Behavioural Access Control in Distributed Environments

Yining Zhao

Submitted for the degree of Doctor of Philosophy

The University of York

Department of Computer Science

June 2013

Abstract

Applications and services in distributed environments are an increasingly important topic. Hence approaches to security issues in such applications are also becoming essential. Crucial information is needed to be protected properly and mechanisms must be developed for this protection.

Access control is one of the topics that underline security problems. It concerns assuring that data or resources are accessed by the correct entities. A commonly used access control approach is called *access control lists*, which is widely applied in most operating systems. However, this approach has some weaknesses with regard to scalability, and so it is not very suitable for distributed environments that usually have variable populations. *Capabilities* on the other hand offer scalability and adaptability advantages over access control lists. Capabilities are unforgeable tickets that can be propagated between entities, and fit well in distributed environments. But capabilities also have limits due to their simple structure. They grant infinite number of accesses for given types of actions, but are not able to capture sequences and branches of actions, which may be called 'aspects of behaviours'.

In this thesis, behaviour control approaches are introduced, through *Vistas* to *Treaties*. Vistas can provide explicit access control for each component of objects, and provide primitive control over action sequences. Treaties develop behaviour control further by containing behaviour descriptors which can specify those sequencing, branching and terminating aspects, and hence can provide much finer control over behaviours. Because treaties inherit the scalable attributes of capabilities, they also fit well in distributed environments.

An interesting feature in treaty systems is that they allow users to refine the specifications of behaviours and generate new treaties from existing

ones. A number of treaty combinator operations are proposed to realize this functionality, and they are shown to be safe with respect to the security of access control.

A novel issue created by the treaty approach is identified in the thesis. The new problem is called the *duplication problem*, which could cause users being able to gain more permissions than they should have by making copies of unprotected treaties. Any treaty systems must provide solutions to this problem. Three models which solve the duplication problem are proposed, with an analysis of their differences, and advantages and disadvantages.

Treaties are a general concept and in real cases they can be represented in various ways. There are components in treaties that have given a variety of implementation options, and the developers of services and applications can choose to combine these options to fit their special requirements. This makes treaties more flexible and adaptable.

The implementations of concreted treaties and treaty systems are introduced, and these implemented treaties are used to test their behaviour control abilities. Evaluations for different treaty representations are provided to compare their performance. Scalability of treaty systems is also evaluated, showing that treaties are good to be deployed in distributed environments.

Contents

List of Figures

List of Tables

Acknowledgments

As this PhD thesis comes to the stage of consummation, I would like to look back to the path I walked on, and express my gratitude to all the people who gave me plenty of help. I would never have a chance to enjoy my success without you:

- Dr. Alan Wood, my supervisor. Alan gave me the inspiration for my work, guided me throughout the course of the research, showed me different skills and techniques, encouraged me when I felt perturbed, and provided help when I met academic or living problems. There was always a warm welcome when I knocked on his door. It is impossible for me to fully express my gratitude to him.

- Dr. Jeremy Jacob and Manuel Oriol who were my internal examiners. They gave me much support, especially the experience of being questioned, and hence trained me to be better in explaining and presenting my research.

- Prof. Simon Dobson who was my external examiner. His kindness encouraged me a lot, and his comments and suggestions have shown me how I should improve myself.

- Prof. Colin Runciman who took care of me during the period that my supervisor was away from the department. Colin guided me from a different point of view and approach to my research so that I could expand my vision.

- All members of the PLASMA (Programming Languages and Systems) research group. The warm atmosphere and academic discussions always pushed me forward on my path.

- All members and staff in the Department of Computer Science and the University of York that have ever provided various academic, technical and living assistance to me.
- My father Zhao Hong and my mother Huang Lili. They not only raised me up, guided me how to be a good human being, but also provided the financial support during my student years. I can always feel their love for me, and here I want to say the words that I have rarely said because of my introversion: I love you both.
- My relatives, friends, teachers, tutors, classmates, workmates. You will share my memories all my life.
- Finally, my fiancée Zheng Xiaxing. Thank you for your company and patience. I love you.

Declaration

This work is solely that of author, except where cited otherwise. Some parts of the thesis are based on other published work by the author, listed as follows:

1. Parts of Chapter 3 have been published in Proceedings of Euro-Par 2010 Parallel Processing Workshops, Lecture Notes in Coputer Science 6586, September 2010 [72].

2. The initial ideas in Chapter 4 and 5 have been published in Proceedings of 2nd Annual International Conference on Advances in Distributed and Parallel Computing, September 2011 [76].

3. Parts of Section 4.2.2 and 7.2.2 have been published in Proceedings of The 2012 International Conference on Control Engineering and Communication Technology, December 2012 [77].

4. Parts of Section 5.3, 5.5 and 7.3.1 have been published in Proceedings of The Thirteenth International Conference on Parallel and Distributed Computing, Applications and Technologies, December 2012 [78]. An extended version of this paper including contents in Section 7.3.2 has been submitted to the journal of Computer Science and Information Systems (ComSIS).

Chapter 1

Introduction

Access control is an important concept in computer science. It provides safety for different users when accessing resources in systems. In the early days of computer science, programs and resources were only available to individual users, hence there was little need to restrict users' access to resources — they were simply the owner of those resources. But as the systems evolved, programs and resources that were visible to multiple users appeared. These new applications encourage communication and coordination among users in systems, but this introduces the new problem of controlling users' actions, based on their identities. Hence access control concepts were developed as solutions for the problem. In most access control schemes users have to first provide their identities to a central authority, and then the authority (which we may also call the *kernel*) would decide to allow or deny the access request.

However, as systems grow even larger and more complex, the populations of users also increase greatly, and the structure of systems change. Simple access control schemes cannot keep up with the development of these systems, thus new ideas are proposed by researchers for providing more sophisticated functionalities to guarantee the security of systems. This work is part of this trend. The thesis introduces a finer control approach aiming towards the

control of behaviours in distributed environments. These behaviours include predefined sequences of actions and repetitions of executions for some type of actions. The improvement is made by changing the data structure in the capability approach for access control. This work is concerned more about system structure and organization than authentication, verification or validation.

In the following sections, we will introduce the general ideas of access control, pointing out the limitations that appear in traditional access control approaches that motivate our research. This introduction concludes with an outline of the thesis, briefly explaining each chapter.

1.1 Approaches to Access Control

Access control "constrains what a user can do directly, as well as what programs executing on behalf of the users are allowed to do. In this way access control seeks to prevent activity that could lead to breach of security" [56]. In principle, access control works by specifying which user can do what actions on the resources. It is important to note that access control itself is not enough for securing systems from invalid accesses, since it has to be combined with other security services. A key area of security schemes that needs to be provided other than access control, is *authentication*, which allows the kernels to ensure the sender of a request is the actual entity that it declares itself to be. Authentication is necessary before access control happens, because access control initially works with the identity of the requester, and it has to assume that the identity has been correctly verified by the authentication mechanism. Moreover, the whole process needs audit control, which will record and analyse the actions taken by users to observe any adverse behaviours.

Traditional access control is based on the identity of each user who re-

quires access to files or objects. In such schemes each object needs to hold the information for all users that would access it, explicitly defining all valid action types that each user is allowed to perform. This works fine if the scale of the system is small, and it provides a very strict and precise control covering all resources for all users. However, when the applications grow larger and more complex, it is hard to maintain explicit access control information for all users. In a system there might be applications that are available to a number of users, and they have different roles that define different responsibilities and different privileges. It would be infeasible for each object to hold access control information for a large number of users individually, and to maintain the same information for users in the same roles would be a redundancy for the system. New requirements appear as systems evolve [63]. As a result of these considerations, Role-Based Access Control schemes [55, 21] emerged.

Role-based access control (RBAC) splits users of an application into several groups, each of which carries a different role for the application. Each role takes a different responsibility and hence different access rights. With role-based access control, the administrator of the system can simply define a number of roles and assign different permissions according these roles. When new users are added into the system and passed the authentication, they will be specified as belonging to a certain role according to their identity, and gain the appropriate permissions. Now, with such a scheme, each object or application in the system only needs to hold the access control information for a small number of roles instead of the large number of users, thus saving the kernel's load greatly. Although role-based access control loses the individuality of users to some extent, it is still a very suitable model for management in companies and organizations, therefore it has become the major option in the area of access control.

Original access control schemes and the RBAC approach are all based on

central control, as they strongly rely on kernels to administrate the access control information. The most widely used approach in centralized access control is the Access Control List, which implies having kernels possess lists of users or user groups, and defining access permissions for objects for each of the users/groups. As the access control lists adopt the centralized control model, where all the information is strictly managed by kernels, it can strongly guarantee the security of the whole process of accesses. Hence the access control list approach is applied by most services and applications which involve access control, including hardware, middleware and software.

1.2 Distributed Environments: the Motivation

The access control introduced in the previous section faces more challenges when systems become even larger and more complex. In recent years, services and applications in distributed environments [2, 68] have become popular and widely used. Distributed environments may include many different areas, such as parallel computing, distributed computing and cloud computing. Applications in these environments could have a very large number of participants or active processes, and these entities can communicate with others. Meanwhile, the growth of the population of users causes the workload of kernels to increase dramatically as well. It is much more difficult to upgrade the performance of a single kernel/server device than to increase the number of kernels/servers, as there will be a limit to the maximum performance of single chips due to physical limitations. Hence the option of distributing the kernels and users in the systems becomes necessary. All these make strictly centralized control unfit for modern computer developments, since the population of users in such environments (especially open systems) is extremely

variable: there are always new users joining the systems and other users quiting. The changes of users are so frequent that it is very difficult to have the kernels manage these changes — the maintenance of user lists becomes too expensive. On the other hand, traditional access control using centralization is suitable for managements in companies and organizations, but nowadays individual-oriented services and applications (which means their purpose is to serve each individual user) are becoming more and more important. The lack of individuality in RBAC makes it hard to adapt the requirements of these applications. Therefore, a different approach to access control, *capabilities*, has shown its attractiveness.

Capabilities [56] are another access control technique. The major difference between capabilities and centralized access controls is that capabilities are held by users. They are only presented to kernels when the holder of a capability wants to access resources. Hence, kernels no longer need to maintain access control information for each user, but only to check the validity of capabilities when they are shown. Thus this approach reduces the load on kernels greatly. Moreover, capabilities can be propagated among users without the necessity of notifying kernels. This makes the capability approach more dynamic in spreading the permissions for resources than centralized access control approaches, and is particularly suitable in distributed environments. We will introduce the capability ideas and their advantages and drawbacks in detail in Chapter 2.

Although capabilities show some attractive properties for access control in distributed environments, there are still aspects that could (and should) be improved. In capabilities there are only independent permissions. They cannot, for instance, define specific sequences of actions. In real cases we often have to do things in a number of steps, and the order of these steps cannot be altered. These ordered actions constitute procedures, or what we call the *behaviours*. Simple capabilities cannot specify the control of

behaviours in the way that centralized access control could, where the kernels could define and change the valid permissions for users or user groups, as all access control information is held by kernels. However, in the capability approach access control information is stored in capabilities that are held by users, hence it is difficult for kernels to control the sequence of user actions in the same way as centralized control. To provide behaviour control in distributed environments, we need further research and development, and that is the main motivation for the work presented in this thesis.

In this work, we will explore the capability approach to access control in distributed environments, and propose new ideas and approaches that can cope with the behaviour control aspect addressed above. The development of behaviour control in our research is divided into *Vistas* and *Treaties*, which aim towards a finer behaviour control. Vistas extend the capability approach by splitting the control to per-attributes and per-functions in the target resource, and Treaties provide finer behaviour control by specifying the sequences, branches and terminations of actions to target resources. There are new security problems that arise from the components in the new approaches, and possible solutions are presented to overcome these problems.

1.3 Contributions

The main goal of this work is to develop new approaches for coping with behaviour control in distributed environments. The approaches of Vistas and Treaties are proposed that aim to provide the functionality of behaviour control. In this thesis we present the following:

- The concept of Vistas and Treaties, how they are structurally evolved from capabilities, and why these evolutions are beneficial.
- Combinator Operations for vistas, and for treaties, which form new

access control and behaviour control specifications. These operations provide the functionality of allowing users to refine their behaviour specification after the vistas or treaties are produced, and hence further increase the dynamic control over accesses and behaviours.

- A proof of the correctness of treaty combinator operations using the idea of behaviour sets that shows that refining treaties with treaty operations will not destroy the safety of behaviour control.

- A new issue called the *duplication problem* arising from the behaviour control approaches, suggesting solutions that solve this problem. The performance of the candidate solutions for the duplication problem are also tested and compared.

- The various implementation options for treaties and treaty systems that allow developers and service providers to satisfy their particular requirements.

- Implementation examples of capabilities, vistas and treaties. We have realized the functionality of behaviour controls and the refinement of behaviour specifications using treaty operations, have evaluated the correctness of behaviour control using treaties, and have evaluated the computational requirements for implemented treaty components. Finally, we have evaluated the scalability of treaty systems to show the practicality of treaties.

1.4 Thesis Organization

Here the content of each chapter will be briefly introduced by their organizations, providing an overall view of the thesis.

Background In this chapter the required knowledge will be introduced. The major content of this chapter will be an examination of the access control schemes and especially capabilities, which are an access control approach that is particularly suitable to be applied in distributed environments. The advantages and disadvantages of capabilities will be discussed, and we point out the limitation that capabilities do not provide adequate behaviour control.

