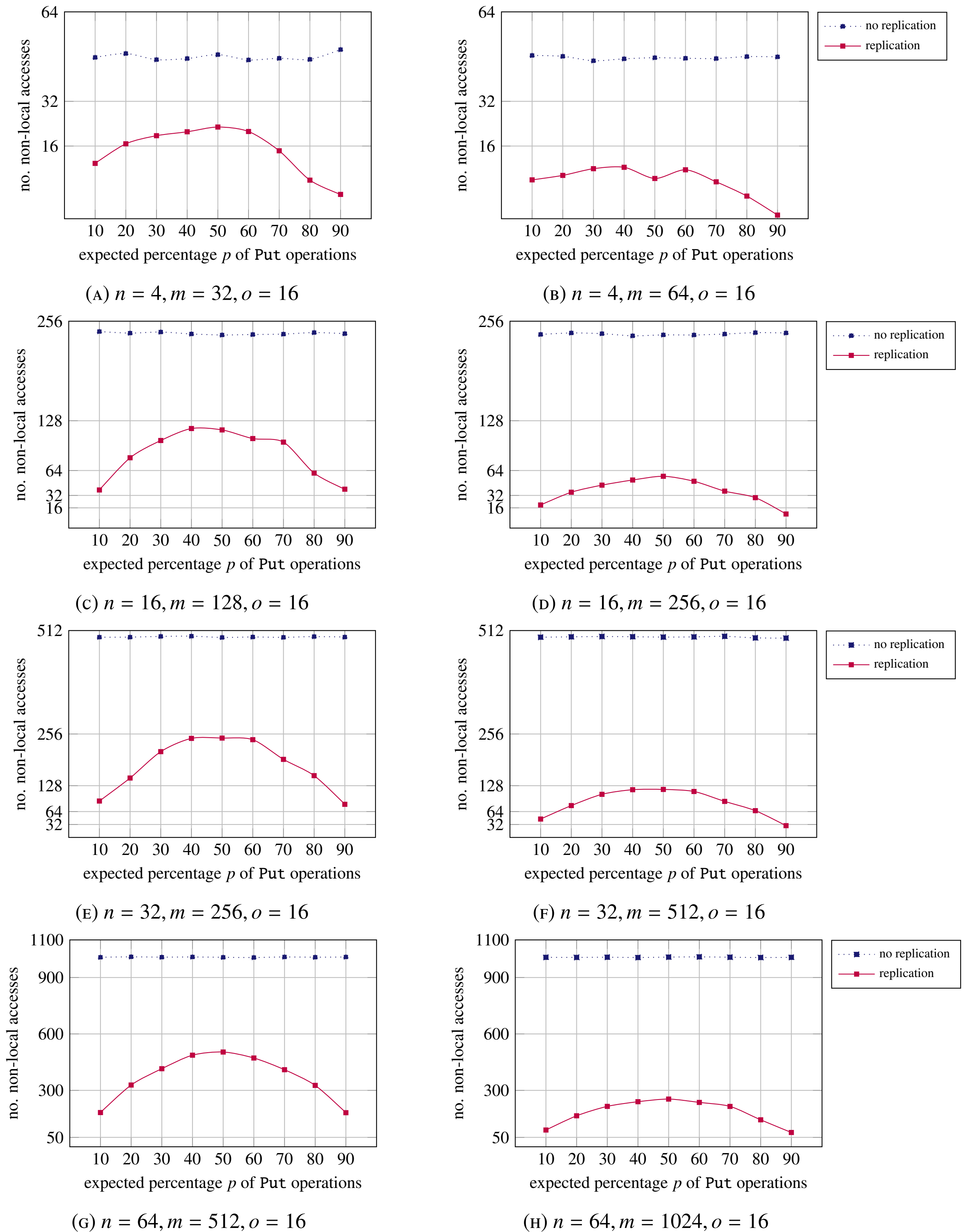
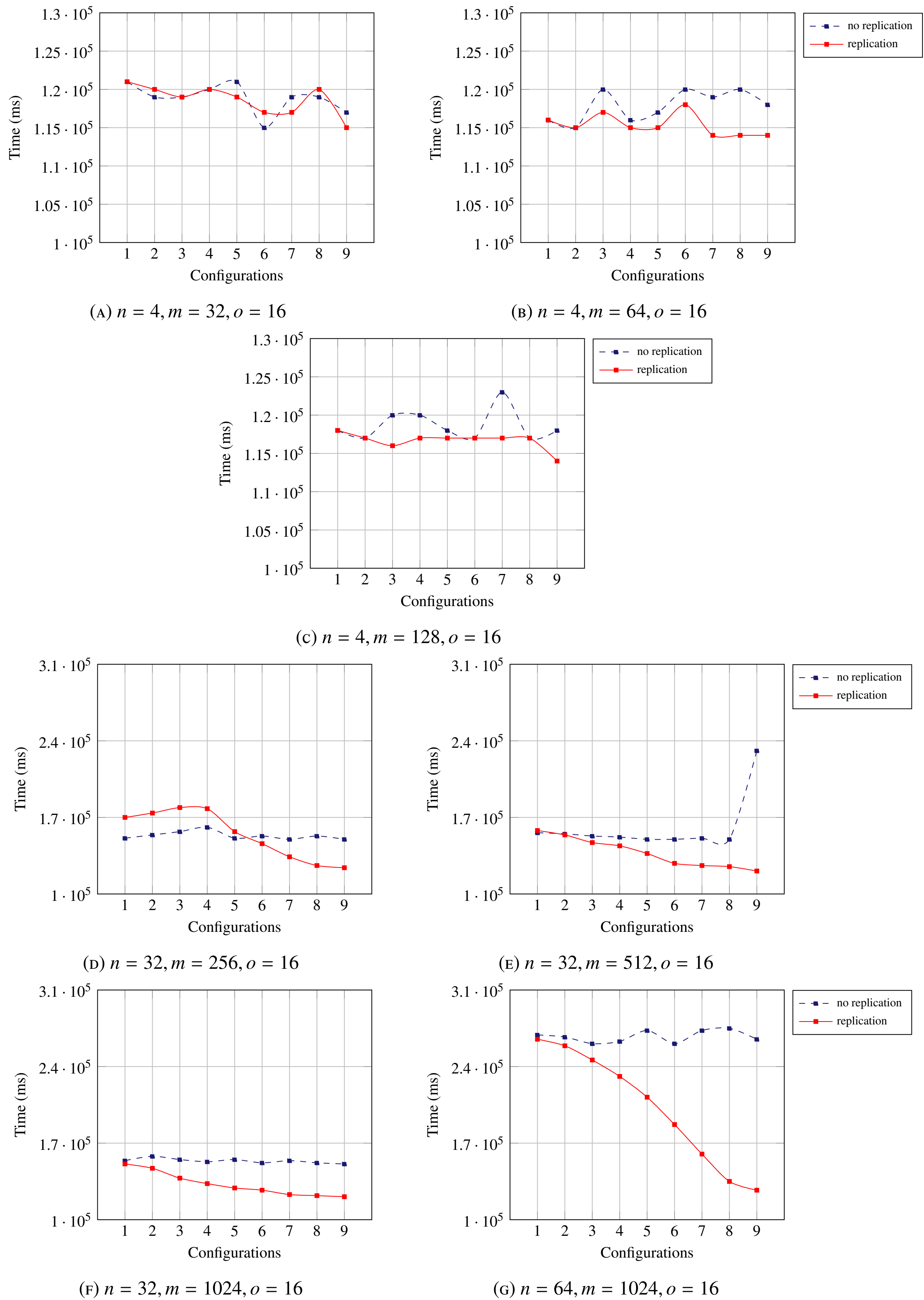


FIGURE 5.1: Non-local read or write operations with and without replication



(A) $n = 4, m = 32, o = 16$

(B) $n = 4, m = 64, o = 16$

(C) $n = 4, m = 128, o = 16$

(D) $n = 32, m = 256, o = 16$

(E) $n = 32, m = 512, o = 16$

(F) $n = 32, m = 1024, o = 16$

(G) $n = 64, m = 1024, o = 16$

FIGURE 5.2: Execution times (ms) of non-local read or write operations with and without replication

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We have provided the background knowledge related to the research pursued in this thesis, including a brief review of research work related to ours. We have also presented some concepts of Go, the host programming language of our prototypes framework implementations.

We have then introduced RepligoSpaces, a replica-aware extension of goSpace, an implementation of Klaim (pSpaces) in Go programming language. The extended coordination primitives of the new language embed a tracking mechanism for the target spaces of all store actions to consistently remove the replicated data at need. Moreover, we have discussed how RepligoSpaces coordination language fits within a fully-mechanisable procedure for automated replication of programs over tuple space that relies on combining static analysis and program transformation.

We have then described a static analyzer and a prototype replicator that we have developed to transform the initial model written in goSpace into the equivalent replicated version written in RepligoSpaces. A lightweight static analysis pass on the initial program computes the sets of target spaces for replication, so that the standard tuple manipulation routines can be replaced by equivalent replica-aware versions. The combined approach preserves strong consistency, thanks to a tracking mechanism embedded in the tuple manipulation routines and to the fact that the set of target spaces is safely over-approximated.

Our workflow is simple and easily extensible, given the modularity between the data-handling layer, the program transformation schema, and the static analysis procedure. Different static analysis techniques may be plugged in with a relatively limited effort.

To evaluate our prototype framework, we conducted experiments on a scenario of a case study considering a system consisting of different computational nodes, each executing a separate program, and interacting through a decentralised data store. For our experiments, we used a standard laptop. The open-source code of our prototype framework and implementation of a case study are publicly released, free for the community to use.

6.2 Future Work

We consider our contribution to be a first step towards developing an integrated framework to experiment with data replication in distributed systems with tuple spaces. We aim to provide different analyses and consistency models to choose from; in order to appreciate the effect of different consistency levels on many interesting classes of more or less complex distributed systems where data replication is used. This would allow us, for instance, to evaluate many interesting classes of systems such as models of hardware cache or complex interaction models, where replication is heavily used and performance is particularly sensitive to variations in the data distribution.

In the near future, we plan to consider further scenarios where the replication technique has successfully been applied to other contexts, such as database systems [140, 141], cloud computing [142–146], and mobile ad-hoc network databases [147], as well as other consistency models for replicated data beyond strong consistency [60, 148, 149].

Further possible directions of future work also include alternative implementations of our replica-aware language extension to, for instance, preserve other forms of consistency levels. We also plan further research on the static analysis procedure to improve the accuracy of the over-approximation in the presence of formal fields, i.e., placeholders in the pattern or variables in the tuple to be stored. We plan to initially focus on simple and efficient techniques to complement the existing analysis with limited effort. To name a few, constant propagation [150] can reduce the overall number of formal fields, or at least restrict the possible values of a given formal field collecting them over different branching paths, for instance; abstract interpretation [151] can overapproximate the interval ranges of the integer variables used as formal fields.

Appendix A
Example of program transformation

We provide an example of a program where the tuple spaces are actually used. We aim at providing some insights on program transformation from a `Get` operation (goSpace) into an `MGetP` operation (RepligoSpaces).

```
1 space.GetP(Template)
```

LISTING 6.1: Before transformation

```
1 MGetP(Template,uri [space])
```

LISTING 6.2: After transformation

As shown in Figure 6.1, first, we created a function that tracks where all get operations (Line 3) block identifications (block id) occur in the given program. Then, we visited the blocks of all statements (lines 6–59), such as loop statements, for statements, switch statements, select statements, if statements, range statements, labelled statements. When visiting the blocks of statements, the reflect method is used to examine types and values of each of the visited block of statements.