Vistas This chapter will propose an approach called *Vistas* that extends the idea of capabilities, and provides a finer access control scheme. We show the extra functionalities of combining vistas with defined operations, and suggest some use-cases that would benefit from vistas. Vistas show progress towards behaviour control but which is still not precise enough, thus indicating that further development is needed.

Treaty Concepts The ideas of capabilities and vistas are extended further to *Treaties* in this chapter. We introduce the concepts, definitions and structures of treaties in an abstract way. Like Vistas, Treaties also have combinator operations to change the specifications of behaviours. There emerges a new issue that treaties cause, called the *duplication problem*. We discuss the reasons and the situations that might cause this problem.

Representations of Treaties This chapter discusses treaties in a more concrete way. Several components of treaties and treaty systems can have different implementation options that can be chosen by developers and service providers to satisfy their special requirements. An example of a complete treaty system implementation is shown in the chapter, in which all options have been fixed.

Implementation In this chapter the way the behaviour control scheme of

treaties is implemented using Java is presented. We introduce components of treaties and systems to be implemented as classes, and some of the field values and methods are also discussed. The key algorithms of treaty combinator operations are illustrated by pseudo-code.

Evaluation We evaluate our treaty systems according to their integrity on behaviour control, their time and space consumption, and their scalability in distributed environments. The efficiency of combinator operations in Finite-State Machine (FSM) and Regular Expression (RE) representations are compared, and the performance of different models to solve the duplication problem are also evaluated.

Conclusion This chapter reviews the whole thesis and emphasizes the essential contents. It addresses the contribution of this work again, and discusses topics that have not been included in this thesis but could be a good direction for future work.

Chapter 2

Background

In this chapter the background and literiture that contributes relevant knowledge is presented. A main area to be introduced is *Capabilities* that are a dynamic access control approach for distributed environments. We will look at the positives and negatives that the capability approach brings, and different uses and modifications that are applied on capabilities for various cases. It also suggest the limitations which indicate the need for further researches.

2.1 Capabilities vs ACLs

The prime work of this project will largely be based on the idea of capabilities. The term *capability* in the context of computer science generally refers to object access control. The concept was invented several decades ago, and the debate on the applicability and effectiveness of capabilities has also lasted for a long period. Therefore, to illustrate the proposed working directions of the project, it is necessary to look at the concept of capabilities and capability-related research work.

In the context of security systems in computer science, the word 'capability' means a particular approach to address and protect data or resources.

user	Read	Write	eXecute
Alice	-	-	-
Bob	√	-	√
Carol	√	√	√
David	-	-	√

Table 2.1: Example ACL for a file *f*

The main classification of this approach is access control. In the traditional methods of access control, the most commonly used approach is the Access Control List (ACL), whose variants have been applied to Unix and many other systems [70, 10]. This mechanism is implemented by attaching an access rights list to items of data, each list specifying the particular access rights for every user or process in the system. These lists are used to inform the kernel whether an action to a file, say, from a user is valid. Table 2.1 shows an example of an access control list for an individual file. Assume this list has been applied to file *f*, then any access from Alice to *f* will be denied, and any access from Carol is allowed. David only has the permission to execute while modification from Bob is invalid.

The capability approach also provides an access control mechanism between users and data, but it does this the other way: instead of attaching the access control information on the data side, capabilities are stored on the user's device or storage spaces. Capabilities are unforgable keys that are created by the kernel, and can be passed among users or processes. They are not allowed to be modified, but reproduction is legal [46]. In the classical approach a capability contains two major parts: a reference to the particular data item, and a set of access rights that this capability is granting. The use of capabilities acts like ticket verification. Suppose in a capability based system there is a user or process that wants to read a file *f*, it must possess a capability *c* which has the reference to *f*, and in *c* the `read` right is in the

rights set. Then the user or process may hand the capability c to the kernel and tell it that it want to access file f with a `read` operation. After the kernel has checked the validity of c (that it is not forged) and the `read` right is in the set of access rights, this `read` action can be allowed and performed. Fig. 2.1 shows the capability approach in the same case of that in Table 2.1.

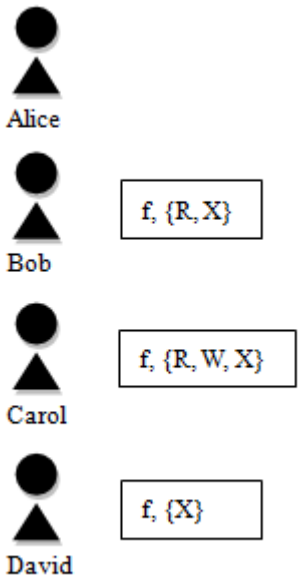


Figure 2.1: Example Capabilities for a file f

The key difference of the capability approach from the ACL approach is that, these capability objects are possessed by the subjects (users or processes), and subjects are responsible for propagating capabilities. Initially, a capability c is created by the kernel when a new file f is created, and the kernel will pass this capability c to the creator of the new file. Usually the new capability c contains the full set of access rights to file f at this stage. Then, if the creator of the file wants to share f with other users or processes, it can pass a capability c' referencing for f to others. This capability c' is

generated from c. They have the same reference to f, but the set of rights in c' can be either the same as that in c, or it can be a subset. Then other subjects having c' can also pass capability c'' generated from c' to others, and so on. The information contained in capabilities is visible to their possessors, so that the possessors know what access actions are valid to them.

The original capability idea came from Dennis and Van Horn's work back in the mid 1960s, and it was illustrated in their work [17]. Although this paper has been widely recognized as the introduction of the capability idea, it is in fact not purely about this, and the capability concept is rather like a by-product. In the paper the authors discussed the Multiprogrammed Computation System (MCS), which is a kind of operating system related to programming and concurrency. Dennis and Van Horn used capabilities as the addressing and protection tool. In an MCS there are many processes running at the same time, and each process should have a list of capabilities called a *C-list*. Capabilities in the C-list contain all references to the data or computing objects that the process needs. It also contains the set of access rights of combinations of readable, writeable and executable. The paper defines a computation as "a set of processes having a common C-list such that all processes using that same C-list are members of the same computation". In other words it is the group of processes computing over the same objects. Capabilities are used here to protect data from incorrect or meaningless modification. If any process wants to write over a piece of data D through some routine, there must be the capability in its C-list that shows the validity of writing D.

There are some important rules that have been generally agreed. The most significant one is that: one can never edit the contents of any capability. This basic rule shall always be applied. Two major principles are contained in the rule. First, the possessor of capabilities cannot replace the references to the object by other references. Second, one cannot increase, reduce or

replace the rights in the set of rights part of capabilities. The behaviour of increasing and replacing permissions are especially forbidden. The reason for this rule is that if the possessor can replace the references in capabilities, he is able to easily access an object to which he has only restricted actions by creating a new object with full permissions and change the reference of the new object to the restricted object. Similarly forbidden in the behaviour of changing set of rights, as the possessor could *increase the available access rights* which go beyond the level issued by the kernel. Although reducing the rights in a capability seems to do no harm to the capability system, it is an action that brings no benefits to the holder of this capability, thus this can also be omitted.

However, capabilities can be changed to generate *new* capabilities, and these are referred as operations on capabilities. The set of rights in a capability can be increased in case of its possessor also having another valid capability referring to the same object, and has some issued rights that are not in the first capability. We call this a *combination* of capabilities. The new capability generated by the combining operation results in a reference to the object that the two combining capabilities both pointed to, and a set of rights that is the union of the right sets of the two combining capabilities. This combination operation is useful when the possessor gained two capabilities from two paths or at different times.

Another valid capability operation is *subtraction*. This operation can be performed from any capability that does not have an empty set of rights. The capability derived by a subtraction operation contains the reference pointing to exactly the same object as the original capability, and a subset of the set of rights from the original capability. This has no benefits to the holder himself, but it can be used in a case when a user is willing to pass a capability to share the object with others, but he does not want to give the full permissions that he has to other users. With the subtraction operation, the possessor can

keep the same capability he has and give a restricted version out.

The capability approach in general is just a concept. In practice, there is more than one way to implement a capability model, and because of their differences, they can be used in different applications.

A simple way of using the capability features without any complex mechanism is to follow the original invention, which is proposed in the original work [17], in the age before object-oriented ideas had become popular. In this style, capabilities are just a set of attribute values grouped together in each process. There are no real organized structures defining capabilities, and they are statically handled by processes themselves. To use a capability the process needs to find out the correct attribute values among the locations of all capabilities that are reachable, and understand what these data represent. This approach is appropriate for a closed system, where there are no dramatic changes in computing processes, and processes and their executing styles tend to be homogeneous. Otherwise it requires some interpreters to work for the synchronization.

Another way of implementing capabilities, in a network environment, is to adopt the ideas from network security: encryption. Gifford [25] introduced the idea of sealing information using cryptography to enhance the secrecy and authentication of the system. This is done by first generating a pair of asymmetric keys, and the owner of the information uses one of the pair of keys to encrypt the secrets as a 'sealed' object, and then places it in some publicly accessible storage. Only by using the other key in that pair, the information in the sealed object can be decrypted. Thus people possessing the decryption key can read the information, while others will only get the name and size of the object but nothing else. There are many operations for combining and redirecting features that were introduced in the paper as well, and these operations make this approach more feasible for information sharing among multiple users having different key pairs. This cryptography

can be implemented to apply for many different uses, including access control lists, capabilities and information flow controls.

Treating capabilities as objects is a third way of representing the capability approach. This is applicable for those modern programming languages with object-oriented features. Each capability is an object now, and we can define capabilities using special class declarations with clear attribute values, using different levels of protection over the functions to restrict the accessibility to these field values. Because capabilities are actual objects in this way, they can be passed among users and processes conveniently, so it really corresponds with the property of user propagation of the capability approach. The accessibility of capability objects needs to be carefully defined. The users and processes are allowed to do the combining or subtracting operations over capabilities, but they are allowed neither to directly access the field values that represent permissions, nor to change the validity of capabilities. In general, the approach of treating capabilities as objects is quite appropriate for open systems that have frequently changed population.

2.2 Issues of Capabilities

As has been mentioned, the major difference between the capability approach and the ACL approach is the place where access control information is stored. We can have a comparison between these two approaches and thus find the advantages and disadvantages for each one.

In the ACL approach, the information is stored on the resource side, and the kernel has complete knowledge for this. Every time a user comes up with an access operation to a certain file, the kernel will check the access list of that file to see if the user is on the list, and then to check if the operation type has been granted to this user. The good point of this approach is the control power of the kernel. The kernel can actively change the access information

and user groups, and this gives an easy control mode.

However, there is a problem with this: if an unknown user comes to the system, any operations from him will be denied, as the user is not on the access control lists. Therefore, to allow new users to join the system, it must update all lists of all files every time the user set changes. It is clear to see that the cost could be high, especially in systems where there is a big possibility for changes of users. A common solution is to have user groups, which means users can be divided into administrators, common members and visitors and so on, and in the access control lists only these groups are recorded. When new users join the system, the kernel can simply place them into the proper group and there is no need to update ACLs. But this solution only deals with a small number of groups, so it does not support a wide variety of users, and thus is not fit for complex systems.

The capability approach on the other hand has the opposite features compared with ACLs. Since capabilities themselves contain the access information, they have been released from the control of kernel. The propagation of the capabilities is the job of users, and the kernels need only do the work of producing capabilities, passing them to the initial owners of the resources, and checking the validity and issued permissions of the capabilities that are handed in by users who require access to resources. It is clear that in this approach the kernel side needs to know nothing about the users and their changes. The entry and exit of users has no effect on the kernel's operations, thus a more dynamic system has been achieved, and the system structure becomes more flexible.

The capability approach also has its own shortcomings. The supporters on the ACL side argue that there are some major problems that the capability approach cannot overcome easily. First, people say that there are usually many more resources or files than users, so to implement the capability approach means that every user or process has to maintain a huge list

	resource1	resource2
Alice	√	√
Bob	√	-
Carol	-	√

Table 2.2: Reference Relations

of resource references, plus all issued permissions along each reference, and this could be a space problem. Second, it is widely believed that capabilities cannot enforce confinement [5, 39, 40]. The *confinement problem* refers to how to prevent systems from leaking data to untrusted users or subjects. Because the propagation of capabilities is managed by users, it cannot be controlled by the kernel, so people worry that capabilities can go anywhere, even to someone who is not trusted [44]. Another doubt over the capability approach is called the *revocation problem* [27, 41]. People believe it is not possible to get back the capabilities you gave out to somebody previously, as they now have the complete control over those capabilities, and you cannot destroy them on other users' machines because of security reasons [28, 42].

With respect to the above comments, Miller and co-workers have managed to give answers in their work [47, 48, 49]. Miller was trying to show that these major problems of capability approach can be overcome using some special mechanisms. First he stated that capabilities have more optimized referencing structure than the traditional ACL approach. Table 2.2 shows referencing relations between a group of users and a group of resources. It used to be thought that the referencing structures of the two approaches had little difference, as it is just the question of in which way to look at the table: a vertical view for the ACL approach and a horizontal view for the capability approach. But Miller argues if we take a deeper view of the actual references between these two groups, the difference is clear (see Fig. 2.2). In the ACL approach, references from both directions are needed: the users

need to know where the resources are and the resources need to know who can access them. But in the capability approach this has been simplified into one direction: the capabilities themselves contain both references and the access control information. This is believed by Miller to be an optimization.