For example, lines 61–62 inspect the program via *reflect method* and examine expression statements(i.e., `ExprStmt`) in the program having types of selector expressions (i.e., `SelectorExpr`). Line 64 checks the call function where the name is `"GetP"`. We then transformed the calls from goSpace routines (lines 66–90), i.e., transform the `Get` operation into the replica-aware `MGetP` operation for removing the tuples that match the given templates (lines 92–101).

```
1 // Scan the program to populate blockget[] and getid[] maps,
2 // to keep track of where get operations occur in the program.
3 func scanblockget(fset *token.FileSet, node *ast.BlockStmt, blockid string) {
4   count := 0 // block identifier
5
6   // Visit blocks of loop statements
7   for _, n := range node.List {
8     if reflect.TypeOf(n).String() == "*ast.BlockStmt" {
9       scanblockget(fset, n.(*ast.BlockStmt), blockid+":"+strconv.Itoa(count))
10      count += 1
11    }
12    //Visit blocks of for statements
13    if reflect.TypeOf(n).String() == "*ast.ForStmt" {
14      scanblockget(fset, n.(*ast.ForStmt).Body, blockid+":"+strconv.Itoa(count))
15      count += 1
16    }
17    // Visit blocks of switch statements
18    if reflect.TypeOf(n).String() == "*ast.SwitchStmt" {
19      scanblockget(fset, n.(*ast.SwitchStmt).Body, blockid+":"+strconv.Itoa(count))
20      count += 1
21    }
22    // Visit blocks of select statements
23    if reflect.TypeOf(n).String() == "*ast.SelectStmt" {
24      scanblockget(fset, n.(*ast.SelectStmt).Body, blockid+":"+strconv.Itoa(count))
25      count += 1
26    }
27    // Visit blocks of range statements
28    if reflect.TypeOf(n).String() == "*ast.RangeStmt" {
29      scanblockget(fset, n.(*ast.RangeStmt).Body, blockid+":"+strconv.Itoa(count))
30      count += 1
31    }
32    // Visit blocks of if statements
33    if reflect.TypeOf(n).String() == "*ast.IfStmt" {
34      scanblockget(fset, n.(*ast.IfStmt).Body, blockid+":"+strconv.Itoa(count))
35      count += 1
36      t, ok := (n.(*ast.IfStmt).Else).(*ast.BlockStmt)
37      if ok {
38        scanblockget(fset, t, blockid+":"+strconv.Itoa(count))
39        count += 1
40      }
41    }
42    // Visit blocks of switch statements
43    if reflect.TypeOf(n).String() == "*ast.TypeSwitchStmt" {
44      scanblockget(fset, n.(*ast.TypeSwitchStmt).Body, blockid+":"+strconv.Itoa(
      count))
45      count += 1
46      r, ok := (n.(*ast.TypeSwitchStmt).Assign).(*ast.BlockStmt)
47      if ok {
48        scanblockget(fset, r, blockid+":"+strconv.Itoa(count))
49        count += 1
50      }
51    }
52    // Visit blocks of labeled statements
53    if reflect.TypeOf(n).String() == "*ast.LabeledStmt" {
```

```
54        l, ok := (n.(*ast.LabeledStmt).Stmt).(*ast.BlockStmt)
55        if ok {
56          scanblockget(fset, l, blockid+":"+strconv.Itoa(count))
57          count += 1
58        }
59      }
60    // Transform calls from GoSpace routines
61  if reflect.TypeOf(n).String() == "*ast.ExprStmt" {
62    fn1, ok := n.(*ast.ExprStmt)
63    if ok && reflect.TypeOf(fn1.X.(*ast.CallExpr).Fun).String() == "*ast.SelectorExpr
      " {
64      if fn1.X.(*ast.CallExpr).Fun.(*ast.SelectorExpr).Sel.Name == "GetP" {
65        info("transformer: transforming call GetP block (%s)", blockid)
66        newArgs := make([]ast.Expr, len(fn1.X.(*ast.CallExpr).Args))
67        copy(newArgs, fn1.X.(*ast.CallExpr).Args)
68        var spaceidd string
69        if strings.IndexByte(blockid, ':') != -1 {
70          spaceidd = blockid[:strings.IndexByte(blockid, ':')]
71        } else {
72          spaceidd = blockid
73        }
74        spaceidd = "s" + spaceidd
75        expr2 := ast.NewIdent(spaceidd)
76        identcreatetuple2 := ast.NewIdent("CreateTuple")
77        identrsp2 := ast.NewIdent("rsp")
78        callexpr02 := &ast.CallExpr{Fun: identcreatetuple2, Args: newArgs}
79        //newArgs = append(newArgs, indexpr02)
80        newArgs = append(newArgs, expr2)
81
82        // Generate: GetP(Template, uri[spaceid])
83          var expr ast.Expr
84        expr = &ast.CallExpr{Fun:fn1.X.(*ast.CallExpr).Fun.(*ast.SelectorExpr).Sel,
85            Args: []ast.Expr{callexpr02, identrsp2, expr2}}
86            if expr != nil {
87              fn1.X = expr
88            }
89          }
90        }
91    // transform get operations into replica-aware operations
92    func transformgetoperations(fset *token.FileSet, node *ast.File) {
93    count := 0
94    for _, node := range node.Decls {
95      fn, ok := node.(*ast.FuncDecl)
96      if !ok {
97        continue
98      }
99      scanbloc2(fset, fn.Body, strconv.Itoa(count), fn.Body)
100     count++
101   }
```