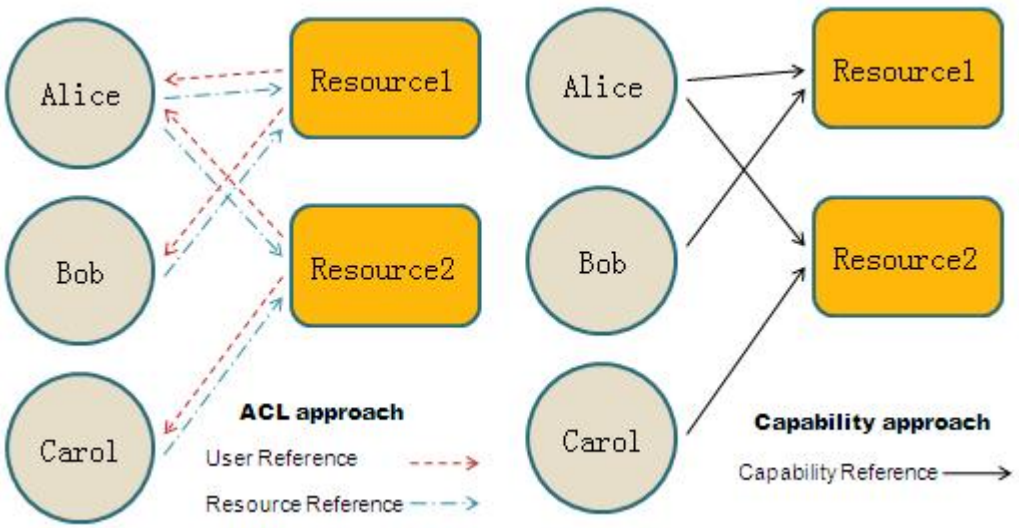


Figure 2.2: Actual referencing relations

Secondly, for the confinement problem, Boebert emphasized the concept of *-property [9]. Briefly speaking this property means that in a secure system, information should flow only from the side of a lower security level to the side of an equal or higher security level, but never the other way. Boebert showed that an unmodified capability machine cannot gain this *-property. Miller's counter argument is illustrated by: assume Carol possesses the resource and Alice has the capability to access them. Now Alice wants to pass this capability to Bob. In order to do this, Alice must be herself authorized to both Bob and Carol. So if Alice passes something to Bob, it means that Alice and Bob have already built a trustworthy connection, which has already gained the confinement. All the trustworthy connections build up a safe environment for capabilities to be propagated, which Miller

calls the Arena. To enter the arena, users must have agreed some terms of entry for confinement. However, it can be argued that this is not a pure unmodified capability approach.

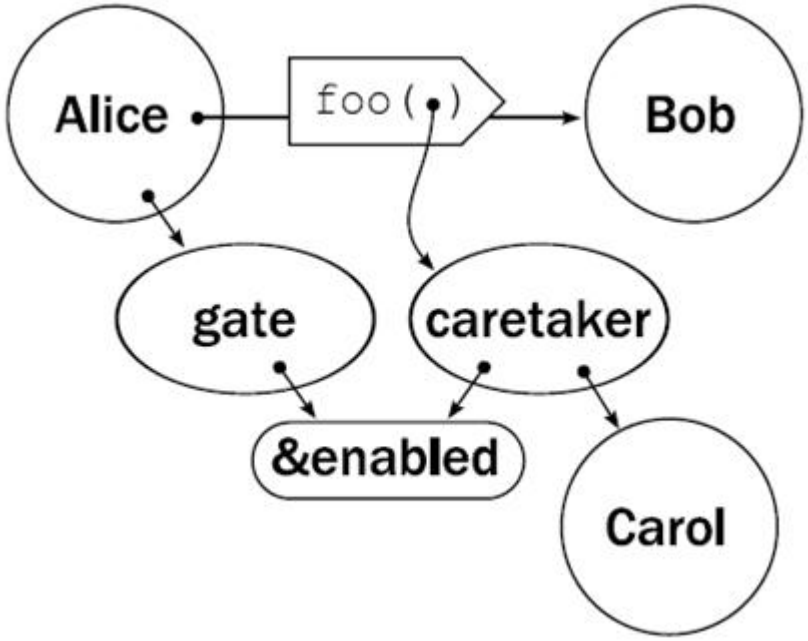


Figure 2.3: Using Caretakers to Solve the Revocation Problem

Finally, Miller discussed the revocation problem. He stated that by introducing a small modification, this problem can be solved (See Fig. 2.3 that is shown in Miller's Work [49]). To pass Carol's capability to Bob, Alice can modify the capability: it carries a caretaker, which will always check a gate value before it can access Carol. This gate was set up by Alice, and it is Alice's business to switch this gate between enabled or disabled. When disabled, the capability to Carol is ineffective. Again, people on the ACL side argue that it is not a pure capability approach but using some ACL techniques, while people on the capability side suggest that at least this is an advantage over attaching access control information to resources.

The debate of ACL vs. capability with respect to their advantages and shortcomings could still continue for a long time.

2.3 Authentication

Authentication is a term with the meaning of confirming that something is the entity it claims to be [52, 12]. In other words, it is the problem of how to know that a subject (human, computer user, process, etc.) is itself. Authentication is a general term in the field of security in computer science, especially in the topic of networking. In traditional access control models, the authentication processes happens just before the access control processes. In the original pure capability approach, authentication is not such a big issue, as the central unit needs only to check the validity and permission sets of the capabilities themselves, but nothing about the agents who are presenting the capabilities. However, considering the confinement problem and other security problems, it is often necessary to modify capability based systems to recognize the holders. Thus it is worth looking at the authentication issues that may relate to capabilities.

There are some issues that show the need for authentication for capability systems. The confinement problem was introduced in the previous paragraphs. It is never a good idea to give too much permission to untrusted users, so before any propagation of capabilities, an authentication process is required. In this case, the authentication processes should happen at the users' end, but depending on the implementation of the system, the kernel may also join in this mechanism, e.g. some capability systems could be implemented so all propagations of capabilities must be confirmed by the kernel. Another situation in which the kernel needs to know the holders of the capabilities is where systems need to implement garbage collection. Before removing garbage resources the kernel must make sure there are no refer-

ences to the garbage, but in classical capability approaches the kernel does not have such knowledge. Therefore changes might be needed when kernels need to know whether users possess references to some extent (although we do not want the kernel to have full knowledge of users or processes, as this may destroy the scalability of the capability approach).

One of the ideas involving authentication issues is to assign each possible possessor an identity. This has been proposed in Li Gong's work [28], and in this paper the author suggested the mechanism of identity-based capability system (ICAP), which added an identity value into each capability in the system. The identity is created by the kernel or server when a new object is created by a user, and it is a value generated by a one-way function. The parameters of the function include the authentication information of the user (maybe the user's ID), the new object, issued permissions and an issuing number that is generated and stored together with the new object. The key difference of the ICAP from traditional capabilities is that any other user with a different ID cannot use this capability to access the object, because when accessing is being required the server will check the validity regarding the user's ID. In this way it enforces the control of the server over capabilities and prevents the use of stolen capabilities. To legally propagate the ICAPs, the original user will issue a message explaining that he agrees to pass the capability to the receiving user. Then when the receiving user wants to use this capability, he needs to present both the capability and the message to the server, and the server will check the validity of the capability and message. If all these tests are passed, the server will create a new capability for the receiving user, by using his ID to assign the issuing number, and after this the receiving user can use his own capability to access the object. In this paper the author also proposed an enhanced version of ICAP introducing the ancestor algorithm. The idea is that every time a capability is propagated, the server will generate a new issuing number using the previous issuing

number and the sending user's ID. Therefore when a user wants to revoke any capabilities that he passed out previously, he can simply inform the server and the server can reject any capabilities of his descendants. The ICAP system is claimed to be able to solve confinement and revocation problems to a certain degree [28].

However, authentication is a much more complex question than this. It is never guaranteed that only capabilities can be stolen. Assuming the user's ID has been leaked to the threatening agent, these protections could be greatly weakened. Moreover, assuming that a threatening agent has got the full information of a user, the agent can completely pretend to be the user. There are no ways to discover this fake identity, unless the real user and the threatening agent appear in the system at the same time. But the point is, this is an uncommon situation. It is very difficult to steal all information even from a low-level protected user device. On the other hand, this is a situation beyond all protections. If the full information has been leaked to another agent, no protection mechanisms can deal with this threat - not just capability systems.

Another problem is, suppose a user carelessly lost his identification, how can he prove to the server that he is the valid user but has lost the identifying information? How does an identity prove him to be himself? The general problem of authentication is too big to discuss within the scope of this work, and we assume procedures of authentication happens before procedures of access control. Therefore, this work will not focus on authentication problems.

2.4 Uses of Capabilities

Although there are always debates about capabilities, their advantages are still attractive. Thus, since the invention of capabilities, this concept has been

applied to many fields. The most straight forward application is following the original ideas from Dennis and Van Horn [17] of operating systems. Capabilities are used as an alternative to access control lists to protect resources in the system. Any users or processes that want to access data in the system must provide valid capabilities in order to be granted access. Henry Levy wrote a book in 1984 [45] in which he demonstrated many capability-based computing systems, and discussed their histories and evaluations. These systems include some early examples, the Plessey System 250, the Cambridge CAP, Hydra, StarOS, IBM System/38 and Intel iAPX 432. It is obvious that some major research institutes and manufacturers gave attention to the idea of capabilities. Kain and Landwehr [38] give a set of definitions of different concepts in capability-based systems.

A more recent example was proposed by Shapiro and his colleagues. They wrote several papers introducing a capability-based operating system called EROS [59, 60, 61, 62]. In the evaluation part of the paper the authors made a comparison between EROS and Linux over some major measurements (pipe latency, pipe bandwidth, create process time, context switch time, heap growth, page fault and trivial syscall). From the result of their experiments, 6 out of 7 measurements of EROS return a better performance than those of Linux. Shapiro and colleagues believe that this proves that capability-based systems can do as good as Linux, without any specialized hardware assists.

Another modern use of the capability concept is in memory management. This relates to the compiling and running of programs. The researchers on this topic use capabilities as a protection for memory regions against unmatched type accesses. In region-based memory management the memory is divided into regions and does not trace the structure of the heap, so it avoids some complexity for garbage collection. Walker et al. [69] show a way of using capability protection to provide safe region operations, and Charguéraud and Pottier [13] extend this to include the idea of type preservation. The

authors defined their variation of capabilities as a pair $\{\alpha : \theta\}$, where α represents a region in memory, while θ describes the type of the structure in that region. The abstraction can be clearly seen here: it is no longer a set of rights, but a type matching. Only if the capability represents the same type or sub-type of the type in that region, can the access be validated. This checking can be performed at both compile and run time.

A wider use of capabilities as a method of access control is in the field of distributed environments and systems, and especially in open systems. Open systems are systems where users or nodes may join and leave frequently, i.e. they are opened to all users. This feature of the system means that the population can be large and extremely unpredictable. This could be a disaster for the traditional ACL approach of access control as it would have to update and maintain a huge list again and again. The advantage of the dynamic properties of the capability approach becomes obvious in these systems, thus it is a great choice for distributed systems. Another reason that might make distributed systems prefer capabilities is that, in a distributed open system there might be many more users than resources, which addresses the point that people used to treat as a weakness of capabilities when there are more resources than users. In an open system, having everybody carry a small set of capabilities seems to be better than letting the resources hold huge lists of users, in both storage and load comparisons.

LINDA-like tuple space systems [8, 23, 24, 66, 67] are good models of distributed open systems. It has been shown that to build a private channel in open system could not be achieved in traditional ways. Alan Wood proposed a mixed system combining ACL and capability techniques to deal with this problem in 1999 [71]. A comparison between ACL and capability approaches in LINDA-like systems is also contributed in this paper.

Recent work on the use of capabilities in distributed environments has been reported by Gorla and Pugliese [29]. They proposed a language call

μKLAIM, which is a modification to KLAIM. The original KLAIM also adopted a tuple space approach, but it has the weakness of static access control which cannot handle the dynamic changes during the programs executing while nodes join and leave. So the authors exploit capabilities to overcome the weakness, and propose a combination of dynamic and static approaches.

An interesting use of capabilities was explained in a paper in 2008 [57]. In the paper the authors took capabilities into an Internet-based virtual environment called Second Life.[1] This is a kind of network program where users can interact, communicate and trade in a shared environment. Participants can use the provided tools to create their buildings and vehicles and so on. Second Life uses the ACL approach to restrict the accesses. The authors of the paper experimented with this kind of virtual environment using the capability approach to replace ACLs. This work also uses caretakers to deal with the revocation problem. One of the main points that was highlighted in this work is the idea of *facet*, which is a pattern "to create capabilities that only allow certain messages to be sent to the target object". This starts to bring the concept of *behaviour* into capabilities. It allows users to predefine certain simple access patterns and reject any messages outside the facet definition. However, this is still a simple model of behaviours, as it only defines the type of accesses without any order or sequences.

Capabilities can also be used to integrate external uniqueness of object-oriented systems. Haller and Odersky in their work [30] suggest adding some semantics on the language of Scala [31] to ensure the uniqueness of the messages in distributed environment. This work also aim to "ensure race safety with a type system that is simple and expressive enough to be deployed in production systems by normal users".

[1] http://www.secondlife.com

2.5 Modifications of Capabilities

Recall the original structure of the capabilities that has been agreed for tens of years: each capability consists of two parts of a reference to an identified object and a set of rights (permissions). This is the standard representation of capabilities. However, this is not a fixed definition. Capabilities can be structured differently, and various workers have modified the representations of capabilities to fit their own requirement.

ICAP [28], which has been mentioned in previous sections, is a good example illustrating the modification of the structure of capabilities. In this concept, there are three parts that constitute an ICAP capability: reference to the object and the set of rights as usual, but a new identifying value which is generated from the former two parts of the capability, the user's identity and a random value that is stored together with the object. This identity feature is added to fulfill the aim of solving the confinement and revocation problems.

Another example of the modification of capabilities is called *split capabilities* [42]. In this work, capabilities are split into two parts: file locks and keys (in original words, " handle to the resource being accessed and a handle to a separate resource representing the access rights being requested"). The *lock* part is attached to the files. These locks are generated by the system, and define the subjects able to unlock these locks. The lock part of the split capabilities acts rather like an ACL feature, and it groups the subjects by rights. The *key* part can be propagated by users, thus it follows the approach of traditional capabilities. Every time a new access permission is assigned, the system will generate the key with its value (ID), and add this value to the lock part of the object that allows the access, and then distribute this key to the user who is allowed to perform this accessing action. Now the user is free to pass this key, or new keys generated from it, to other users.

When a user requests the system to access a certain file, he should show his key. The system then checks the value of the key to see if it is on the lock part of the file, without knowing which user is doing this request and how he got the key. If the value is correct, the system unlocks the protection for the file and the access is granted. An interesting feature of split capabilities is that each *key* also has an access action and that is the 'Destroy' operation. It performs also as a lock, and can be unlocked by other keys. The split capability approach is claimed to be able to achieve the *-property [9] and solve the revocation problem. However, the disadvantage of this approach is that there is still information attached to the objects, thus it decreases the flexibility to some degree. But it still an improvement over ACLs as the locks do not list all users, but only keys which could be much fewer than the number of users, compared to the traditional ACL approach.

Multicapabilities [65] modifies the capability structure in another way. This work proposed by Udzir is based on LINDA systems [8, 23, 24, 66, 67]. LINDA systems are distributed systems that adopt tuple space ideas. In tuple spaces there are tuples which are unnamed entries, and users or processes input and output tuples by using templates. It is clear that the traditional capability approach cannot be directly applied to LINDA systems, as the unnamed tuples cannot be referred by capabilities. Udzir solves this by introducing the idea of multicapabilities, which instead of referring to a single named object, they can match a number of tuples in the same template. The structure of multicapabilities is given by a triple $\{u\ t\ p\}$, where p is the set of permissions, t stands for the template and u is a unique tag to identify this group of tuples (compare to the structure of traditional capabilities: a pair $\{o\ p\}$, as o for object reference and p for set of permissions). The work of multicapabilities moves towards introducing the class concept to capabilities, and the contributions of this work have applications in the fields of garbage collection, deadlock breaking, private channels and replication and caching.

2.6 Limitations of Capabilities

As has been mentioned repeatedly, the main use of capabilities is access control which can be applied in different environments, in either centralized or decentralized systems. They can also protect resources or memory regions and filter valid or invalid accesses over different permissions. However, the potential of the capability approach is greater than just access control. We would like a more powerful dynamic control for accesses and other actions, and maybe some of these requirements cannot be expressed using the traditional capability concept due to the simple structure. That is, there are some limitations to capabilities.

The confinement and revocation problems have been discussed earlier: they are the well known limitations. Now let us look at something more. Traditional capabilities use the notion of permission sets. There is a naive premise: these permissions have been assigned permanently, and users can use their capabilities to access objects at any time and any number of times, if the capabilities are valid. This is good enough where it meets requirements of systems. However, people often do not do things like this, as we have different situations and conditions, and we do different behaviour at different times. Take an example: assume one has just opened a new bank account, so in the capability view, he now should have the rights to deposit and withdraw money to and from the account. But in fact it is not allowed to withdraw anything until one has saved a certain amount into his new account. Apparently, in this case it is a matter of sequence of actions, and the objects usually have their states. In different states there are different possible actions. These sequences of actions cannot be represented by traditional capabilities. In order to make capabilities able to handle these behaviours, an upgrade is necessary, and this is exactly the topic this work is looking at. We will explore this in following chapters.

2.7 Session Types

Session Types [33, 43, 54, 58] provide mechanism that "allows the specification of a protocol to be expressed as a type" [58]. In many communications between two parties, the interaction is likely to last for some time, and hence they should be treated differently from a 'one-time only' style of communication (e.g. call-return, pass-away, etc.). The session type theory can define communications with a prescribed conversation plan, which specifies the types of individual messages and the allowable sequences of messages. When a communication with session types is established, the communicating parties will have expectations of the following message exchanging pattern and the type of values being sent and received. With session types, it is ensured that the two parties are communicating with the same and correct protocol; that the sent messages or values are received by the opposite party; that the types of messages and the communication patterns are agreed in advance; that branching and looping flow control are correctly implemented by both parties [54].

The session type theory provides an approach that describes the types and sequences (including branching and looping) of messages in interactions. This is similar to the concept we defined as behaviours, so it is worth coupling with this project. However, the session type theory has a different orientation from the research that is presented in this thesis. Firstly, the session types define types and sequences on communications between two parties, while the behaviour access control evolved from the capability approach concerns data being accessed by multiple entities, who may share common behaviour patterns and propagate access permissions. Secondly, session types are mainly used for verifying the correct implementation and use of communication protocols at compile-time, while the capability-like approaches are more practical components that have their effectiveness at

run-time.

2.8 Summary

Capabilities, being an approach to access control, has been introduced in this chapter. Compared with the traditional access control approach of ACLs, capabilities show less dependence on kernels and central controls. In such systems, capabilities are components that contain access control information, and provide it to kernels when necessary. They are held by users and processes, which means there is less load on kernels. Moreover, the holders of capabilities can pass them to other entities in the system, such that the permissions for accessing objects can be propagated instantly when they are required, and do not need to inform kernels (or less information is needed to notify kernels). These makes capabilities a better choice of access control approaches in parallel and distributed systems. Capabilities have a number of advantages over ACLs, and good properties of dynamic access control, thus the idea of capabilities has been applied to various areas such as operating systems, building private communication channels, etc.

However, capabilities also have some drawbacks that need to be solved, such as the confinement problem and the revocation problem. Researchers have been working on this and have proposed a number of solutions, most of them involving changing the structure of capabilities. There are also limitations on capabilities in that they only provide simple permission controls, and are not precise enough in controlling sequences of actions, or behaviours. This motivates us to develop better solutions that have finer behaviour control functionalities.

We will see a progressed development of access control approach that is extended from capabilities in the next chapter called *Vistas*, which show the direction of evolving towards the behaviour controlling.

Chapter 3

Vistas

The Capability concept for access control is quite abstract, as components in such systems are defined implicitly. It just describes a mechanism in principle: agents carry some sets of permissions and each set of permissions refers to a target. There are many gaps in this which have not been finalised and have been left for developers to adjust to get a specified capability-based system that best fits. These spaces include: how do the set of permissions and the reference to the target group to form a capability; how do references in capabilities link to targets; what are targets, etc. It is because of these parameters that the modifications of capabilities becomes feasible.

Taking the advantage of capabilities in dynamic control, we would like to keep extending this idea to make it more applicable in distributed environments. Since capabilities have been proposed as an access control approach, they are used for protecting objects from invalid operations. However, as more sophisticated applications are developed, these objects become more and more complex. An object may contain a set of data, many references to other objects, field variables that indicate the current state of the object, large data blocks, and so on. It can be far beyond a simple file that only needs permissions to read and write. Hence it is worth expanding the simple

idea of capabilities to catch such complexity of objects, and *Vistas* are such a development.

Vistas focus on the relationship between references in capabilities and components in objects, and are a step towards better behaviour modelling and control.

3.1 Vista Ideas

It has been mentioned that an object may have many field variables, each representing some particular information that may or may not be accessed by agents or users. Then to the access control view, the accesses to different field variables shall be treated individually. Thus we define a *visibility* to be a pair consisting of: a) an object reference, and b) one of its variables, or *names* (or a sequence of names, see the operation of *product* in the following section). The object reference can be held by several users, and having this visibility means a user can access to the variable or name that is referred.

If we take a relational view [73, 74] of visibilities, then clearly it means that a name in an object is mapped from one object reference by the 'relation' of visibilities, and a collection of visibilities becomes a relation from the set (domain) of object references held by users to the set (codomain) of names (or name chains) in targeted objects. We call this collection a *vista*. Vistas are first-class reified runtime objects that are used for access control. Users hold vistas to claim their rights to access those targeted objects. Since visibilities in vistas are not ordered, and the number of appearances of a visibility has no influence on the permissions granted by the vista, it can be seen that vistas are *sets* of visibilities. This set view of Vistas could be useful, as it means that we can apply properties and operations in set theory [36, 1] to our Vista concepts. This is particularly important when we come to the topic of Vista Combinator Operations.

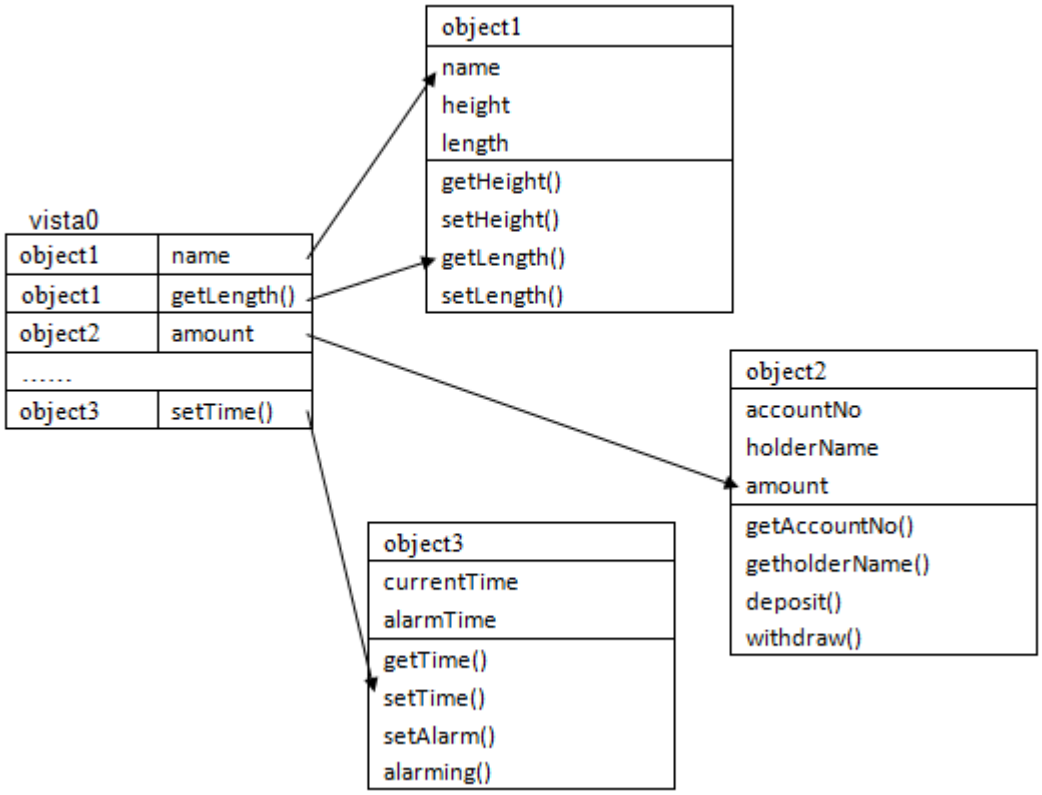


Figure 3.1: An Illustration of How Vistas Refer to Objects

It should be noted that a vista may contain visibilities to *multiple* objects, which means that with one vista the holder is able to operate on many objects if he wants. Another notable thing is that one vista does not need to contain visibilities for all names for a targeted object. This means the case is possible that the holder of a vista can only access part of its targeted object. These properties make vistas more precise when controlling accesses for those complex objects, compared with capabilities. For example, imagine there is an object having the profile of a student in a university, and there are methods that can view and change the content or status of different parts in this profile. The university has the vista that contains all visibilities to every part of this object of profile, which means the administration of the university can freely access, view and change the profile. The student himself can be given a vista which only has permissions to methods of 'apply for a course'. There can also be a vista that open to the public, and with it the holder can view some data of the student such as name, gender and the programme he is taking.

Vistas are collections, and so there is no difference between a user having one vista containing all visibilities he is holding, or having many vistas grouped by partitions of the set of all visibilities — the permissions for accesses are the same. However, it is still possible for a user to have multiple vistas at some point of time. This will be illustrated in the following sections.

3.2 Vista Combinator Operations

As defined, vistas are first-class objects, and they can be operated on in the same way as other objects, in object-oriented systems. Assuming there is a vista α and an action `read`, then we can use α.`read` to express a `read` action using the vista α. However, according to the property of vistas described in the previous section, a vista might contain references to multiple targeted

objects, and it is possible that more than one object is permitted to be read by this vista. This would not be a fault though, as it allows the possibility of executing one command on multiple targets. This introduces an interesting feature of forcing the users to execute actions to multiple targets at the same time. For instance, imagine there is a user u_1 writing an article, and for some reason he cannot finish it. Thus he would like to let another user u_2 to continue writing the article instead of him, but he still want to keep a private copy of the article such that he can read it when u_2 append new contents into the article. With the multiple targets of vistas we can realize this by: u_1 make a copy a_2 from the original article a_1 and save it privately, and giving a_1 and a vista v containing both visibilities of the `write` permission to a_1 and the `write` permission to a_2 to u_2. When u_2 performs a `write` action using v, it will simultaneously update both a_1 and a_2. Fig. 3.2 illutrates how this process happens. Note that we do not want u_2 to prevent a_2 from being written simultaneously, because this is exactly what u_1 wants to ensure that u_2 behaves correctly. To achieve this the system/language implementing this vista approach must be able to handle multiple selection.