FIGURE 6.1: Program transformation for the code shown in Listing 6.2

Additional illustrative examples

We provide an additional illustrative example aimed at showing the impact of the proposed approach and implementation aspects in practical scenarios. In particular, we show an input program that consists of structured processes performing several write and read operations. The source code for the input program is shown in Listing 6.3. In this example, we describe how different operations for an input program are transformed into their replicated operations versions. For example, the coordination primitives shown in lines 72, 80 and 88 (see Listing 6.3) are transformed into their replicated operations shown in lines 89, 97, and 105 (see Listing 6.4), respectively. In the continuation of program transformation, the spaces `(s1,s2,s3,s4)` (lines 45–48 of Listing 6.3) are also transformed into their replicated versions (lines 47–58 of Listing 6.4).

```
1  package main
2
3  import (
4    "os"
5    "fmt"
6    "sync"
7    "time"
8    "math/rand"
9    . "github.com/pspaces/gospace"
10 )
11
12 var debug bool
13 var l sync.Mutex
14
15 var writes_local int
16 var writes_remote int
17 var reads_local int
18 var reads_remote int
19 var writes_replicated int
20 var writestotal int
21 var reads_success int
22 var reads_insuccess int
23 var writes_extra int
24 var wg sync.WaitGroup
25
26 var s1 Space
27 var s2 Space
28 var s3 Space
29 var s4 Space
```

```
30
31
32 func main() {
33   debug = false
34   writes_local = 0
35   writes_remote = 0
36   reads_local = 0
37   reads_remote = 0
38   writes_replicated = 0
39   reads_success = 0
40   reads_insuccess = 0
41   rand.Seed(time.Now().UTC().UnixNano())
42
43   wg.Add(4)
44
45   s1 = NewSpace("tcp://localhost:34001/s1")
46   s2 = NewSpace("tcp://localhost:34002/s2")
47   s3 = NewSpace("tcp://localhost:34003/s3")
48   s4 = NewSpace("tcp://localhost:34004/s4")
49
50
51   go p1()
52   go p2()
53   go p3()
54   go p4()
55
56
57   wg.Wait()
58
59   writes_replicated = 0   // these include the normal put too
60
61   fmt.Println("       loc w,    rem w,   repl w,    loc r,    rem r,    tot w,   succ r,   fail
        r")
62   fmt.Printf ("    %8d, %8d, %8d, %8d, %8d, %8d, %8d, %8d\n", writes_local, writes_remote,
       writes_replicated, reads_local, reads_remote, writestotal, reads_success, reads_insuccess)
63 }
64
65 func p1() {
66   defer wg.Done()
67   var value int
68
69   delay()
70   l.Lock()
71   value = -1
72   s1.QueryP(8,&value)
73   log("p1 r:%d", value)
74   countreads(value,1,1)
75   l.Unlock()
76
77   delay()
78   l.Lock()
79   value = -1
80   s1.QueryP(8,&value)
81   log("p1 r:%d", value)
82   countreads(value,1,1)
83   l.Unlock()
84
85   delay()
86   l.Lock()
87   value = 2
88   s2.Put(15,value)
89   log("p1 w:%d", value)
90   countwrites(value,1,2)
91   l.Unlock()
92
```

```
93   delay()
94   l.Lock()
95   value = -1
96   s1.QueryP(3,&value)
97   log("p1 r:%d", value)
98   countreads(value,1,1)
99   l.Unlock()
100
101  delay()
102  l.Lock()
103  value = -1
104  s3.QueryP(21,&value)
105  log("p1 r:%d", value)
106  countreads(value,1,3)
107  l.Unlock()
108
109  delay()
110  l.Lock()
111  value = -1
112  s2.QueryP(15,&value)
113  log("p1 r:%d", value)
114  countreads(value,1,2)
115  l.Unlock()
116
117  delay()
118  l.Lock()
119  value = -1
120  s1.QueryP(4,&value)
121  log("p1 r:%d", value)
122  countreads(value,1,1)
123  l.Unlock()
124
125  delay()
126  l.Lock()
127  value = -1
128  s1.QueryP(6,&value)
129  log("p1 r:%d", value)
130  countreads(value,1,1)
131  l.Unlock()
132
133  delay()
134  l.Lock()
135  value = -1
136  s4.QueryP(26,&value)
137  log("p1 r:%d", value)
138  countreads(value,1,4)
139  l.Unlock()
140
141  delay()
142  l.Lock()
143  value = -1
144  s1.QueryP(1,&value)
145  log("p1 r:%d", value)
146  countreads(value,1,1)
147  l.Unlock()
148
149  delay()
150  l.Lock()
151  value = -1
152  s1.QueryP(2,&value)
153  log("p1 r:%d", value)
154  countreads(value,1,1)
155  l.Unlock()
156
157  delay()
```

```
158   l.Lock()
159   value = 4
160   s4.Put(30,value)
161   log("p1 w:%d", value)
162   countwrites(value,1,4)
163   l.Unlock()
164
165   delay()
166   l.Lock()
167   value = 4
168   s4.Put(26,value)
169   log("p1 w:%d", value)
170   countwrites(value,1,4)
171   l.Unlock()
172
173   delay()
174   l.Lock()
175   value = -1
176   s2.QueryP(9,&value)
177   log("p1 r:%d", value)
178   countreads(value,1,2)
179   l.Unlock()
180
181   delay()
182   l.Lock()
183   value = -1
184   s1.QueryP(3,&value)
185   log("p1 r:%d", value)
186   countreads(value,1,1)
187   l.Unlock()
188
189   delay()
190   l.Lock()
191   value = -1
192   s1.QueryP(1,&value)
193   log("p1 r:%d", value)
194   countreads(value,1,1)
195   l.Unlock()
196 }
197
198 func p2() {
199   defer wg.Done()
200   var value int
201
202   delay()
203   l.Lock()
204   value = -1
205   s4.QueryP(32,&value)
206   log("p2 r:%d", value)
207   countreads(value,2,4)
208   l.Unlock()
209
210   delay()
211   l.Lock()
212   value = -1
213   s3.QueryP(22,&value)
214   log("p2 r:%d", value)
215   countreads(value,2,3)
216   l.Unlock()
217
218   delay()
219   l.Lock()
220   value = -1
221   s4.QueryP(25,&value)
222   log("p2 r:%d", value)
```

```
223     countreads(value,2,4)
224     l.Unlock()
225
226     delay()
227     l.Lock()
228     value = -1
229     s1.QueryP(4,&value)
230     log("p2 r:%d", value)
231     countreads(value,2,1)
232     l.Unlock()
233
234     delay()
235     l.Lock()
236     value = -1
237     s2.QueryP(10,&value)
238     log("p2 r:%d", value)
239     countreads(value,2,2)
240     l.Unlock()
241
242     delay()
243     l.Lock()
244     value = -1
245     s2.QueryP(12,&value)
246     log("p2 r:%d", value)
247     countreads(value,2,2)
248     l.Unlock()
249
250     delay()
251     l.Lock()
252     value = -1
253     s2.QueryP(12,&value)
254     log("p2 r:%d", value)
255     countreads(value,2,2)
256     l.Unlock()
257
258     delay()
259     l.Lock()
260     value = 4
261     s4.Put(28,value)
262     log("p2 w:%d", value)
263     countwrites(value,2,4)
264     l.Unlock()
265
266     delay()
267     l.Lock()
268     value = -1
269     s2.QueryP(15,&value)
270     log("p2 r:%d", value)
271     countreads(value,2,2)
272     l.Unlock()
273
274     delay()
275     l.Lock()
276     value = -1
277     s2.QueryP(10,&value)
278     log("p2 r:%d", value)
279     countreads(value,2,2)
280     l.Unlock()
281
282     delay()
283     l.Lock()
284     value = -1
285     s1.QueryP(7,&value)
286     log("p2 r:%d", value)
287     countreads(value,2,1)
```

```
288     l.Unlock()
289
290     delay()
291     l.Lock()
292     value = -1
293     s3.QueryP(21,&value)
294     log("p2 r:%d", value)
295     countreads(value,2,3)
296     l.Unlock()
297
298     delay()
299     l.Lock()
300     value = -1
301     s1.QueryP(7,&value)
302     log("p2 r:%d", value)
303     countreads(value,2,1)
304     l.Unlock()
305
306     delay()
307     l.Lock()
308     value = -1
309     s2.QueryP(13,&value)
310     log("p2 r:%d", value)
311     countreads(value,2,2)
312     l.Unlock()
313
314     delay()
315     l.Lock()
316     value = -1
317     s1.QueryP(8,&value)
318     log("p2 r:%d", value)
319     countreads(value,2,1)
320     l.Unlock()
321
322     delay()
323     l.Lock()
324     value = -1
325     s4.QueryP(25,&value)
326     log("p2 r:%d", value)
327     countreads(value,2,4)
328     l.Unlock()
329   }
330
331   func p3() {
332     defer wg.Done()
333     var value int
334
335     delay()
336     l.Lock()
337     value = -1
338     s1.QueryP(3,&value)
339     log("p3 r:%d", value)
340     countreads(value,3,1)
341     l.Unlock()
342
343     delay()
344     l.Lock()
345     value = -1
346     s3.QueryP(17,&value)
347     log("p3 r:%d", value)
348     countreads(value,3,3)
349     l.Unlock()
350
351     delay()
352     l.Lock()
```

```
353    value = -1
354    s2.QueryP(10,&value)
355    log("p3 r:%d", value)
356    countreads(value,3,2)
357    l.Unlock()
358
359    delay()
360    l.Lock()
361    value = -1
362    s1.QueryP(5,&value)
363    log("p3 r:%d", value)
364    countreads(value,3,1)
365    l.Unlock()
366
367    delay()
368    l.Lock()
369    value = -1
370    s3.QueryP(21,&value)
371    log("p3 r:%d", value)
372    countreads(value,3,3)
373    l.Unlock()
374
375    delay()
376    l.Lock()
377    value = -1
378    s2.QueryP(13,&value)
379    log("p3 r:%d", value)
380    countreads(value,3,2)
381    l.Unlock()
382
383    delay()
384    l.Lock()
385    value = -1
386    s3.QueryP(20,&value)
387    log("p3 r:%d", value)
388    countreads(value,3,3)
389    l.Unlock()
390
391    delay()
392    l.Lock()
393    value = 1
394    s1.Put(2,value)
395    log("p3 w:%d", value)
396    countwrites(value,3,1)
397    l.Unlock()
398
399    delay()
400    l.Lock()
401    value = -1
402    s3.QueryP(23,&value)
403    log("p3 r:%d", value)
404    countreads(value,3,3)
405    l.Unlock()
406
407    delay()
408    l.Lock()
409    value = -1
410    s4.QueryP(31,&value)
411    log("p3 r:%d", value)
412    countreads(value,3,4)
413    l.Unlock()
414
415    delay()
416    l.Lock()
417    value = -1
```

```
418     s2.QueryP(12,&value)
419     log("p3 r:%d", value)
420     countreads(value,3,2)
421     l.Unlock()
422
423     delay()
424     l.Lock()
425     value = -1
426     s4.QueryP(28,&value)
427     log("p3 r:%d", value)
428     countreads(value,3,4)
429     l.Unlock()
430
431     delay()
432     l.Lock()
433     value = -1
434     s1.QueryP(1,&value)
435     log("p3 r:%d", value)
436     countreads(value,3,1)
437     l.Unlock()
438
439     delay()
440     l.Lock()
441     value = -1
442     s2.QueryP(12,&value)
443     log("p3 r:%d", value)
444     countreads(value,3,2)
445     l.Unlock()
446
447     delay()
448     l.Lock()
449     value = -1
450     s4.QueryP(30,&value)
451     log("p3 r:%d", value)
452     countreads(value,3,4)
453     l.Unlock()
454
455     delay()
456     l.Lock()
457     value = 4
458     s4.Put(27,value)
459     log("p3 w:%d", value)
460     countwrites(value,3,4)
461     l.Unlock()
462 }
463
464 func p4() {
465     defer wg.Done()
466     var value int
467
468     delay()
469     l.Lock()
470     value = -1
471     s4.QueryP(28,&value)
472     log("p4 r:%d", value)
473     countreads(value,4,4)
474     l.Unlock()
475
476     delay()
477     l.Lock()
478     value = -1
479     s4.QueryP(26,&value)
480     log("p4 r:%d", value)
481     countreads(value,4,4)
482     l.Unlock()
```

```
483
484     delay()
485     l.Lock()
486     value = -1
487     s1.QueryP(5,&value)
488     log("p4 r:%d", value)
489     countreads(value,4,1)
490     l.Unlock()
491
492     delay()
493     l.Lock()
494     value = -1
495     s2.QueryP(12,&value)
496     log("p4 r:%d", value)
497     countreads(value,4,2)
498     l.Unlock()
499
500     delay()
501     l.Lock()
502     value = -1
503     s2.QueryP(15,&value)
504     log("p4 r:%d", value)
505     countreads(value,4,2)
506     l.Unlock()
507
508     delay()
509     l.Lock()
510     value = 3
511     s3.Put(24,value)
512     log("p4 w:%d", value)
513     countwrites(value,4,3)
514     l.Unlock()
515
516     delay()
517     l.Lock()
518     value = -1
519     s2.QueryP(11,&value)
520     log("p4 r:%d", value)
521     countreads(value,4,2)
522     l.Unlock()
523
524     delay()
525     l.Lock()
526     value = -1
527     s4.QueryP(29,&value)
528     log("p4 r:%d", value)
529     countreads(value,4,4)
530     l.Unlock()
531
532     delay()
533     l.Lock()
534     value = -1
535     s3.QueryP(18,&value)
536     log("p4 r:%d", value)
537     countreads(value,4,3)
538     l.Unlock()
539
540     delay()
541     l.Lock()
542     value = -1
543     s4.QueryP(27,&value)
544     log("p4 r:%d", value)
545     countreads(value,4,4)
546     l.Unlock()
547
```

```
548   delay()
549   l.Lock()
550   value = -1
551   s2.QueryP(13,&value)
552   log("p4 r:%d", value)
553   countreads(value,4,2)
554   l.Unlock()
555
556   delay()
557   l.Lock()
558   value = -1
559   s1.QueryP(1,&value)
560   log("p4 r:%d", value)
561   countreads(value,4,1)
562   l.Unlock()
563
564   delay()
565   l.Lock()
566   value = -1
567   s1.QueryP(2,&value)
568   log("p4 r:%d", value)
569   countreads(value,4,1)
570   l.Unlock()
571
572   delay()
573   l.Lock()
574   value = -1
575   s1.QueryP(8,&value)
576   log("p4 r:%d", value)
577   countreads(value,4,1)
578   l.Unlock()
579
580   delay()
581   l.Lock()
582   value = -1
583   s1.QueryP(2,&value)
584   log("p4 r:%d", value)
585   countreads(value,4,1)
586   l.Unlock()
587
588   delay()
589   l.Lock()
590   value = -1
591   s1.QueryP(4,&value)
592   log("p4 r:%d", value)
593   countreads(value,4,1)
594   l.Unlock()
595 }
596
597
598 func log(format string, a ...interface{}) {
599   if debug {
600       fmt.Fprintf(os.Stdout, format+"\n", a ...)
601     }
602 }
603
604 func delay() {
605   time.Sleep(time.Duration(rand.Int63n(75))*time.Millisecond)
606 }
607
608 func countreads(value int, localspace int, targetspace int) {
609   if value == -1 {
610     reads_insuccess+=1
611   } else {
612     reads_success+=1
```

```
613     }
614
615     if localspace == targetspace {
616       reads_local+=1
617     } else {
618       reads_remote+=1
619     }
620   }
621
622   func countwrites(value int, localspace int, targetspace int) {
623     writestotal+=1
624
625     if localspace == targetspace {
626       writes_local+=1
627     } else {
628       writes_remote+=1
629     }
630   }
```

Listing 6.3: Example of input program (Non-replicated)