There are a number of ways to obtain a vista. As vistas are extensions of capability ideas, the methods used to obtain capabilities are also inherented by vista systems [72].

- *Initiality.* Some vistas will be available to an entity (agent or user) on its creation or when joining the system. These vistas are likely to cover the access visibilities for the new entity itself (so that other entities can use these visibilities to access this new one).

- *Parenthood.* Vistas for a newly created object are available to the creator of this object. It is assumed that this vista contains visibilities to all names in the object.

- *Endowment.* The creator of new objects can pass any vistas he holds to these new objects. In object-oriented systems this corresponds to passing

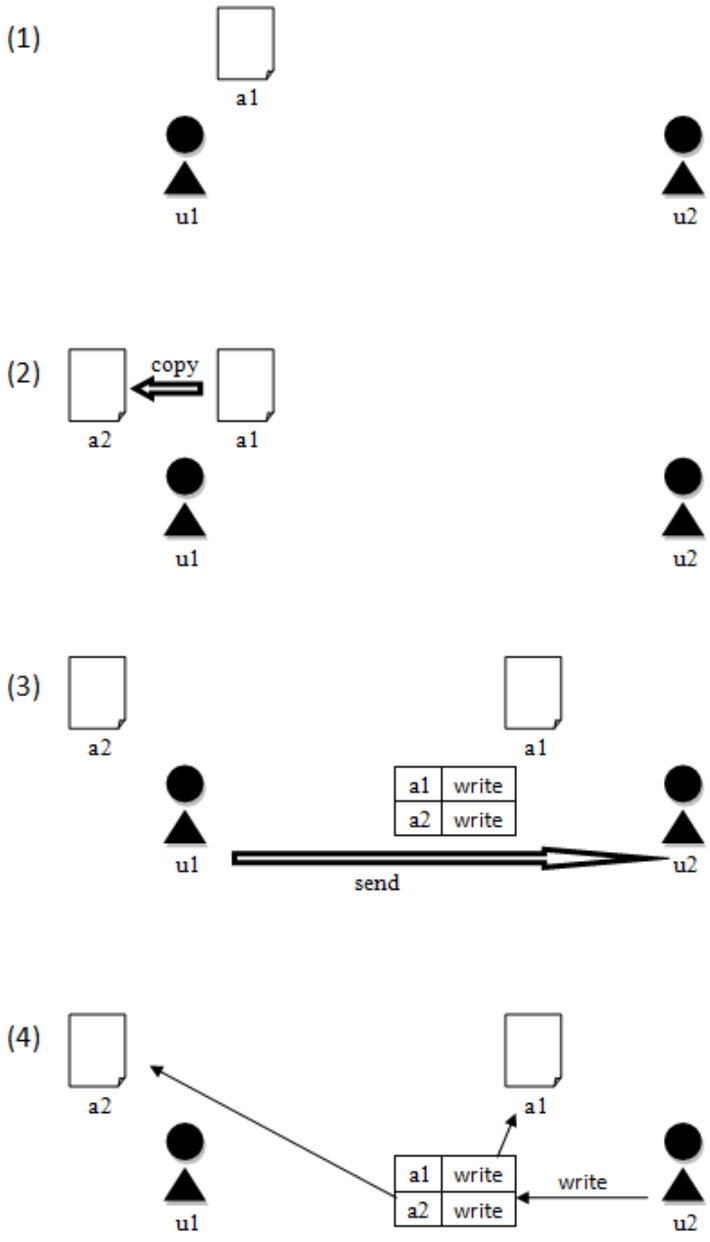


Figure 3.2: Process of Applying Multiple Targets Aspect

vistas as arguments to the constructor of the new object.

- *Introduction*. An entity can pass vistas to another by sending messages containing vistas, either by actively giving out or passively replying to requests from other entities. In non-distributed object-oriented systems this process corresponds to parameters and returned results for methods. In distributed systems this can be realized by physical conmmunication between two entities.

Other than these, there is a new path to obtain vistas:

- *Combination*. An entity can use vistas he is holding to produce new vistas, using some operations. Entities can use these operations to adjust permissions as required and send the new vista to other entities.

Before any operations can be introduced, there are two fundamental requirements that must be kept.

Requirement 1 Visibilities cannot be increased.

Requirement 2 Vistas cannot be forged.

The first requirement implies that however the vista operation is designed, the resulting vista must not contain visibilities that do not exist in the vistas the creator is holding. Not satisfying this requirement means entities can gain more access than already assigned to them. The second requirement ensures that entities cannot obtain vistas by ways other than the proposed ones. Again this protects systems from illegitimate accesses.

Table 3.1 shows a general view of the four basic vista operations that could be useful for entities to combine and create new vistas. And more detailed explanation is given below:

Sum is the very operation that makes a vista refer to multiple objects. Taking the set view of vistas, the result of a *sum* operation is a set of visibilities, which is the union of the two original vistas i.e. sets of visibilities. As for the requirements, the summed vista is the largest set of any sets that result from any binary vista operation, as it contains

Sum	$\alpha + \beta$	is a vista representing *all* the visibilities of α and β.
Difference	$\alpha - N$, $\alpha - \beta$	is a vista containing the visibilities of α *without* those, if any, referring to the names contained in the set N, or in the vista β. Since a vista can be regarded as a set of visibilities, this operation is naturally overloaded to allow a vista as its second argument.
Intersection	$\alpha \wedge \beta$, $\alpha \wedge N$	is the vista containing visibilities, if any, that are common to α *and* β. This is extended to work with a set of names N as in the case of *difference*.
Product	$\alpha \times \beta$	is a vista that represents ordered pairs of visibilities from α and β.

Table 3.1: Four Basic Vista Operations

all visibilities from the two original vistas. Vistas combined from other operations can only be a subset of the result of a *sum*.

Difference works like the difference operation in set theory. The result of $\alpha - \beta$ will have all visibilities of α without those in β. The vista operation of *difference* can use a set of names as the second operand, in which case the result will not contain any visibilities refereing to names in the given set.

Intersection between two vistas acts as the intersection of these two sets of visibilities. In the intersected vista there are only visibilities that appear in both of the original vistas. Again *intersection* can also have a set of names as its second operand, in which case the resulting vista will contain visibilities referring to names in this set only.

Product performs differently from the other three operations, and it gives a taste of behaviour modelling. A *product* between two vistas means 'the holder must perform actions in the two vistas consecutively', and

this consecutive execution shall not be intercepted by other actions. In the resulting vista of a *product* between α and β, there are no longer simple visibilities, but a collection of pairs/strings of visibilities. The first part of visibilities represented in these pairs is from α and the second part of visibilities is from β. These pairs/strings become tied actions that will be executed in order atomically. In an object oriented view, to use a product vista α, some command like α.<f,g >will be called. There are cases when the first action in this pair may not be executed properly, which could be caused either by the user not having the permission to perform this action, or errors or exceptions happening while the action is processing. In the former case it is suggested to deny the second action in the action pair as well, but in the latter case it may be acceptable to let the second action being normally performed if it will not be affected by the error of the first action.

In vistas, although one can keep using the *product* produce strings/chains of actions to have a 'taste' of behaviours, they are still very simple. These are only sequences, and as they are performed atomically, once the tied actions started, it cannot be interrupted and turn to other actions. The aspect of behaviour should include branches that would let the actor to choose actions. Controlling behaviours will be examined later in the context of *Treaties* (Chapter 4).

3.3 Use-Cases

Due to the difference between vistas and capabilities, there are some new ways of applying vistas in practice. In this section some examples of novel use-cases which use vistas are introduced.

Instant Messaging

Consider a simple chat system with users distributed across the network. Users are allowed to communicate peer-to-peer, and they are also able to create chat rooms divided by topics, and these rooms allow multiple users to 'broadcast' their messages to all other users in the same room at the same time. In a capability-like system, to send a message to another user you will need to obtain `write` access to him. To send one message to multiple targets, copies of the message will be sent one by one.[1] In a vista-based system, we can realize this function by using the operation of *sum* (shown in Fig. 3.3).

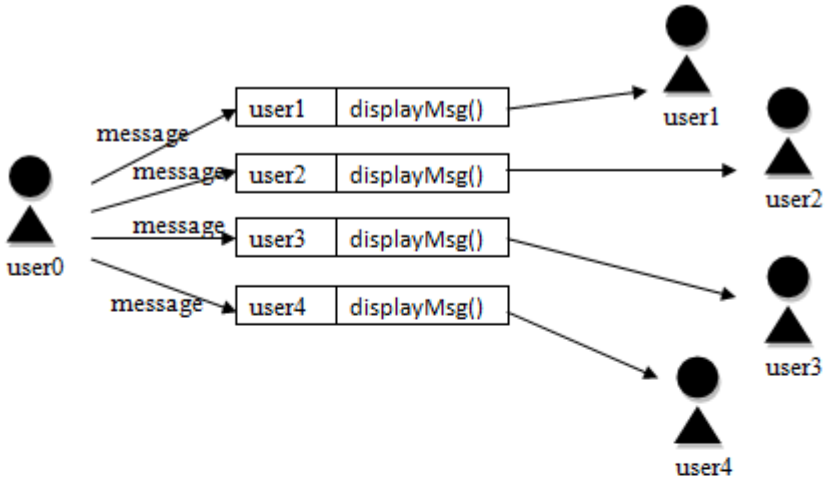
To initialize, an entity sends the vista including his `write` visibility to the manager of the chat room (either the creator of the room or the server), and the manager sends this vista to members of the room, then these members return acknowledge with their vistas for `write`. Each member now uses the *sum* operation to combine their newly obtained vista with their room vista *topic*. To broadcast a message, members can execute *topic*.`write`(message), and the message will be spread to all members in the chat room. At the implementation level, how the system will actually do this action depends on the facilities of the language. If the language supports concurrency or multicast functionality, then it can broadcast the message to all members.

To terminate, the member can do a *topic*.`leave` action which informs other members of his leaving, and then the other members can use a *difference* operation to remove the exiting member's `write` visibility.

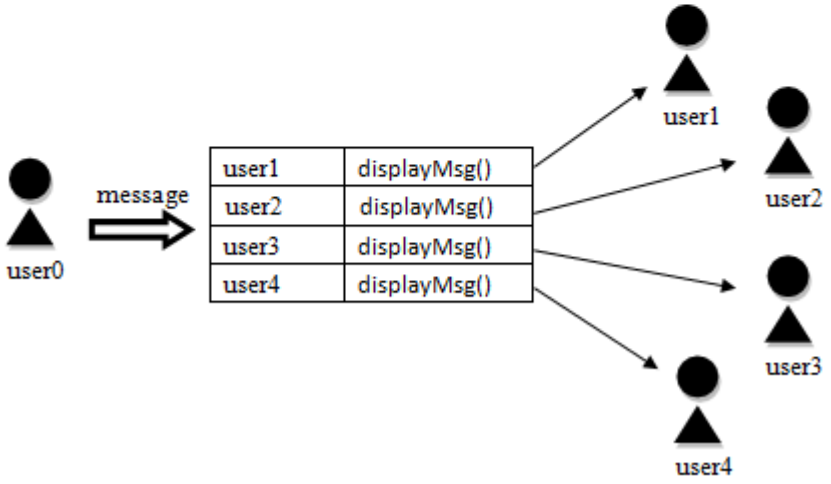
Security Proxy

Imagine a new student, Alice, is enrolled by a university. She will need to set up an account so that her tuition fees can be saved and collected; the fees are provided by her parents. But rather than a simple account, the parents would like to add some control on it so that the account is guaranteed to be

[1] Actually there is another way of dealing with this, and it is to create a new object that represents the chat room, and let each member in the room have both `read` and `write` permission to this room object, but this requires extra resources supplied by the system.



a) using uncombined vistas



b) using combined vista

Figure 3.3: Difference Between Using a Set of Vistas and a Summed Vista

used for tuition fees only. The specified requirements are: Alice can view the balance of the account and deposit money in it (in case she may do some part-time jobs), and the University can withdraw money from it and also deposit into it (rewards or new sponsorship), while the deposit permission is open to the public (for donations). The parents have full permissions including `save, withdraw` and (view) `balance`.

Assume the full vista is called *account*. A simple way of achieving the requirements would be to create the vista ($account \wedge \{\texttt{save,balance}\}$). However, due to the properties of vistas, these visibilities could be propagated and other entities may access to the balance of the account. Thus as well as the *account* object, the bank provides an extra security object called *guard*. This object can examine the identity information of entities by a `login` action with the entities' details. Now an enhanced version of Alice's vista can be produced by using product operation:

$$alice = (guard.\texttt{login}(aliceDetails) \times account.\texttt{balance}) + account.\texttt{save}$$

Using this vista, anyone who wants to view the balance will have to provide Alice's details, so a stronger security is provided. And because the `save` permission is not restricted by *guard*, Alice can distribute the vista to others for any donation and presents.

Similarly, the financial office of the university will be offered a vista like:

$$fees = guard.\texttt{login}(uniDetails) \times (account \wedge \{\texttt{withdraw,save}\})$$

which means the university will need to provide relevant information before access to the tuition account, but viewing balance is not available to the university. Vistas held by different roles in this case are shown in Fig. 3.4.

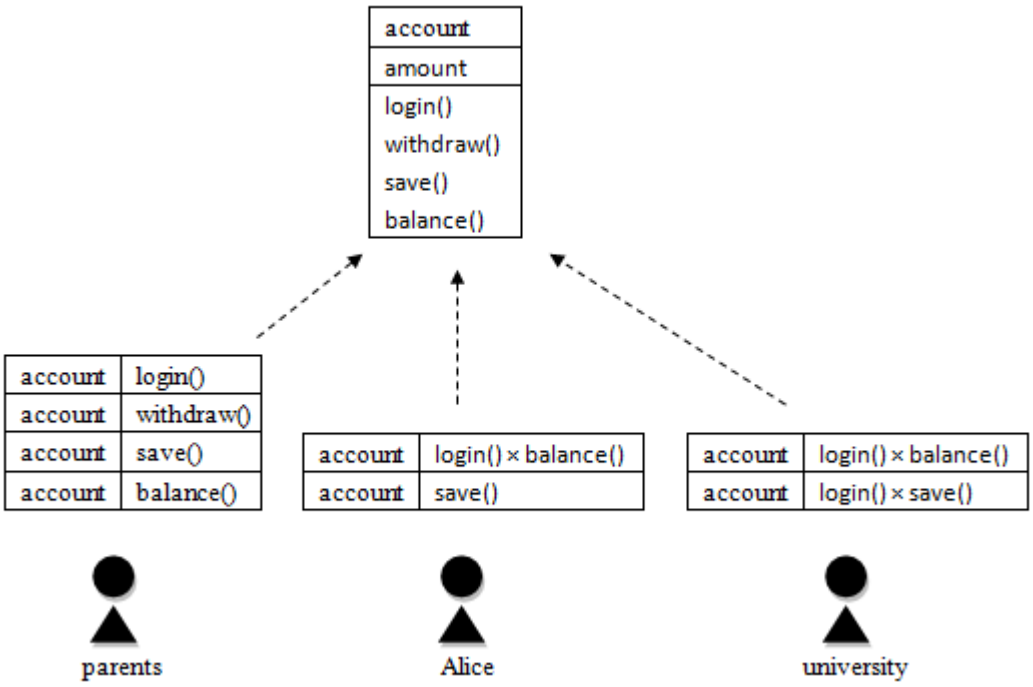


Figure 3.4: Users Holding Different Vistas Access to a Same Object

3.4 Summary

In this chapter the idea of Vistas has been introduced which has been further described in the previous published paper [72]. Vistas are access control components that extend the capability approach a further step. In the Vista approach access control is made finer by specifying accesses to the field values and methods (in object-oriented sense) in each target object, and these explicit accesses are represented by visibilities. Each visibility is a pair consisting of a reference to the target object and the identifier of one of the field values or methods in this object. The holder of a visibility can view and modify the field value or call the method in the referred object. Vistas are sets of any visibilities one holds, hence a vista can grant accesses to multiple objects.

Vistas can be constructed using vista operations, by regrouping visibilities inside. With the constructed vistas, users can define different access patterns. This enhances the services such that the holder of the vista can access different sets of target objects, or he can pass the vista to other users with the access pattern specified by him.

The *product* operation in vistas realizes sequences of accesses, and this shows a particularly interesting development: controlling the accesses in a behavioural way. In the real world many behaviours exist, made by humans, machines and many other active objects. These behaviours usually include a number of actions happening in particular orders and patterns. For example, the behaviour of someone going home after work: he will unlock the door first, then walk into the house, lock the door from inside, then either take off his coat then shoes, or take off his shoes first then coat. The behaviour stops here, followed by other behaviours such as cooking and taking a shower. It would be nice if we could take control over accesses step by step so that we can force the actions taken by entities to become meaningful behaviours.

Vistas formed by *product* operations can make sequences of actions that show an aspect of behaviours, but they are still not able to simulate the 'choices', or branches, of actions. They cannot allow and then stop accesses to variables in objects, as vistas also grant unlimited number of accesses like capabilities. It is necessary to look for a new approach of making behaviour control to make development progress.

The following chapters will introduce an extension to vistas which allow a more complete control over behaviours.

Chapter 4

Treaty Concepts

Treaties extend the capability and vista ideas to a new level. Capabilities and vistas only deal with rights. These rights in capabilities are independent, unlimited and unordered. For example, assuming there is a capability containing both `read` and `write` rights, the two access rights are not dependent on each other, and the holder of the capability can do `read` or `write` an unlimited number of times, and in any order. This is a straight-forward mechanism that is suitable for simple systems. However, as services become more sophisticated, the simple mechanism is no longer satisfactory for complex system requirements. There are many cases that require the access actions to be in certain orders, or a limited number of times. We call these sequences of actions *behaviours*. Clearly capabilities are unable to specify behaviours, except in the most trivial sense. Vistas make one step towards behavioural control, by the *product* operation, but they are still too simple: there is only sequencing of actions. Hence we need to find new ways to fit the behavioural specifications, such as sequences, choices and limited repetitions of actions. We will introduce the solution in this chapter.

4.1 What Treaties Are

Recall our motivation of progressing from action control to behaviour control: the simple capabilities can only do the former control. In capabilities, there are only sets of action types that allow the holder of the capability to perform actions of types defined in the set to the referred object. The valid actions granted by the capability are independent, unlimited and unordered, which means, performing an action will not be affected by the performing of other actions, these actions in the action set can be performed as many times as users like, and the performance of these actions has no particular sequence. These properties cannot completely define behaviours, which are characterised by sequences, branches and terminations. Capabilities have such a simple structure that they are not capable of controlling behaviours, therefore we intend to upgrade them to solve this problem, and we call our solution *Treaties*.

Treaties evolved from the idea of capabilities and vistas, and so they can also be used as access control components. This inheritance implies that treaties adopt many properties from capabilities:

- They are structures that define the rights to operating on resources.
- They are not centrally controlled, but are held by user-level agents.
- In treaty systems a user must present an appropriate treaty to the kernel (middleware) in order to perform actions on the resource that he is accessing.
- Only kernels can create the initial treaty to a certain resource - users cannot forge any treaties.
- Treaties shall also inherit the aspect of dynamic access control of capabilities, as they can be propagated throughout systems.

The aspect of dynamic access control is characterised by the fact that: a user who possesses a treaty can pass it to another user, and the receiver of the treaty then gains the same rights to access the object it refers to. In this way the permissions can be distributed without notifying the kernels, as in the case of capabilities. This means that treaties are suitable to be deployed in distributed environments as well.

The key function that treaties provide is 'behaviour control'. This differs from the 'rights-only' mode of capabilities in the sense that a treaty can allow or deny different actions at different times or stages, depending on the current state of the treaty or current stage of the process/behaviour (capabilities are stateless entities, and so cannot control sequences of actions. See Fig. 4.1). With treaties, sequences of allowed actions are specified and enforced, in order to form behaviours.

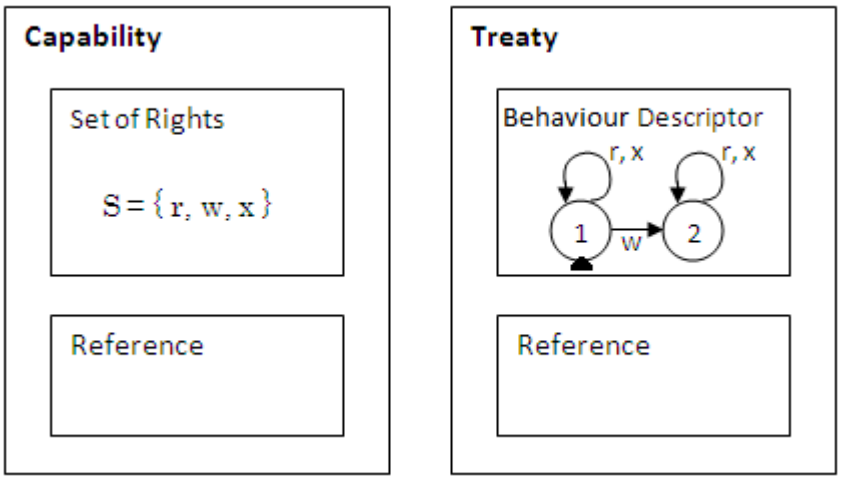


Figure 4.1: Structure of Capabilities and Treaties

As an example of a behaviour as a sequence of actions from everyday-life, in the morning we sign-in to our email services, read new emails, reply to senders and finally shut-down mail boxes. This is exactly a simple but

typical behaviour. At the beginning, one cannot read or compose mails but only log in. After logging in one can read and compose an unlimited number of times, until logging out, after which only logging in is allowed. Treaties aim to capture these attributes of behaviours, such as sequencing, branching and terminating.

States are a vital part of the Treaty concept. A behaviour can have multiple states, and these states are connected by actions/accesses. Every time a user performs a valid action over a resource, a transition between the current state and the target state is made (keep in mind that these are not necessarily two distinct states). The *current state* is a pointer or indicator to one of states in the behaviour contained in a treaty. This is similar to finite-state machines [11, 14, 15], and in fact an FSM is a good option for representing behaviours [76].

Abstractly, treaties are access control components that provide specifications of behaviours that can be followed for a number of resources, together with the current state of the evolution of the behaviours. As fundamental requirements, treaties shall:

- refer to resources.
- provide information to a kernel to enable it to allow or deny an action request.
- not be strictly bound to agents, i.e. kernels do not need to check their holders' identities in order to grant or deny actions.
- be able to 'evolve' behaviours according to their specifications, i.e. maintain and update the current state.

Therefore each treaty can be seen as consisting of three components: the reference pointing to the resource (or resources), the behaviour descriptor

modelling the behaviours permitted by the treaty, and the current state of the behaviour in this treaty. From this, we can see that that capabilities are a special type of treaty, in which there is only a single behavioural state, and all allowed actions are reflexive transitions on this state.

In a treaty-based system, the usual process of using treaties can be as follows:

1. a user U requires the kernel to create a resource R;

2. the kernel creates the resource R and returns a treaty T to user U, where T contains all possible action types regarding to the resource R with unlimited number of accessing repetitions;

3. user U can refine the behaviour model represented by the behaviour descriptor in T so that specific treaties $T_1...T_n$ are constructed;

4. U can pass these treaties (both newly refined and that have been executed) to other users $U_1...U_n$ to distribute the right to access R;

5. U_m who received T_m can show this treaty to the kernel to access R, if this action is validated by T_m.

4.1.1 Terms and Definitions in Treaty Systems

Before any formal definitions can be given, we will need to understand the following terms that will be or have been used, to avoid any confusion.

Kernel - Also referred to as server. This is the unit that processes requests from users. In traditional approaches kernels act as the role of administration, and in the capability and treaty approach kernels still do the important job of checking validity and addressing objects.

Subject - The holder of treaties, including users and processes, depending on different types of systems. Subjects can exchange treaties among themselves, and will show treaties to kernels when they need to access objects.

Object - Can also be called resources or files. Objects are the part that is protected by treaties. Note that in distributed systems, objects are usually not located centrally, but on devices of different subjects. And in some cases, subjects themselves can be objects, where 'send message' is the possible accessing action, and need to be protected by a treaty. Subjects and Objects are both *entities* in systems.

Action - This is the term used to call a single access. For example, in a file system, `read` and `write` can be two types of actions. The following — to read from a file, then append (write) some information to the file, and finally read the file again — is regarded as three actions.

Behaviour - We define a behaviour as a sequence of actions. Each behaviour contains an ordered list of actions. The mentioned '`read-write-read`' example is a behaviour. The number of actions in a behaviour can be any non-negative integer, including zero (empty behaviour) and infinity (might be a loop of actions).

Behaviour Model - Sometimes we need to describe behaviours instead of listing them one by one. A behaviour model defines a group of behaviours using sequencing, branching and terminating. One behaviour model contains multiple behaviours, and we can say these behaviours are modelled by the behaviour model.

Behaviour Descriptor - This term is used with regards to implementations, and behaviour descriptors are a component of treaties. A behaviour descriptor represents one behaviour model (or more than one,

depending on the choice of referencing style which will be discussed later).

Current State - This is used to indicate the current point in a behaviour model. Each time a valid action is performed, it will cause a move or transition from the current state (may move back to the same state). Again the current states are a component of treaties.

Example of the structure of a treaty system is shown in Fig. 4.2. In this simple case there are three users: *user1, user2, user3*, which are in the classification of subjects. There are also three objects of *resource1, resource2, resource3*, and users possess treaties that contain permissions to some of the objects. Two kernels exist in this system and each of them directly communicates with some entities (either subject or objects). If *user3* want to access *resource2* (assuming he has the correct permission), he will first report to *kernel2* and hand in the *treaty:R2*, and *kernel2* will communicate with *kernel1*. *Kernel1* then checks the premissions in *treaty:R2*, and grant the action requested by *user3*.

Treaties and their representing behaviours can be also seen as regular languages [20, 75]. All action types for the resource can be seen as the alphabet of the language, and the behaviours are strings in the language (we will use $[\![\alpha\ ]\!]$to represent the *behaviour set* given by the treaty α). Taking an example, suppose there is a treaty α which allows its holder to do only three actions '`read-write-read`' to the resource consecutively, then the valid behaviour set $[\![\alpha\ ]\!]$granted by α, i.e. the valid string set in this language is: {ϵ, `read, read.write, read.write.read`}. ϵ is the empty behaviour that means doing nothing, and it will exist in any behaviour set defined by any behaviour model.