A replicated program for the input program given in Listing 6.3 is shown in Listing 6.4.

```
1   package main
2   import (
3     "os"
4     "fmt"
5     "sync"
6     "time"
7     "math/rand"
8     . "github.com/pspaces/gospace"
9     . "github.com/repligospaces"
10  )
11
12  var uri = make(map[Space]string)
13  var Sp = make(map[string]*Space)
14  var rsp Replispace = Replispace{Sp: Sp}
15
16  var debug bool
17  var l sync.Mutex
18  var writes_local int
19  var writes_remote int
20  var reads_local int
21  var reads_remote int
22  var writes_replicated int
23  var writestotal int
24  var reads_success int
25  var reads_insuccess int
26  var writes_extra int
27  var wg sync.WaitGroup
28
29  var s1 Space
30  var s2 Space
31  var s3 Space
32  var s4 Space
33
34  func main() {
35    debug = false
36    writes_local = 0
37    writes_remote = 0
38    reads_local = 0
39    reads_remote = 0
40    writes_replicated = Getwcount()-writes_local
```

```
41   reads_success = 0
42   reads_insuccess = 0
43   rand.Seed(time.Now().UTC().UnixNano())
44
45   wg.Add(4)
46
47   s1 = NewSpace("tcp://localhost:34001/s1")
48   Sp["tcp://localhost:34001/s1"] = &s1
49   uri[s1] = "tcp://localhost:34001/s1"
50   s2 = NewSpace("tcp://localhost:34002/s2")
51   Sp["tcp://localhost:34002/s2"] = &s2
52   uri[s2] = "tcp://localhost:34002/s2"
53   s3 = NewSpace("tcp://localhost:34003/s3")
54   Sp["tcp://localhost:34003/s3"] = &s3
55   uri[s3] = "tcp://localhost:34003/s3"
56   s4 = NewSpace("tcp://localhost:34004/s4")
57   Sp["tcp://localhost:34004/s4"] = &s4
58   uri[s4] = "tcp://localhost:34004/s4"
59
60   go p1()
61   go p2()
62   go p3()
63   go p4()
64
65   wg.Wait()
66
67   writes_replicated = Getwcount()-writes_local  // these include the normal put too
68
69   fmt.Println("       loc w,    rem w,   repl w,    loc r,    rem r,    tot w,   succ r,   fail
       r")
70   fmt.Printf("    %8d, %8d, %8d, %8d, %8d, %8d, %8d, %8d\n", writes_local, writes_remote,
      writes_replicated, reads_local, reads_remote, writestotal, reads_success, reads_insuccess)
71 }
72
73 func p1() {
74   targets2 := make([]string, 2)
75   targets2[1] = uri[s4]
76   targets2[0] = uri[s1]
77   targets1 := make([]string, 1)
78   targets1[0] = uri[s3]
79   targets0 := make([]string, 3)
80   targets0[2] = uri[s4]
81   targets0[1] = uri[s2]
82   targets0[0] = uri[s1]
83
84   defer wg.Done()
85   var value int
86   delay()
87   l.Lock()
88   value = -1
89   QueryP(CreateTuple(8, &value), rsp, s1)
90   log("p1 r:%d", value)
91   countreads(value, 1, 1)
92   l.Unlock()
93
94   delay()
95   l.Lock()
96   value = -1
97   QueryP(CreateTuple(8, &value), rsp, s1)
98   log("p1 r:%d", value)
99   countreads(value, 1, 1)
100  l.Unlock()
101
102  delay()
103  l.Lock()
```

```
104    value = 2
105    Put(CreateTuple(15, value), rsp, targets0)
106    log("p1 w:%d", value)
107    countwrites(value, 1, 2)
108    l.Unlock()
109
110    delay()
111    l.Lock()
112    value = -1
113    QueryP(CreateTuple(3, &value), rsp, s1)
114    log("p1 r:%d", value)
115    countreads(value, 1, 1)
116    l.Unlock()
117
118    delay()
119    l.Lock()
120    value = -1
121    QueryP(CreateTuple(21, &value), rsp, s1)
122    log("p1 r:%d", value)
123    countreads(value, 1, 3)
124    l.Unlock()
125
126    delay()
127    l.Lock()
128    value = -1
129    QueryP(CreateTuple(15, &value), rsp, s1)
130    log("p1 r:%d", value)
131    countreads(value, 1, 2)
132    l.Unlock()
133
134    delay()
135    l.Lock()
136    value = -1
137    QueryP(CreateTuple(4, &value), rsp, s1)
138    log("p1 r:%d", value)
139    countreads(value, 1, 1)
140    l.Unlock()
141
142    delay()
143    l.Lock()
144    value = -1
145    QueryP(CreateTuple(6, &value), rsp, s1)
146    log("p1 r:%d", value)
147    countreads(value, 1, 1)
148    l.Unlock()
149
150    delay()
151    l.Lock()
152    value = -1
153    QueryP(CreateTuple(26, &value), rsp, s1)
154    log("p1 r:%d", value)
155    countreads(value, 1, 4)
156    l.Unlock()
157
158    delay()
159    l.Lock()
160    value = -1
161    QueryP(CreateTuple(1, &value), rsp, s1)
162    log("p1 r:%d", value)
163    countreads(value, 1, 1)
164    l.Unlock()
165
166    delay()
167    l.Lock()
168    value = -1
```

```
169     QueryP(CreateTuple(2, &value), rsp, s1)
170     log("p1 r:%d", value)
171     countreads(value, 1, 1)
172     l.Unlock()
173 
174     delay()
175     l.Lock()
176     value = 4
177     Put(CreateTuple(30, value), rsp, targets1)
178     log("p1 w:%d", value)
179     countwrites(value, 1, 4)
180     l.Unlock()
181 
182     delay()
183     l.Lock()
184     value = 4
185     Put(CreateTuple(26, value), rsp, targets2)
186     log("p1 w:%d", value)
187     countwrites(value, 1, 4)
188     l.Unlock()
189 
190     delay()
191     l.Lock()
192     value = -1
193     QueryP(CreateTuple(9, &value), rsp, s1)
194     log("p1 r:%d", value)
195     countreads(value, 1, 2)
196     l.Unlock()
197 
198     delay()
199     l.Lock()
200     value = -1
201     QueryP(CreateTuple(3, &value), rsp, s1)
202     log("p1 r:%d", value)
203     countreads(value, 1, 1)
204     l.Unlock()
205 
206     delay()
207     l.Lock()
208     value = -1
209     QueryP(CreateTuple(1, &value), rsp, s1)
210     log("p1 r:%d", value)
211     countreads(value, 1, 1)
212     l.Unlock()
213 }
214 
215 func p2() {
216     targets0 := make([]string, 2)
217     targets0[1] = uri[s4]
218     targets0[0] = uri[s3]
219 
220     defer wg.Done()
221     var value int
222 
223     delay()
224     l.Lock()
225     value = -1
226     QueryP(CreateTuple(32, &value), rsp, s2)
227     log("p2 r:%d", value)
228     countreads(value, 2, 4)
229     l.Unlock()
230 
231     delay()
232     l.Lock()
233     value = -1
```

```
234     QueryP(CreateTuple(22, &value), rsp, s2)
235     log("p2 r:%d", value)
236     countreads(value, 2, 3)
237     l.Unlock()
238
239     delay()
240     l.Lock()
241     value = -1
242     QueryP(CreateTuple(25, &value), rsp, s2)
243     log("p2 r:%d", value)
244     countreads(value, 2, 4)
245     l.Unlock()
246
247     delay()
248     l.Lock()
249     value = -1
250     QueryP(CreateTuple(4, &value), rsp, s2)
251     log("p2 r:%d", value)
252     countreads(value, 2, 1)
253     l.Unlock()
254
255     delay()
256     l.Lock()
257     value = -1
258     QueryP(CreateTuple(10, &value), rsp, s2)
259     log("p2 r:%d", value)
260     countreads(value, 2, 2)
261     l.Unlock()
262
263     delay()
264     l.Lock()
265     value = -1
266     QueryP(CreateTuple(12, &value), rsp, s2)
267     log("p2 r:%d", value)
268     countreads(value, 2, 2)
269     l.Unlock()
270
271     delay()
272     l.Lock()
273     value = -1
274     QueryP(CreateTuple(12, &value), rsp, s2)
275     log("p2 r:%d", value)
276     countreads(value, 2, 2)
277     l.Unlock()
278
279     delay()
280     l.Lock()
281     value = 4
282     Put(CreateTuple(28, value), rsp, targets0)
283     log("p2 w:%d", value)
284     countwrites(value, 2, 4)
285     l.Unlock()
286
287     delay()
288     l.Lock()
289     value = -1
290     QueryP(CreateTuple(15, &value), rsp, s2)
291     log("p2 r:%d", value)
292     countreads(value, 2, 2)
293     l.Unlock()
294
295     delay()
296     l.Lock()
297     value = -1
298     QueryP(CreateTuple(10, &value), rsp, s2)
```

```
299     log("p2 r:%d", value)
300     countreads(value, 2, 2)
301     l.Unlock()
302 
303     delay()
304     l.Lock()
305     value = -1
306     QueryP(CreateTuple(7, &value), rsp, s2)
307     log("p2 r:%d", value)
308     countreads(value, 2, 1)
309     l.Unlock()
310 
311     delay()
312     l.Lock()
313     value = -1
314     QueryP(CreateTuple(21, &value), rsp, s2)
315     log("p2 r:%d", value)
316     countreads(value, 2, 3)
317     l.Unlock()
318 
319     delay()
320     l.Lock()
321     value = -1
322     QueryP(CreateTuple(7, &value), rsp, s2)
323     log("p2 r:%d", value)
324     countreads(value, 2, 1)
325     l.Unlock()
326 
327     delay()
328     l.Lock()
329     value = -1
330     QueryP(CreateTuple(13, &value), rsp, s2)
331     log("p2 r:%d", value)
332     countreads(value, 2, 2)
333     l.Unlock()
334 
335     delay()
336     l.Lock()
337     value = -1
338     QueryP(CreateTuple(8, &value), rsp, s2)
339     log("p2 r:%d", value)
340     countreads(value, 2, 1)
341     l.Unlock()
342 
343     delay()
344     l.Lock()
345     value = -1
346     QueryP(CreateTuple(25, &value), rsp, s2)
347     log("p2 r:%d", value)
348     countreads(value, 2, 4)
349     l.Unlock()
350   }
351 
352   func p3() {
353     targets1 := make([]string, 1)
354     targets1[0] = uri[s4]
355     targets0 := make([]string, 3)
356     targets0[2] = uri[s4]
357     targets0[1] = uri[s4]
358     targets0[0] = uri[s1]
359 
360     defer wg.Done()
361     var value int
362 
363     delay()
```

```
364     l.Lock()
365     value = -1
366     QueryP(CreateTuple(3, &value), rsp, s3)
367     log("p3 r:%d", value)
368     countreads(value, 3, 1)
369     l.Unlock()
370
371     delay()
372     l.Lock()
373     value = -1
374     QueryP(CreateTuple(17, &value), rsp, s3)
375     log("p3 r:%d", value)
376     countreads(value, 3, 3)
377     l.Unlock()
378
379     delay()
380     l.Lock()
381     value = -1
382     QueryP(CreateTuple(10, &value), rsp, s3)
383     log("p3 r:%d", value)
384     countreads(value, 3, 2)
385     l.Unlock()
386
387     delay()
388     l.Lock()
389     value = -1
390     QueryP(CreateTuple(5, &value), rsp, s3)
391     log("p3 r:%d", value)
392     countreads(value, 3, 1)
393     l.Unlock()
394
395     delay()
396     l.Lock()
397     value = -1
398     QueryP(CreateTuple(21, &value), rsp, s3)
399     log("p3 r:%d", value)
400     countreads(value, 3, 3)
401     l.Unlock()
402
403     delay()
404     l.Lock()
405     value = -1
406     QueryP(CreateTuple(13, &value), rsp, s3)
407     log("p3 r:%d", value)
408     countreads(value, 3, 2)
409     l.Unlock()
410
411     delay()
412     l.Lock()
413     value = -1
414     QueryP(CreateTuple(20, &value), rsp, s3)
415     log("p3 r:%d", value)
416     countreads(value, 3, 3)
417     l.Unlock()
418
419     delay()
420     l.Lock()
421     value = 1
422     Put(CreateTuple(2, value), rsp, targets0)
423     log("p3 w:%d", value)
424     countwrites(value, 3, 1)
425     l.Unlock()
426
427     delay()
428     l.Lock()
```

```
429   value = -1
430   QueryP(CreateTuple(23, &value), rsp, s3)
431   log("p3 r:%d", value)
432   countreads(value, 3, 3)
433   l.Unlock()
434
435   delay()
436   l.Lock()
437   value = -1
438   QueryP(CreateTuple(31, &value), rsp, s3)
439   log("p3 r:%d", value)
440   countreads(value, 3, 4)
441   l.Unlock()
442
443   delay()
444   l.Lock()
445   value = -1
446   QueryP(CreateTuple(12, &value), rsp, s3)
447   log("p3 r:%d", value)
448   countreads(value, 3, 2)
449   l.Unlock()
450
451   delay()
452   l.Lock()
453   value = -1
454   QueryP(CreateTuple(28, &value), rsp, s3)
455   log("p3 r:%d", value)
456   countreads(value, 3, 4)
457   l.Unlock()
458
459   delay()
460   l.Lock()
461   value = -1
462   QueryP(CreateTuple(1, &value), rsp, s3)
463   log("p3 r:%d", value)
464   countreads(value, 3, 1)
465   l.Unlock()
466
467   delay()
468   l.Lock()
469   value = -1
470   QueryP(CreateTuple(12, &value), rsp, s3)
471   log("p3 r:%d", value)
472   countreads(value, 3, 2)
473   l.Unlock()
474
475   delay()
476   l.Lock()
477   value = -1
478   QueryP(CreateTuple(30, &value), rsp, s3)
479   log("p3 r:%d", value)
480   countreads(value, 3, 4)
481   l.Unlock()
482
483   delay()
484   l.Lock()
485   value = 4
486   Put(CreateTuple(27, value), rsp, targets1)
487   log("p3 w:%d", value)
488   countwrites(value, 3, 4)
489   l.Unlock()
490 }
491
492 func p4() {
493   targets0 := make([]string, 0)
```

```
494     defer wg.Done()
495     var value int
496 
497     delay()
498     l.Lock()
499     value = -1
500     QueryP(CreateTuple(28, &value), rsp, s4)
501     log("p4 r:%d", value)
502     countreads(value, 4, 4)
503     l.Unlock()
504 
505     delay()
506     l.Lock()
507     value = -1
508     QueryP(CreateTuple(26, &value), rsp, s4)
509     log("p4 r:%d", value)
510     countreads(value, 4, 4)
511     l.Unlock()
512 
513     delay()
514     l.Lock()
515     value = -1
516     QueryP(CreateTuple(5, &value), rsp, s4)
517     log("p4 r:%d", value)
518     countreads(value, 4, 1)
519     l.Unlock()
520 
521     delay()
522     l.Lock()
523     value = -1
524     QueryP(CreateTuple(12, &value), rsp, s4)
525     log("p4 r:%d", value)
526     countreads(value, 4, 2)
527     l.Unlock()
528 
529     delay()
530     l.Lock()
531     value = -1
532     QueryP(CreateTuple(15, &value), rsp, s4)
533     log("p4 r:%d", value)
534     countreads(value, 4, 2)
535     l.Unlock()
536 
537     delay()
538     l.Lock()
539     value = 3
540     Put(CreateTuple(24, value), rsp, targets0)
541     log("p4 w:%d", value)
542     countwrites(value, 4, 3)
543     l.Unlock()
544 
545     delay()
546     l.Lock()
547     value = -1
548     QueryP(CreateTuple(11, &value), rsp, s4)
549     log("p4 r:%d", value)
550     countreads(value, 4, 2)
551     l.Unlock()
552 
553     delay()
554     l.Lock()
555     value = -1
556     QueryP(CreateTuple(29, &value), rsp, s4)
557     log("p4 r:%d", value)
558     countreads(value, 4, 4)
```

```
559     l.Unlock()
560
561     delay()
562     l.Lock()
563     value = -1
564     QueryP(CreateTuple(18, &value), rsp, s4)
565     log("p4 r:%d", value)
566     countreads(value, 4, 3)
567     l.Unlock()
568
569     delay()
570     l.Lock()
571     value = -1
572     QueryP(CreateTuple(27, &value), rsp, s4)
573     log("p4 r:%d", value)
574     countreads(value, 4, 4)
575     l.Unlock()
576
577     delay()
578     l.Lock()
579     value = -1
580     QueryP(CreateTuple(13, &value), rsp, s4)
581     log("p4 r:%d", value)
582     countreads(value, 4, 2)
583     l.Unlock()
584
585     delay()
586     l.Lock()
587     value = -1
588     QueryP(CreateTuple(1, &value), rsp, s4)
589     log("p4 r:%d", value)
590     countreads(value, 4, 1)
591     l.Unlock()
592
593     delay()
594     l.Lock()
595     value = -1
596     QueryP(CreateTuple(2, &value), rsp, s4)
597     log("p4 r:%d", value)
598     countreads(value, 4, 1)
599     l.Unlock()
600
601     delay()
602     l.Lock()
603     value = -1
604     QueryP(CreateTuple(8, &value), rsp, s4)
605     log("p4 r:%d", value)
606     countreads(value, 4, 1)
607     l.Unlock()
608
609     delay()
610     l.Lock()
611     value = -1
612     QueryP(CreateTuple(2, &value), rsp, s4)
613     log("p4 r:%d", value)
614     countreads(value, 4, 1)
615     l.Unlock()
616
617     delay()
618     l.Lock()
619     value = -1
620     QueryP(CreateTuple(4, &value), rsp, s4)
621     log("p4 r:%d", value)
622     countreads(value, 4, 1)
623     l.Unlock()
```

```
624 }
625
626 func log(format string, a ...interface{}) {
627   if debug {
628     fmt.Fprintf(os.Stdout, format+"\n", a...)
629   }
630 }
631
632 func delay() {
633   time.Sleep(time.Duration(rand.Int63n(75)) * time.Millisecond)
634 }
635
636 func countreads(value int, localspace int, targetspace int) {
637   if value == -1 {
638     reads_insuccess += 1
639   } else {
640     reads_success += 1
641   }
642
643   if localspace == targetspace {
644     reads_local += 1
645   } else {
646     reads_remote += 1
647   }
648 }
649
650 func countwrites(value int, localspace int, targetspace int) {
651   writestotal += 1
652
653   if localspace == targetspace {
654     writes_local += 1
655   } else {
656     writes_remote += 1
657   }
658 }
```