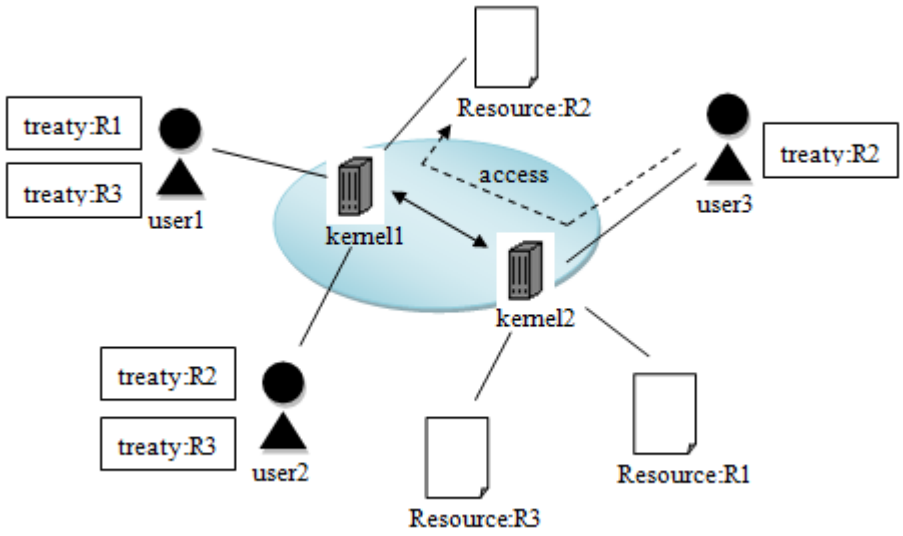


Figure 4.2: An example of the Structure of a Treaty-based System

We define a behaviour model M on an object as

$$M = (Q, \Sigma, t, q)$$

where Q stands for the set of all states in this model; Σ is the set of all action types for the type of the targeted object; t represents the set of all action transitions between pairs of states in (a subset of) Q; q the current state of this behaviour model. If we take a look at the definition of finite-state machines [15] we find that in our definition of behaviour model, there is no set of accepting states F. This is because our behaviour model is used for access control, and in access control systems the actions are dealt with individually. Every time a subject asks the kernel for a single action, the kernel will return the validation result of it, and no future actions concerned, which means it does not consider whether there will be a next action and what it is — the validation process is only based on the current state of

the behaviour. The subject can stop performing actions at any state in the behaviour model and this is perfectly valid. Thus, any state in behaviour models could be seen as an accepting state, i.e. $F = Q$.

4.2 Operations and the Rule of Treaties

As has already been mentioned, when a new resource object is created, the kernel provides the creator of this object a 'complete' treaty, which contains all possible action types with unlimited repetitions, i.e. a capability. To transform the complete treaty into a treaty that specifies more refined behaviours, we need a number of combinator operations. However, there is a fundamental rule for these treaty combinator operations. To keep the integrity of treaty systems, we must ensure the action permissions held by a user would never exceed the scope of permissions that have been assigned to him.

Rule. *An agent cannot increase the totality of behaviours defined by the treaties it holds.*

This is the core requirement of treaty operations, all operations and their results must obey this rule, in any representation and implementation of treaty systems.

4.2.1 The Five Basic Operations

Corresponding to attributes of behaviours of sequencing, branching and terminating, there are three basic operations: *concatenate*, *join* and *restrict*. In addition, two operations of *intersect* and *difference* are also introduced below:

Restrict is a unary operation that can decrease the number of times a certain type of action can be executed to an assigned number. The

actual way of designing this operation can be varied depending on different representations of treaty systems. For a treaty α, *restrict* will be shown: $[\alpha]^{\mathtt{a}}_n$, where $\mathtt{a}$ is the type of action to be restricted, and n is the maximum number of times that $\mathtt{a}$ can be performed.

Difference $(-)$ operates between two treaties, or a treaty and action type. For two treaties α and β, $\alpha - \beta$ contains behaviours in α that are not in β. $\alpha -\ \mathtt{a}$ removes all $\mathtt{a}$ actions from α.

Concatenate $(\cdot)$ is a binary operation which connects behaviours in two treaties in order. $\alpha \cdot \beta$ is a treaty that allows all behaviours consisting of a behaviour allowed by α followed by a behaviour allowed by β.

Intersect $(\sqcap)$ is a binary operation which results in a treaty whose behaviour is the intersection of the behaviours of the two original treaties. Treaty $\alpha \sqcap \beta$ specifies behaviours that are allowed by both α and β.

Join $(\sqcup)$ is also a binary operation which constructs a treaty from two original treaties, so that the holder can choose to perform a behaviour from either of the original treaties. Treaty $\alpha \sqcup \beta$ allows all behaviours in α and β.

To help understanding, Fig. 4.3 demonstrates how the four binary operations perform. Assuming there are two behaviour models in two treaties, in each figure the left circle denotes the starting state of the result, and the right circle denotes the finishing state. The path between the two states is the valid behaviour model that results from the two original ones. In the case of *intersect* operation the path is the region with slashed lines, and for the *difference* operation the path is the shadowed region.

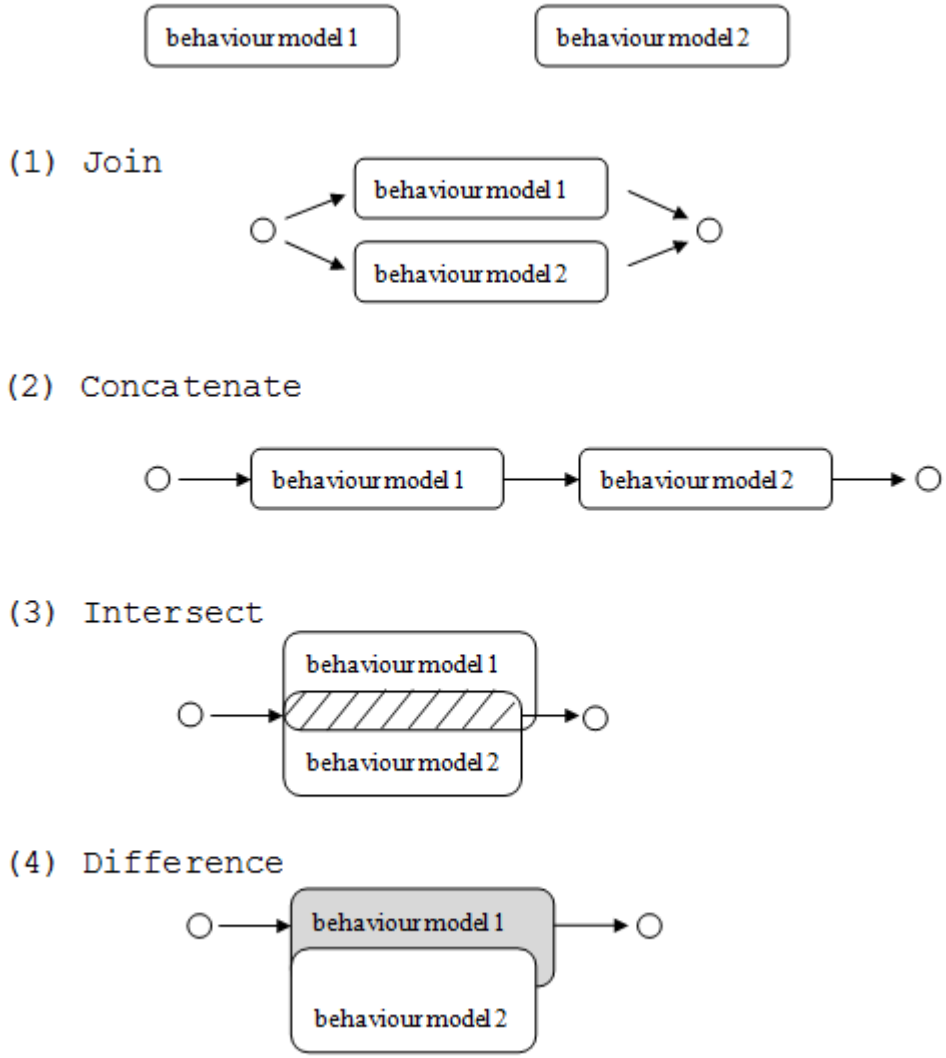


Figure 4.3: Graphical Illustration of Binary Treaty Operations

4.2.2 Using Behaviour Sets to Prove Correctness of Operations

In order to make sure these operations do not break the rule, it is necessary to know the upper bound of the behaviour set that the holder of the original treaties can do. Assume one has two independent treaties α and β. Let treaty α contain only two consecutive actions `read;write`, and treaty β contain two consecutive actions `execute;rename`. As these actions in two treaties are independent, the sequences of performing actions can be in any order, and all sequences are given by:

```
read;write;execute;rename
read;execute;write;rename
read;execute;rename;write
execute;read;write;rename
execute;read;rename;write
execute;rename;read;write
```

From this observation we find that the upper bound of the behaviour set by two independent treaties is a *shuffle* [26, 37, 6] of the two original behaviour sets.[1] Generally speaking, the *shuffle* operation produces a set of expressions from two expressions such that in the resulting expression every letter will follow its sequence in the original expression, but the order of letters from two different expressions are completely independent.

There is, however, another point that needs to be noticed. The behavioural accesses allow users to perform actions in some order as the behaviour model defines, but it does not mean that to complete a behaviour one must perform *all* actions until the model reaches a terminating state (in fact many models do not have such states). In treaty-based access control systems, all users need to do is to ask the kernel whether they can do

[1] Here it is assumed each action is separated, and there is no notion of atomicity in treaties in the abstract treaty concept.

an action before it is performed, and users can stop performing actions at any point in behaviour models, regardless of the following actions. In other words, every state in behaviour models is an 'accepting' state. From this, we have found a property of behaviour sets defined by behaviour models: these sets are *prefix closed* [53], which means that if a sequence of actions (behaviour) is in a behaviour set, then its prefix is also in this set:

---

let an action $\mathtt{a} \in \Sigma$

let a behaviour $B \in \Sigma^*$

for a behaviour set S, if $B.\mathtt{a} \in S$, then $B \in S$

Let us define a *shuffle* operation between two treaties that will produce a prefix-closed shuffled behaviour set from the two treaties, using the symbol $\odot$. Let α and β be the two original treaties, $u(\alpha)$ be any unary operation over α and $\circledcirc$ be any binary treaty combinator operator, we can rewrite the fundamental rule formally as the following:

$[\![u(\alpha)\ ]\!] \subseteq [\![\alpha\ ]\!]$

$[\![\alpha \circledcirc \beta\ ]\!] \subseteq [\![\alpha \odot \beta\ ]\!]$

Consequently, it must be shown that the treaty operations conform to this rule.[2] Note that *shuffle* is a commutative operator, i.e. $\alpha \odot \beta = \beta \odot \alpha$.

Restrict From the definition of restrict it is clear that restrict operation will only decrease behaviours from the original treaty, thus it certainly obeys the rule.

$[\![[\alpha]_n^a\ ]\!] \subseteq [\![\alpha\ ]\!]$

Concatenate Because the *shuffle* operation allows actions from different sets of behaviours to be performed in arbitrary order, it is perfectly valid to let actions from the first treaty to be performed before the

[2]For now, let us assume treaties that are applied by operations should refer to the same target object.

second treaty, which is exactly what the *concatenate* operation does. Hence:

$[\![\alpha \cdot \beta]\!] \subseteq [\![\alpha \odot \beta]\!]$

Join* and *Intersect Since the behaviour sets are prefix closed, we have

$[\![\alpha]\!] \subseteq [\![\alpha \cdot \beta]\!]$,

as behaviours in α are prefixes of behaviours in $\alpha \cdot \beta$. Since '$\subseteq$' is transitive, we have

$[\![\alpha]\!] \subseteq [\![\alpha \odot \beta]\!]$,

and similarly

$[\![\beta]\!] \subseteq [\![\alpha \odot \beta]\!]$.

By the definition of the join and intersect operations

$[\![\alpha \sqcup \beta]\!] = [\![\alpha]\!] \cup [\![\beta]\!]$,

$[\![\alpha \sqcap \beta]\!] = [\![\alpha]\!] \cap [\![\beta]\!]$.

Due to the set theory, $[\![\alpha]\!] \cup [\![\beta]\!] \subseteq [\![\alpha \odot \beta]\!]$, $[\![\alpha]\!] \cap [\![\beta]\!] \subseteq [\![\alpha \odot \beta]\!]$. Thus

$[\![\alpha \sqcup \beta]\!] \subseteq [\![\alpha \odot \beta]\!]$,

$[\![\alpha \sqcap \beta]\!] \subseteq [\![\alpha \odot \beta]\!]$.

Difference For the difference operation, taking a naive view, we might think that $[\![\alpha - \beta]\!] = [\![\alpha]\!] \setminus [\![\beta]\!]$(set minus). However, assuming there are two treaties γ and δ referring to a same target, γ allows a `read` action then a `write` while δ allows only a `read`.

$[\![\gamma]\!] = \{ \epsilon$, `read, read.write` $\}$

$[\![\delta]\!] = \{ \epsilon$, `read` $\}$

Thus

$[\![\gamma]\!] \setminus [\![\delta]\!] = \{ \epsilon, \texttt{read.write} \}$

This set is *not* a prefix closed set, so it does not conform the property of behaviour sets. This illustrates that the operation of difference cannot simply be defined to be a set difference between two treaties. All behaviours that have prefixes in the behaviour set defined by the second operand treaty must also be taken from the behaviour set from the first operand treaty. This is also naturally reasonable, as if one is banned from doing something, one will certainly not be allowed to do anything that *must* be preceded by the banned behaviour.