LISTING 6.4: Example of replicated program for the code shown in Listing 6.3

Bibliography

[1] Aline Uwimbabazi, Omar Inverso, and Rocco De Nicola. Automated replication of tuple spaces via static analysis. In Hossein Hojjat and Mieke Massink, editors, *Fundamentals of Software Engineering - 9th International Conference, FSEN 2021, Virtual Event, May 19-21, 2021, Revised Selected Papers*, volume 12818 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2021. doi: 10.1007/978-3-030-89247-0_2. URL https://doi.org/10.1007/978-3-030-89247-0_2.

[2] Fernando Pedone, Matthias Wiesmann, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems, Taipei, Taiwan, April 10-13, 2000*, pages 464–474. IEEE Computer Society, 2000. doi: 10.1109/ICDCS.2000.840959. URL https://doi.org/10.1109/ICDCS.2000.840959.

[3] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Trans. Software Eng.*, 24(5): 315–330, 1998. doi: 10.1109/32.685256. URL https://doi.org/10.1109/32.685256.

[4] Rocco De Nicola, Daniele Gorla, and Rosario Pugliese. On the expressive power of Klaim-based calculi. *Theor. Comput. Sci.*, 356(3):387–421, 2006. doi: 10.1016/j.tcs.2006.02.007. URL https://doi.org/10.1016/j.tcs.2006.02.007.

[5] Marina Andrić. *Programming Abstractions for Data Sharing in Distributed Spaces*. PhD thesis, IMT School of advanced Studies, Italy, 2017.

[6] Marina Andrić, Rocco De Nicola, and Alberto Lluch-Lafuente. Replica-based high-performance tuple space computing. In *Coordination Models and Languages - 17th IFIP WG 6.1 International Conference, COORDINATION 2015*, pages 3–18, 2015.

[7] The pSpaces Authors. *Programming with Spaces*. URL `https://github.com/pSpaces/`.

[8] David Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985. doi: 10.1145/2363.2433. URL `https://doi.org/10.1145/2363.2433`.

[9] Nicholas Carriero and David Gelernter. Linda in Context. *Commun. ACM*, 32 (4):444–458, 1989. doi: 10.1145/63334.63337. URL `https://doi.org/10.1145/63334.63337`.

[10] David Gelernter. Multiple Tuple Spaces in Linda. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *PARLE '89: Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 12-16, 1989, Proceedings*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27. Springer, 1989. doi: 10.1007/3-540-51285-3_30. URL `https://doi.org/10.1007/3-540-51285-3_30`.

[11] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–107, 1992. doi: 10.1145/129630.376083. URL `https://doi.org/10.1145/129630.376083`.

[12] Peter Wyckoff, Stephen W. McLaughry, Tobin J. Lehman, and Daniel Alexander Ford. T spaces. *IBM Syst. J.*, 37(3):454–474, 1998. doi: 10.1147/sj.373.0454. URL `https://doi.org/10.1147/sj.373.0454`.

[13] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., GBR, 1st edition, 1999. ISBN 0201309556.

[14] Paolo Ciancarini, Robert Tolksdorf, Fabio Vitali, Davide Rossi, and Andreas Knoche. Coordinating Multiagent Applications on the WWW: A reference architecture. *IEEE Trans. Software Eng.*, 24(5):362–375, 1998. doi: 10.1109/32.685259. URL `https://doi.org/10.1109/32.685259`.

[15] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda Meets Mobility. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, pages 368–377. ACM, 1999. doi: 10.1145/302405.302659. URL https://doi.org/10.1145/302405.302659.

[16] Amy L. Murphy and Gian Pietro Picco. Using Lime to support replication for availability in mobile ad hoc networks. In Paolo Ciancarini and Herbert Wiklicky, editors, *Coordination Models and Languages, 8th International Conference, COORDINATION 2006, Bologna, Italy, June 14-16, 2006, Proceedings*, volume 4038 of *Lecture Notes in Computer Science*, pages 194–211. Springer, 2006. doi: 10.1007/11767954_13. URL https://doi.org/10.1007/11767954_13.

[17] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Auction-based agent negotiation via programmable tuple spaces. In Matthias Klusch and Larry Kerschberg, editors, *Cooperative Information Agents IV, The Future of Information Agents in Cyberspace, 4th International Workshop, CIA 2000, Boston, MA, USA, July 7-9, 2000, Proceedings*, volume 1860 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2000. doi: 10.1007/978-3-540-45012-2_9. URL https://doi.org/10.1007/978-3-540-45012-2_9.

[18] Andrea Omicini and Franco Zambonelli. Coordination for internet application development. *Auton. Agents Multi Agent Syst.*, 2(3):251–269, 1999. doi: 10.1023/A:1010060322135. URL https://doi.org/10.1023/A:1010060322135.

[19] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Trans. Softw. Eng. Methodol.*, 18(4):15:1–15:56, 2009. doi: 10.1145/1538942.1538945. URL https://doi.org/10.1145/1538942.1538945.

[20] Lorenzo Bettini, Emanuela Merelli, and Francesco Tiezzi. X-Klaim is Back. In Michele Boreale, Flavio Corradini, Michele Loreti, and Rosario Pugliese, editors, *Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday*, volume 11665 of *Lecture Notes in Computer Science*, pages 115–135. Springer, 2019. doi: 10.1007/978-3-030-21485-2_8. URL https://doi.org/10.1007/978-3-030-21485-2_8.

[21] Vitaly Buravlev, Rocco De Nicola, and Claudio Antares Mezzina. Tuple spaces implementations and their efficiency. In Alberto Lluch-Lafuente and José Proença, editors, *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9686 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2016. doi: 10.1007/978-3-319-39519-7_4. URL `https://doi.org/10.1007/978-3-319-39519-7_4`.

[22] Mirko Viroli, Matteo Casadei, Sara Montagna, and Franco Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Trans. Auton. Adapt. Syst.*, 6(2):14:1–14:24, 2011. doi: 10.1145/1968513.1968517. URL `https://doi.org/10.1145/1968513.1968517`.

[23] Rob Pike. Go at google. In Gary T. Leavens, editor, *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*, pages 5–6. ACM, 2012. doi: 10.1145/2384716.2384720. URL `https://doi.org/10.1145/2384716.2384720`.

[24] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. ISBN 3-540-10235-3. doi: 10.1007/3-540-10235-3. URL `https://doi.org/10.1007/3-540-10235-3`.

[25] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN 0-13-153271-5.

[26] Linas Kaminskas and Alberto Lluch-Lafuente. Aggregation policies for tuple spaces. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings*, volume 10852 of *Lecture Notes in Computer Science*, pages 181–199. Springer, 2018. doi: 10.1007/978-3-319-92408-3_8. URL `https://doi.org/10.1007/978-3-319-92408-3_8`.

[27] Andrea Omicini, Franco Zambonelli, Matthias Klusch, and Robert Tolksdorf, editors. *Coordination of Internet Agents: Models, Technologies, and Applications*. Springer, 2001. ISBN 3-540-41613-7.

[28] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Comput. Surv.*, 21(3):323–357, 1989. doi: 10.1145/72551.72553. URL https://doi.org/10.1145/72551.72553.

[29] Iain Merrick and Alan Wood. Coordination with scopes. In Barrett R. Bryant, Janice H. Carroll, Ernesto Damiani, Hisham Haddad, and Dave Oppenheim, editors, *Applied Computing 2000, Proceedings of the 2000 ACM Symposium on Applied Computing, Villa Olmo, Via Cantoni 1, 22100 Como, Italy, March 19-21, 2000. Volume 1*, pages 210–217. ACM, 2000. doi: 10.1145/335603.335747. URL https://doi.org/10.1145/335603.335747.

[30] Jerrold Sol Leichter. *Shared Tuple Memories, Shared Memories, Buses and Lan's–Linda Implementations across the Spectrum of Connectivity*. PhD thesis, Yale University, USA, 1989.

[31] Susanne Hupfer, David Kaminsky, Nicholas Carriero, and David Gelernter. Coordination Applications of Linda. In Jean-Pierre Banâtre and Daniel Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages, Mont Saint-Michel, France, June 17-19, 1991, Procedings*, volume 574 of *Lecture Notes in Computer Science*, pages 187–194. Springer, 1991. doi: 10.1007/3-540-55160-3_43. URL https://doi.org/10.1007/3-540-55160-3_43.

[32] Naftaly H. Minsky and Jerrold Leichter. Law-Governed Linda as a Coordination Model. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems, ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Bologna, Italy, July 5, 1994, Selected Papers*, volume 924 of *Lecture Notes in Computer Science*, pages 125–146. Springer, 1994. doi: 10.1007/3-540-59450-7_8. URL https://doi.org/10.1007/3-540-59450-7_8.

[33] David Gelernter and Lenore D. Zuck. On What Linda Is: Formal description of Linda as a reactive system. In David Garlan and Daniel Le Métayer, editors, *Coordination Languages and Models, Second International Conference, COORDINATION '97, Berlin, Germany, September 1-3, 1997, Proceedings*, volume 1282 of *Lecture Notes in Computer Science*, pages 187–204. Springer, 1997. doi: 10.1007/3-540-63383-9_81. URL https://doi.org/10.1007/3-540-63383-9_81.

[34] Nur Izura Udzir. *Capability-based coordination for open distributed systems*. PhD thesis, University of York, UK, 2006. URL http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.434161.

[35] Lorenzo Bettini and Rocco De Nicola. Mobile Distributed Programming in X-Klaim. In Marco Bernardo and Alessandro Bogliolo, editors, *Formal Methods for Mobile Computing, 5th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-Moby 2005, Bertinoro, Italy, April 26-30, 2005, Advanced Lectures*, volume 3465 of *Lecture Notes in Computer Science*, pages 29–68. Springer, 2005. doi: 10.1007/11419822_2. URL https://doi.org/10.1007/11419822_2.

[36] Lorenzo Bettini, Rocco De Nicola, Rosario Pugliese, and Gian Luigi Ferrari. Interactive Mobile Agents in X-Klaim. In *7th Workshop on Enabling Technologies (WETICE '98), Infrastructure for Collaborative Enterprises, June 17-19, 1998, Palo Alto, CAUSA, Proceedings*, pages 110–117. IEEE Computer Society, 1998. doi: 10.1109/ENABL.1998.725680. URL https://doi.org/10.1109/ENABL.1998.725680.

[37] Lorenzo Bettini, Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Mobile Applications in X-Klaim. In Antonio Corradi, Andrea Omicini, and Agostino Poggi, editors, *WOA 2000: Dagli Oggetti agli Agenti. 1st AI*IA/TABOO Joint Workshop "From Objects to Agents": Evolutive Trends of Software Systems, 29-30 May 2000, Parma, Italy*, pages 1–6. Pitagora Editrice Bologna, 2000.

[38] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Programming access control: The KLAIM experience. In Catuscia Palamidessi, editor, *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, volume 1877 of *Lecture Notes in Computer Science*, pages 48–65. Springer, 2000. doi: 10.1007/3-540-44618-4_5. URL https://doi.org/10.1007/3-540-44618-4_5.

[39] Lorenzo Bettini, Michele Loreti, and Rosario Pugliese. Structured Nets in Klaim. In Barrett R. Bryant, Janice H. Carroll, Ernesto Damiani, Hisham Haddad, and Dave Oppenheim, editors, *Applied Computing 2000, Proceedings of the 2000 ACM Symposium on Applied Computing, Villa Olmo, Via Cantoni 1, 22100 Como, Italy, March 19-21, 2000. Volume 1*, pages 174–180. ACM, 2000. doi: 10.1145/335603.335736. URL https://doi.org/10.1145/335603.335736.

[40] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. XKlaim and Klava: Programming Mobile Code. *Electron. Notes Theor. Comput. Sci.*, 62:24–37, 2001. doi: 10.1016/S1571-0661(04)00317-2. URL `https://doi.org/10.1016/S1571-0661(04)00317-2`.

[41] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. Klava: a Java package for distributed and mobile applications. *Softw. Pract. Exp.*, 32(14):1365–1394, 2002. doi: 10.1002/spe.486. URL `https://doi.org/10.1002/spe.486`.

[42] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gian Luigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. The Klaim project: Theory and practice. In Corrado Priami, editor, *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems, IST/FET International Workshop, GC 2003, Rovereto, Italy, February 9-14, 2003, Revised Papers*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer, 2003. doi: 10.1007/978-3-540-40042-4_4. URL `https://doi.org/10.1007/978-3-540-40042-4_4`.

[43] Rocco De Nicola, Diego Latella, and Mieke Massink. Formal modeling and quantitative analysis of KLAIM-based mobile systems. In Hisham Haddad, Lorie M. Liebrock, Andrea Omicini, and Roger L. Wainwright, editors, *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005*, pages 428–435. ACM, 2005. doi: 10.1145/1066677.1066777. URL `https://doi.org/10.1145/1066677.1066777`.

[44] Xi Wu, Ximeng Li, Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. Klaim-DB: A modeling language for distributed database applications. In Tom Holvoet and Mirko Viroli, editors, *Coordination Models and Languages - 17th IFIP WG 6.1 International Conference, COORDINATION 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9037 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2015. doi: 10.1007/978-3-319-19282-6_13. URL `https://doi.org/10.1007/978-3-319-19282-6_13`.

[45] Rocco De Nicola, Daniele Gorla, and Rosario Pugliese. Pattern matching over a dynamic network of tuple spaces. In Martin Steffen and Gianluigi Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 International Conference, FMOODS 2005, Athens, Greece, June 15-17,*

2005, Proceedings, volume 3535 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2005. doi: 10.1007/11494881_1. URL https://doi.org/10.1007/11494881_1.

[46] Sang Hyuk Son. Synchronization of replicated data in distributed systems. *Inf. Syst.*, 12(2):191–202, 1987. doi: 10.1016/0306-4379(87)90043-3. URL https://doi.org/10.1016/0306-4379(87)90043-3.

[47] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems - concepts and design (2. ed.)*. International computer science series. Addison-Wesley, 1994. ISBN 978-0-201-62433-5.

[48] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. Klava: a Java package for distributed and mobile applications. *Softw. Pract. Exp.*, 32(14):1365–1394, 2002.

[49] Rocco De Nicola, Diego Latella, Alberto Lluch-Lafuente, Michele Loreti, Andrea Margheri, Mieke Massink, Andrea Morichetta, Rosario Pugliese, Francesco Tiezzi, and Andrea Vandin. The SCEL language: Design, implementation, verification. In *The ASCENS Approach*, volume 8998 of *LNCS*, pages 3–71. Springer, 2015.

[50] Marina Andrić, Rocco De Nicola, and Alberto Lluch-Lafuente. Replicating data for better performances in X10. In Christian W. Probst, Chris Hankin, and René Rydhof Hansen, editors, *Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, volume 9560 of *Lecture Notes in Computer Science*, pages 236–251. Springer, 2016. doi: 10.1007/978-3-319-27810-0_12. URL https://doi.org/10.1007/978-3-319-27810-0_12.

[51] Vijay A. Saraswat and Radha Jagadeesan. Concurrent clustered programming. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, volume 3653 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2005.

[52] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15(3):279–328, 2006. doi: 10.1145/1151695.1151698. URL https://doi.org/10.1145/1151695.1151698.

[53] Jiankuan Xing, Zheng Qin, and Jinxue Zhang. A replication-based distribution approach for tuple space-based collaboration of heterogeneous agents. *Research Journal of Information Technology*, 2:201–214, 04 2010. doi: 10.3923/rjit.2010.201.214.

[54] Giovanni Russello, Michel R. V. Chaudron, and Maarten van Steen. Exploiting differentiated tuple distribution in shared data spaces. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Pisa, Italy, August 31-September 3, 2004, Proceedings*, volume 3149 of *Lecture Notes in Computer Science*, pages 579–586. Springer, 2004. doi: 10.1007/978-3-540-27866-5_76. URL https://doi.org/10.1007/978-3-540-27866-5_76.

[55] Vitaly Buravlev, Rocco De Nicola, Alberto Lluch-Lafuente, and Claudio Antares Mezzina. Improving availability in distributed tuple spaces via sharing abstractions and replication strategies. In Ivan Merelli, Pietro Liò, and Igor V. Kotenko, editors, *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018*, pages 302–305. IEEE Computer Society, 2018. doi: 10.1109/PDP2018.2018.00052. URL https://doi.org/10.1109/PDP2018.2018.00052.

[56] Rocco De Nicola, Rosario Pugliese, and Antony I. T. Rowstron. Proving the correctness of optimising destructive and non-destructive reads over tuple spaces. In António Porto and Gruia-Catalin Roman, editors, *Coordination Languages and Models, 4th International Conference, COORDINATION 2000, Limassol, Cyprus, September 11-13, 2000, Proceedings*, volume 1906 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2000.

[57] Antony I. T. Rowstron and Alan Wood. An efficient distributed tuple space implementation for networks of workstations. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume I*, volume 1123 of *Lecture Notes in Computer Science*, pages 510–513. Springer, 1996. doi: 10.1007/3-540-61626-8_69. URL https://doi.org/10.1007/3-540-61626-8_69.

[58] Antonio Corradi, Letizia Leonardi, and Franco Zambonelli. Distributed tuple spaces in highly parallel systems. Technical report, DEISLIA-96-005, UNIBO (Italy), 1996. LIA Series, 1996.

[59] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. *T*uples *O*n *T*he *A*ir: a middleware for context-aware multiagent systems. In Flavio De Paoli, Sara Manzoni, and Agostino Poggi, editors, *WOA 2002: Dagli Oggetti agli Agenti. 3rd AI*IA/TABOO Joint Workshop "From Objects to Agents": From Information to Knowledge, 18-19 November 2002, Milano, Italy*, pages 108–116. Pitagora Editrice Bologna, 2002.

[60] Alan David Fekete and Krithi Ramamritham. Consistency models for replicated data. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2010. doi: 10.1007/978-3-642-11294-2_1. URL https://doi.org/10.1007/978-3-642-11294-2_1.

[61] Elisa Gonzalez Boix, Christophe Scholliers, Wolfgang De Meuter, and Theo D'Hondt. Programming mobile context-aware applications with TOTAM. *J. Syst. Softw.*, 92:3–19, 2014. doi: 10.1016/j.jss.2013.07.031. URL https://doi.org/10.1016/j.jss.2013.07.031.

[62] Dries Harnie, Elisa Gonzalez Boix, Theo D'Hondt, and Wolfgang De Meuter. Programming urban-area applications by exploiting public transportation. *ACM Trans. Auton. Adapt. Syst.*, 9(2):8:1–8:20, 2014. doi: 10.1145/2619999. URL https://doi.org/10.1145/2619999.

[63] David E. Bakken and Richard D. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Trans. Parallel Distrib. Syst.*, 6(3):287–302, 1995. doi: 10.1109/71.372777. URL https://doi.org/10.1109/71.372777.

[64] Alysson Neves Bessani, Eduardo Adílio Pelinson Alchieri, Miguel Correia, and Joni da Silva Fraga. Depspace: a byzantine fault-tolerant coordination service. In Joseph S. Sventek and Steven Hand, editors, *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 163–176. ACM, 2008. doi: 10.1145/1352592.1352610. URL https://doi.org/10.1145/1352592.1352610.

[65] Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. An efficient byzantine-resilient tuple space. *IEEE Trans. Computers*, 58

(8):1080–1094, 2009. doi: 10.1109/TC.2009.71. URL `https://doi.org/10.1109/TC.2009.71`.

[66] Matteo Casadei, Mirko Viroli, and Luca Gardelli. On the collective sort problem for distributed tuple spaces. *Sci. Comput. Program.*, 74(9):702–722, 2009. doi: 10.1016/j.scico.2008.09.018. URL `https://doi.org/10.1016/j.scico.2008.09.018`.

[67] Giovanni Russello, Michel R. V. Chaudron, and Maarten van Steen. Dynamically adapting tuple replication for managing availability in a shared data space. In Jean-Marie Jacquet and Gian Pietro Picco, editors, *Coordination Models and Languages, 7th International Conference, COORDINATION 2005, Namur, Belgium, April 20-23, 2005, Proceedings*, volume 3454 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 2005. doi: 10.1007/11417019_8. URL `https://doi.org/10.1007/11417019_8`.

[68] Danilo Pianini, Sascia Virruso, Ronaldo Menezes, Andrea Omicini, and Mirko Viroli. Self organization in coordination systems using a wordnet-based ontology. In *2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 114–123. IEEE, 2010.

[69] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[70] The Go Authors. *Go release history*. . URL `https://golang.org/project/`.

[71] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in Go. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 865–878. ACM, 2019. doi: 10.1145/3297858.3304069. URL `https://doi.org/10.1145/3297858.3304069`.

[72] Nicolas Dilley and Julien Lange. An empirical study of messaging passing concurrency in Go projects. In Xinyu Wang, David Lo, and Emad Shihab, editors, *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 377–387. IEEE, 2019. doi: 10.1109/SANER.2019.8668036. URL `https://doi.org/10.1109/SANER.2019.8668036`.

[73] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis E. Shasha. The dangers of replication and a solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 173–182. ACM Press, 1996. doi: 10.1145/233269.233330. URL `https://doi.org/10.1145/233269.233330`.

[74] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012. doi: 10.1109/MC.2012.33. URL `https://doi.org/10.1109/MC.2012.33`.

[75] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009. doi: 10.1145/1435417.1435432. URL `https://doi.org/10.1145/1435417.1435432`.

[76] Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, 2013. doi: 10.1145/2447976.2447992. URL `https://doi.org/10.1145/2447976.2447992`.

[77] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[78] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003. doi: 10.1023/A:1022920129859. URL `https://doi.org/10.1023/A:1022920129859`.

[79] Dirk Beyer, Sumit Gulwani, and David A. Schmidt. Combining model checking and data-flow analysis. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 493–540. Springer, 2018. doi: 10.1007/978-3-319-10575-8_16. URL `https://doi.org/10.1007/978-3-319-10575-8_16`.

[80] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers: Principles, techniques and tools. 1986.

[81] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999. ISBN 978-3-540-65410-0. doi: 10.1007/978-3-662-03811-6. URL `https://doi.org/10.1007/978-3-662-03811-6`.

[82] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010. doi: 10.1145/1646353.1646374. URL `https://doi.org/10.1145/1646353.1646374`.

[83] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Inf. Softw. Technol.*, 88:67–95, 2017. doi: 10.1016/j.infsof.2017.04.001. URL `https://doi.org/10.1016/j.infsof.2017.04.001`.

[84] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi: 10.1145/512950.512973. URL `https://doi.org/10.1145/512950.512973`.

[85] Patrick Cousot. Abstract interpretation based static analysis parameterized by semantics. In Pascal Van Hentenryck, editor, *Static Analysis, 4th International Symposium, SAS '97, Paris, France, September 8-10, 1997, Proceedings*, volume 1302 of *Lecture Notes in Computer Science*, pages 388–394. Springer, 1997. doi: 10.1007/BFb0032759. URL `https://doi.org/10.1007/BFb0032759`.

[86] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992. doi: 10.1093/logcom/2.4.511. URL `https://doi.org/10.1093/logcom/2.4.511`.

[87] Patrick Cousot and Radhia Cousot. Abstract interpretation: past, present and future. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 2:1–2:10. ACM, 2014. doi: 10.1145/2603088.2603165. URL `https://doi.org/10.1145/2603088.2603165`.

[88] Armin Biere. Bounded model checking. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of

Frontiers in Artificial Intelligence and Applications, pages 457–481. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-457. URL https://doi.org/10.3233/978-1-58603-929-5-457.

[89] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000. doi: 10.1007/3-540-40922-X_8. URL https://doi.org/10.1007/3-540-40922-X_8.

[90] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of bdds. In Mary Jane Irwin, editor, *Proceedings of the 36th Conference on Design Automation, New Orleans, LA, USA, June 21-25, 1999*, pages 317–320. ACM Press, 1999. doi: 10.1145/309847.309942. URL https://doi.org/10.1145/309847.309942.

[91] Omar Inverso. *Bounded model checking of multi-threaded programs via sequentialization*. PhD thesis, University of Southampton, UK, 2015. URL http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.678178.

[92] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 585–602. Springer, 2014. doi: 10.1007/978-3-319-08867-9_39. URL https://doi.org/10.1007/978-3-319-08867-9_39.

[93] Patrick Cousot and Radhia Cousot. Modular static program analysis. In R. Nigel Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002. doi: 10.1007/3-540-45937-5_13. URL https://doi.org/10.1007/3-540-45937-5_13.

[94] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications*, pages 581–591. Springer, 2015.

[95] Matthew B Dwyer and Lori A Clarke. Data flow analysis frameworks for concurrent programs. *KSU CIS TR*, pages 97–6, 1995.

[96] Uday P. Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data Flow Analysis - Theory and Practice*. CRC Press, 2009. ISBN 978-0-8493-2880-0. URL http://www.crcpress.com/product/isbn/9780849328800.

[97] Ken Kennedy. A survey of data flow analysis techniques. *IBM Thomas J. Watson Research Division*, 1979.

[98] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In David S. Wile, editor, *SIGSOFT '94, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans, Louisiana, USA, December 6-9, 1994*, pages 62–75. ACM, 1994. doi: 10.1145/193173.195295. URL https://doi.org/10.1145/193173.195295.

[99] Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. Static program analysis as a fuzzing aid. In Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis, editors, *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, volume 10453 of *Lecture Notes in Computer Science*, pages 26–47. Springer, 2017. doi: 10.1007/978-3-319-66332-6_2. URL https://doi.org/10.1007/978-3-319-66332-6_2.

[100] Philipp Obreiter and Guntram Gräf. Towards scalability in tuple spaces. In Gary B. Lamont, Hisham Haddad, George A. Papadopoulos, and Brajendra Panda, editors, *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC), March 10-14, 2002, Madrid, Spain*, pages 344–350. ACM, 2002. doi: 10.1145/508791.508858. URL https://doi.org/10.1145/508791.508858.

[101] Ximeng Li, Xi Wu, Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. A coordination language for databases. *Log. Methods Comput. Sci.*, 13 (1), 2016. doi: 10.23638/LMCS-13(1:10)2017. URL https://doi.org/10.23638/LMCS-13(1:10)2017.

[102] Rocco De Nicola, Daniele Gorla, René Rydhof Hansen, Flemming Nielson, Hanne Riis Nielson, Christian W. Probst, and Rosario Pugliese. From flow logic to static type systems for coordination languages. *Sci. Comput. Program.*, 75(6):376–397, 2010. doi: 10.1016/j.scico.2009.07.009. URL `https://doi.org/10.1016/j.scico.2009.07.009`.

[103] Rocco De Nicola, Gian Luigi Ferrari, Rosario Pugliese, and Betti Venneri. Types for access control. *Theor. Comput. Sci.*, 240(1):215–254, 2000. doi: 10.1016/S0304-3975(99)00232-7. URL `https://doi.org/10.1016/S0304-3975(99)00232-7`.

[104] René Rydhof Hansen, Flemming Nielson, Hanne Riis Nielson, and Christian W. Probst. Static validation of licence conformance policies. In *Proceedings of the The Third International Conference on Availability, Reliability and Security, ARES 2008, March 4-7, 2008, Technical University of Catalonia, Barcelona , Spain*, pages 1104–1111. IEEE Computer Society, 2008. doi: 10.1109/ARES.2008.162. URL `https://doi.org/10.1109/ARES.2008.162`.

[105] Chiara Bodei, Pierpaolo Degano, Gian Luigi Ferrari, and Letterio Galletta. Revealing the trajectories of KLAIM tuples, statically. In Michele Boreale, Flavio Corradini, Michele Loreti, and Rosario Pugliese, editors, *Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday*, volume 11665 of *Lecture Notes in Computer Science*, pages 437–454. Springer, 2019. doi: 10.1007/978-3-030-21485-2_24. URL `https://doi.org/10.1007/978-3-030-21485-2_24`.

[106] Chiara Bodei, Pierpaolo Degano, Letterio Galletta, and Francesco Salvatori. Context-aware security: Linguistic mechanisms and static analysis. *J. Comput. Secur.*, 24(4):427–477, 2016. doi: 10.3233/JCS-160551. URL `https://doi.org/10.3233/JCS-160551`.

[107] Fan Yang, Tomoyuki Aotani, Hidehiko Masuhara, Flemming Nielson, and Hanne Riis Nielson. Combining static analysis and runtime checking in security aspects for distributed tuple spaces. In Wolfgang De Meuter and Gruia-Catalin Roman, editors, *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011.*

Proceedings, volume 6721 of *Lecture Notes in Computer Science*, pages 202–218. Springer, 2011. doi: 10.1007/978-3-642-21464-6_14. URL https://doi.org/10.1007/978-3-642-21464-6_14.

[108] Chiara Bodei, Pierpaolo Degano, Gian Luigi Ferrari, and Letterio Galletta. Tracing where iot data are collected and aggregated. *Log. Methods Comput. Sci.*, 13 (3), 2017. doi: 10.23638/LMCS-13(3:5)2017. URL https://doi.org/10.23638/LMCS-13(3:5)2017.

[109] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 106–117. ACM, 1998. doi: 10.1145/277650.277670. URL https://doi.org/10.1145/277650.277670.

[110] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: liveness and safety for channel-based programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 748–761, 2017.

[111] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in Go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1137–1148, 2018.

[112] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent Go by global session graph synthesis. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 174–184. ACM, 2016. doi: 10.1145/2892208.2892232. URL https://doi.org/10.1145/2892208.2892232.

[113] Kai Stadtmüller, Martin Sulzmann, and Peter Thiemann. Static trace-based deadlock analysis for synchronous Mini-Go. In Atsushi Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 116–136, 2016. doi: 10.1007/978-3-319-47958-3_7. URL https://doi.org/10.1007/978-3-319-47958-3_7.

[114] Gowtham Kaki, Kapil Earanky, K. C. Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. *PACMPL*, 2(OOPSLA):164:1–164:27, 2018. doi: 10.1145/3276534. URL https://doi.org/10.1145/3276534.

[115] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977. doi: 10.1145/321992.321996. URL https://doi.org/10.1145/321992.321996.

[116] Helmuth Partsch and Ralf Steinbrüggen. Program transformation systems. *ACM Comput. Surv.*, 15(3):199–236, 1983. doi: 10.1145/356914.356917. URL https://doi.org/10.1145/356914.356917.

[117] Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' java programs. In Raymond Lister and Alison Young, editors, *Sixth Australasian Computing Education Conference (ACE 2004), Dunedin, New Zealand, January 18-22, 2004*, volume 30 of *CRPIT*, pages 317–325. Australian Computer Society, 2004. URL http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV30Truong.html.

[118] Yong-Yi Fanjiang and Jong Yih Kuo. A pattern-based model transformation approach to enhance design quality. In *9th Joint International Conference on Information Sciences (JCIS-06)*. Atlantis Press, 2006.

[119] Rijnard van Tonder and Claire Le Goues. Tailoring programs for static analysis via program transformation. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 824–834. ACM, 2020. doi: 10.1145/3377811.3380343. URL https://doi.org/10.1145/3377811.3380343.

[120] Eelco Visser. A survey of strategies in program transformation systems. *Electron. Notes Theor. Comput. Sci.*, 57:109–143, 2001. doi: 10.1016/S1571-0661(04)00270-1. URL https://doi.org/10.1016/S1571-0661(04)00270-1.

[121] Sesha Kalyur and GS Nagaraja. A taxonomy of methods and models used in program transformation and parallelization. In *International Conference on Ubiquitous Communications and Network Computing*, pages 233–249. Springer, 2019.

[122] Loli Burgueño, Javier Troya, Manuel Wimmer, and Antonio Vallecillo. On the concurrent execution of model transformations with Linda. In Davide Di

Ruscio, Dimitris S. Kolovos, and Nicholas Matragkas, editors, *Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest, Hungary, June 17, 2013*, page 3. ACM, 2013. doi: 10.1145/2487766.2487770. URL https://doi.org/10.1145/2487766.2487770.

[123] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. Grgen.net - the expressive, convenient and fast graph rewrite system. *Int. J. Softw. Tools Technol. Transf.*, 12(3-4):263–271, 2010. doi: 10.1007/s10009-010-0148-8. URL https://doi.org/10.1007/s10009-010-0148-8.

[124] Romain Robbes and Michele Lanza. Example-based program transformation. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2008. doi: 10.1007/978-3-540-87875-9_13. URL https://doi.org/10.1007/978-3-540-87875-9_13.

[125] Eelco Visser. A survey of strategies in rule-based program transformation systems. *J. Symb. Comput.*, 40(1):831–873, 2005. doi: 10.1016/j.jsc.2004.12.011. URL https://doi.org/10.1016/j.jsc.2004.12.011.

[126] Alberto Pettorossi and Maurizio Proietti. Transformation of logic programs: Foundations and techniques. *J. Log. Program.*, 19/20:261–320, 1994. doi: 10.1016/0743-1066(94)90028-0. URL https://doi.org/10.1016/0743-1066(94)90028-0.

[127] Salvador Tamarit, Guillermo Vigueras, Manuel Carro, and Julio Mariño. A haskell implementation of a rule-based program transformation for C programs. In Enrico Pontelli and Tran Cao Son, editors, *Practical Aspects of Declarative Languages - 17th International Symposium, PADL 2015, Portland, OR, USA, June 18-19, 2015. Proceedings*, volume 9131 of *Lecture Notes in Computer Science*, pages 105–114. Springer, 2015. doi: 10.1007/978-3-319-19686-2_8. URL https://doi.org/10.1007/978-3-319-19686-2_8.

[128] Eelco Visser. Program transformation with stratego/xt: Rules, strategies, tools, and systems in stratego/xt 0.9. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised*

Papers, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer, 2003. doi: 10.1007/978-3-540-25935-0_13. URL https://doi.org/10.1007/978-3-540-25935-0_13.

[129] Dan Quinlan and Chunhua Liao. The ROSE Source-to-Source Compiler Infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.

[130] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, 2008.

[131] The Clang Authors. *Clang-Features and Goals*. . URL https://clang.llvm.org/features.html.

[132] Ira D. Baxter, Christopher W. Pidgeon, and Michael Mehlich. Dms®: Program transformations for practical scalable software evolution. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 625–634. IEEE Computer Society, 2004. doi: 10.1109/ICSE.2004.1317484. URL https://doi.org/10.1109/ICSE.2004.1317484.

[133] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002. doi: 10.1007/3-540-45937-5_16. URL https://doi.org/10.1007/3-540-45937-5_16.

[134] Dan G. Waddington and Bin Yao. High-fidelity C/C++ code transformation. *Sci. Comput. Program.*, 68(2):64–78, 2007. doi: 10.1016/j.scico.2006.04.010. URL https://doi.org/10.1016/j.scico.2006.04.010.

[135] Jesús Sánchez Cuadrado and Jesús García Molina. A phasing mechanism for model transformation languages. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007*, pages 1020–1024. ACM, 2007. doi: 10.1145/1244002.1244223. URL https://doi.org/10.1145/1244002.1244223.

[136] James R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006. doi: 10.1016/j.scico.2006.04.002. URL https://doi.org/10.1016/j.scico.2006.04.002.

[137] Rijnard van Tonder and Claire Le Goues. Lightweight multi-language syntax transformation with parser parser combinators. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 363–378. ACM, 2019. doi: 10.1145/3314221.3314589. URL https://doi.org/10.1145/3314221.3314589.

[138] James Koppel, Varot Premtoon, and Armando Solar-Lezama. One tool, many languages: language-parametric transformation with incremental parametric syntax. *Proc. ACM Program. Lang.*, 2(OOPSLA):122:1–122:28, 2018. doi: 10.1145/3276492. URL https://doi.org/10.1145/3276492.

[139] Ion Stoica, Robert Tappan Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord:a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1): 17–32, 2003. doi: 10.1109/TNET.2002.808407. URL https://doi.org/10.1109/TNET.2002.808407.

[140] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent:postgres-r, A New Way to Implement database replication. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 134–143. Morgan Kaufmann, 2000. URL http://www.vldb.org/conf/2000/P134.pdf.

[141] Sameh Elnikety, Steven G. Dropsho, and Willy Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors, *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, pages 399–412. ACM, 2007. doi: 10.1145/1272996.1273037. URL https://doi.org/10.1145/1272996.1273037.

[142] Sérgio Esteves, João Nuno de Oliveira e Silva, and Luís Veiga. Quality-of-service for consistency of data geo-replication in cloud computing. In

Christos Kaklamanis, Theodore S. Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings*, volume 7484 of *Lecture Notes in Computer Science*, pages 285–297. Springer, 2012. doi: 10.1007/978-3-642-32820-6_29. URL https://doi.org/10.1007/978-3-642-32820-6_29.

[143] Rashmi R Karandikar and MB Gudadhe. Comparative analysis of dynamic replication strategies in cloud. *International Journal of Computer Applications*, pages 26–32, 2016.

[144] Cristina L. Abad, Yi Lu, and Roy H. Campbell. DARE: Adaptive data replication for efficient cluster scheduling. In *2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011*, pages 159–168. IEEE Computer Society, 2011. doi: 10.1109/CLUSTER.2011.26. URL https://doi.org/10.1109/CLUSTER.2011.26.

[145] Najme Mansouri. Adaptive data replication strategy in cloud computing for performance improvement. *Frontiers Comput. Sci.*, 10(5):925–935, 2016. doi: 10.1007/s11704-016-5182-6. URL https://doi.org/10.1007/s11704-016-5182-6.

[146] Bahareh Alami Milani and Nima Jafari Navimipour. A comprehensive review of the data replication techniques in the cloud environments: Major trends and future directions. *Journal of Network and Computer Applications*, 64:229–238, 2016.

[147] Prasanna Padmanabhan, Le Gruenwald, Anita Vallur, and Mohammed Atiquzzaman. A survey of data replication techniques for mobile ad hoc network databases. *VLDB J.*, 17(5):1143–1164, 2008. doi: 10.1007/s00778-007-0055-0. URL https://doi.org/10.1007/s00778-007-0055-0.

[148] Doug Terry. Replicated data consistency explained through Baseball. *Communications of the ACM*, 56(12):82–89, 2013. doi: 10.1145/2500500. URL https://doi.org/10.1145/2500500.

[149] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause i'm strong enough: reasoning about consistency choices in distributed systems. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, pages 371–384. ACM, 2016. doi: 10.1145/2837614.2837625. URL `https://doi.org/10.1145/2837614.2837625`.

[150] Mark N. Wegman and Frank Kenneth Zadeck. Constant propagation with conditional branches. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 291–299, 1985. ISBN 0-89791-147-4.

[151] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.