Thus, the *difference* operation $\alpha - \beta$ is defined by removing all non-empty behaviours in $[\![\beta]\!]$and the behaviours that have these β-behaviours as prefixes from $[\![\alpha]\!]$:

$[\![\alpha - \beta]\!] = [\![\alpha]\!] \setminus \{ x \mid x = b.\Sigma^*$, where $b \in [\![\beta]\!]$and $b \neq \epsilon\}$

And since it is a set minus from α, and $[\![\alpha]\!] \subseteq [\![\alpha \odot \beta]\!]$, we can get

$[\![\alpha - \beta]\!] \subseteq [\![\alpha \odot \beta]\!]$

Therefore we have shown all the five basic operations obeys the rule of treaties, and hence they are safe to be used for behaviour model refinement.

We provide the formal definition for the five operations in Table 4.1 using the concept of behaviour sets. $\texttt{a}@b$ denotes the number of times that action `a` appears in behaviour b.

4.2.3 Laws of Treaty Operations

Basic treaty operations have been conceptually defined, and by their definitions we can find a number of algebraic laws that are followed by these operations. Assume there are three treaties α, β and γ, and we define an empty treaty which allows its holder to do nothing as ϕ, then

join	$\alpha \sqcup \beta$	$[\![\alpha \sqcup \beta]\!] = [\![\alpha]\!] \cup [\![\beta]\!]$
intersect	$\alpha \sqcap \beta$	$[\![\alpha \sqcap \beta]\!] = [\![\alpha]\!] \cap [\![\beta]\!]$
difference	$\alpha - \beta$	$[\![\alpha - \beta]\!] = [\![\alpha]\!] \setminus \{ x \mid x = b.\Sigma^*, b \in [\![\beta]\!], b \neq \epsilon\}$
concatenate	$\alpha \cdot \beta$	$[\![\alpha \cdot \beta]\!] = \{ x \mid x = b_1.b_2, b_1 \in [\![\alpha]\!], b_2 \in [\![\beta]\!] \}$
restrict	$[\alpha]^{\mathtt{a}}_n$	$[\![[\alpha]^{\mathtt{a}}_n]\!] = \{ x \mid x \in [\![\alpha]\!], \mathtt{a}@x \leq n \}$

Table 4.1: Formal Definitions for Treaty Operations

$$\alpha \sqcup \beta = \beta \sqcup \alpha \quad (4.1) \qquad \phi - \alpha = \phi \quad (4.8)$$

$$\alpha \sqcap \beta = \beta \sqcap \alpha \quad (4.2) \qquad \phi \cdot \alpha = \alpha \quad (4.9)$$

$$(\alpha \sqcup \beta) \sqcup \gamma = \alpha \sqcup (\beta \sqcup \gamma) \quad (4.3) \qquad \alpha \cdot \phi = \alpha \quad (4.10)$$

$$(\alpha \sqcap \beta) \sqcap \gamma = \alpha \sqcap (\beta \sqcap \gamma) \quad (4.4) \qquad [\phi]^{\mathtt{a}}_n = \phi \quad (4.11)$$

$$\alpha \sqcup \phi = \alpha \quad (4.5) \qquad \alpha - \alpha = \phi \quad (4.12)$$

$$\alpha \sqcap \phi = \phi \quad (4.6) \qquad \alpha \sqcup \alpha = \alpha \quad (4.13)$$

$$\alpha - \phi = \alpha \quad (4.7) \qquad \alpha \sqcap \alpha = \alpha \quad (4.14)$$

These laws can be used to examine the correctness of different representations and implementations of treaty systems. We can prove these using the formal definitions of treaty operations. *Join* and *intersect* are defined directly using set theory, hence they inherit the properties of set operations, thus:

(4.1) $[\![\alpha \sqcup \beta]\!] = [\![\alpha]\!] \cup [\![\beta]\!] = [\![\beta]\!] \cup [\![\alpha]\!] = [\![\beta \sqcup \alpha]\!]$

(4.2) $[\![\alpha \sqcap \beta]\!] = [\![\alpha]\!] \cap [\![\beta]\!] = [\![\beta]\!] \cap [\![\alpha]\!] = [\![\beta \sqcap \alpha]\!]$

(4.3) $[\![(\alpha \sqcup \beta) \sqcup \gamma]\!] = ([\![\alpha]\!] \cup [\![\beta]\!]) \cup [\![\gamma]\!] = [\![\alpha]\!] \cup ([\![\beta]\!] \cup [\![\gamma]\!])$
$= [\![\alpha \sqcup (\beta \sqcup \gamma)]\!]$

(4.4) $[\![(\alpha \sqcap \beta) \sqcap \gamma]\!] = ([\![\alpha]\!] \cap [\![\beta]\!]) \cap [\![\gamma]\!] = [\![\alpha]\!] \cap ([\![\beta]\!] \cap [\![\gamma]\!])$
$= [\![\alpha \sqcap (\beta \sqcap \gamma)]\!]$

By definition of ϕ we also have $[\![\phi\,]\!] = \{\epsilon\}$, and ϵ exists in every behaviour set, i.e. $[\![\phi\,]\!] \subseteq [\![\alpha\,]\!]$, thus:

$$[\![\alpha \sqcup \phi\,]\!] = [\![\alpha\,]\!] \cup [\![\phi\,]\!] = [\![\alpha\,]\!] \tag{4.5}$$

$$[\![\alpha \sqcap \phi\,]\!] = [\![\alpha\,]\!] \cap [\![\phi\,]\!] = [\![\phi\,]\!] \tag{4.6}$$

$$[\![\alpha - \phi\,]\!] = [\![\alpha\,]\!] \setminus \{\, x \mid x \in [\phi], x \neq \epsilon\} = [\![\alpha\,]\!] \setminus \emptyset = [\![\alpha\,]\!] \tag{4.7}$$

$$[\![\phi - \alpha\,]\!] = [\![\phi\,]\!] \setminus \{\, x \mid x = b.\Sigma^*,\, b \in [\![\alpha\,]\!],\, b \neq \epsilon\} = [\![\phi\,]\!] \tag{4.8}$$

$$[\![\phi \cdot \alpha\,]\!] = \{\, x \mid x = b_1.b_2,\, b_1 \in [\![\phi\,]\!],\, b_2 \in [\![\alpha\,]\!]\,\} \tag{4.9}$$

$$= \{\, x \mid x = \epsilon.b_2,\, b_2 \in [\![\alpha\,]\!]\,\} = [\![\alpha\,]\!]$$

$$[\![\alpha \cdot \phi\,]\!] = \{\, x \mid x = b_1.b_2,\, b_1 \in [\![\alpha\,]\!],\, b_2 \in [\![\phi\,]\!]\,\} \tag{4.10}$$

$$= \{\, x \mid x = b_1.\epsilon,\, b_1 \in [\![\alpha\,]\!]\,\} = [\![\alpha\,]\!]$$

$$[\![[\phi]^{\mathtt{a}}_n\,]\!] = \{\, x \mid x \in [\![\phi\,]\!] \text{ and } \mathtt{a}@x \leq n\,\} = \{\,\epsilon\,\} = [\![\phi\,]\!] \tag{4.11}$$

And for the operations where both their operands are the same:

$$[\![\alpha - \alpha\,]\!] = [\![\alpha\,]\!] \setminus \{\, x \mid x = b.\Sigma^*,\, b \in [\![\alpha\,]\!],\, x \neq \epsilon\} \tag{4.12}$$

$$= [\![\alpha\,]\!] \setminus ([\![\alpha\,]\!] \setminus \{\,\epsilon\,\}) = \{\,\epsilon\,\} = [\![\phi\,]\!]$$

$$[\![\alpha \sqcup \alpha\,]\!] = [\![\alpha\,]\!] \cup [\![\alpha\,]\!] = [\![\alpha\,]\!] \tag{4.13}$$

$$[\![\alpha \sqcap \alpha\,]\!] = [\![\alpha\,]\!] \cap [\![\alpha\,]\!] = [\![\alpha\,]\!] \tag{4.14}$$

Hence all the laws are shown to be correct.

From the laws we may find that it is not beneficial to make a binary treaty operation over a treaty to itself ($\alpha \sqcup \alpha$ and so on), except for the *concatenate* operation that is not specified in these laws. However, to concatenate a treaty with itself may cause more serious problems, as it will be explained in Section 4.3. Hence we can make the decision that one should never use binary treaty operations between treaties and themselves, as they only produce: treaties that have no difference from the operands of the operations, an empty treaty that does nothing, or treaties that break the assurance of behaviour control schemes.

4.2.4 A Use-Case Example

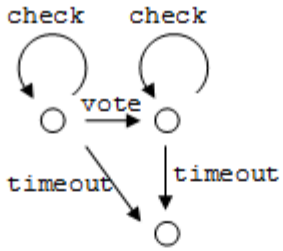
Consider an on-line voting system, where each voter can vote exactly once, and can check the current result at any time. But in contrast to ordinary voting, the organizer of this system would like to make vote actions available only up to the deadline set by himself. If we think in terms of our behaviour models, there will be three states and three types of actions in the behaviour model for each voting treaty, as shown in Fig. 4.4 part a). Note that `vote` and `check` are to be performed by voters, but the `timeout` action, which will stop the system from accepting voting, is triggered by the timer in the voting system. Thus this behaviour model needs to be separated into two parts held by users and the timer respectively, as shown in Fig. 4.4 part b).

To construct this, the organizer of the voting can first obtain a complete treaty α containing two types of action `vote` and `check` with unlimited repetitions, and get the ordinary voting treaty by performing the *restrict* operation $[\alpha]_1^{\texttt{vote}}$. Then there is a treaty with a single action `timeout` created by restricting another complete treaty β containing unlimited `timeout` repetitions to $[\beta]_1^{\texttt{timeout}}$. These two processes are shown in Fig. 4.4 part c) and d). The overall behaviour model of the voting, which is shown in Fig. 4.4 part a), is produced by performing a *concatenate* operation to the two restricted treaties. In general, the operations of constructing the overall voting behaviour model can be represented as:

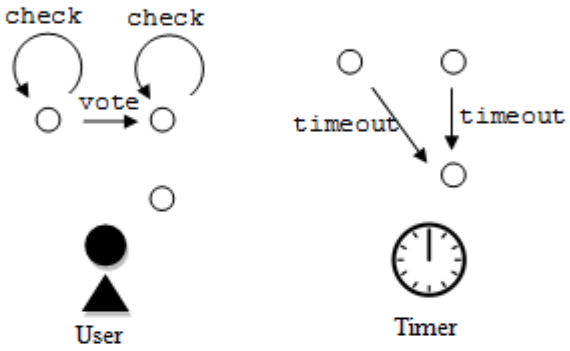
$$[\alpha]_1^{\texttt{vote}} \cdot [\beta]_1^{\texttt{timeout}}$$

With such a behavioural design, a voter can either vote or not, but once the set deadline has passed, the kernel automatically performs this `timeout` action to prevent any further votes being accepted.

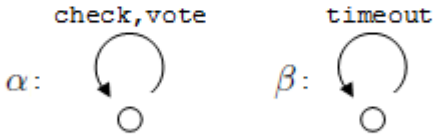
Now the organizer of the election can further refine the treaties to produce separated behaviours for voters and the timer. Let the treaty containing



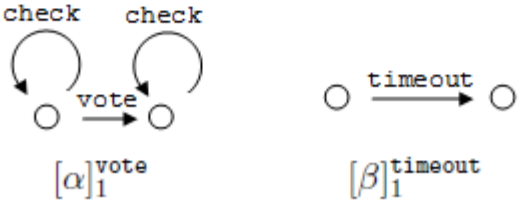
a) Overall Behaviour Model of Voting



b) Separated Behaviour Models of Voting



c) Kernel Created Complete Treaties



d) Restricted Treaties

Figure 4.4: Behaviour Models for a Timed Voting System

overall behaviour model shown in Fig. 4.4 part a) be denoted as γ. We can produce the voter held treaty by:

$$\gamma - \beta$$

or simply

$$\gamma - \texttt{timeout}$$

Similarly, the timer treaty is produced by:

$$\gamma - \texttt{check} - \texttt{vote}$$

Hence treaties containing behaviour models that are shown in Fig. 4.4 part b) are generated. Note there are states that are not reachable using any individual treaty in both behaviour models, but it is valid. Neither of voters and the timer can solely complete the whole voting process, as they rely on and are affected by each other's actions, and this is exactly the desired behaviour.

4.3 Novel Issues Generated by Treaties

Treaties extend vistas, and thus capabilities, in functionality, and so there arise some new practical and theoretical issues that need to be considered. It must also be remembered that treaties not only inherit positive attributes from capabilities such as their flexibility and scalability, but also some negative issues such as the confinement and revocation problems that have been introduced in Chapter 2, since treaties can also be propagated. Thus the development of treaty systems needs to concern these together with their particular challenges.

A main challenge that results from the new functionality of treaties is the

duplication problem. Treaties require state descriptors to keep track of the evolution of behaviours. This means that protecting the state information is crucial. According to the rule of treaties, we must ensure that allowed behaviours cannot be increased. A rollback of the current state may destroy this rule in treaties that are access-repetition sensitive, which means in these treaties there are types of actions that are only allowed to be performed with a limited number of times. If the current state in such treaties is permitted to be rolled back to one of the previous states, and between these two states there are actions with limited repetitions, to rollback the current state to the previous state means these actions can be performed once more. This repetition is not counted to the assigned and limited access times, so the rollback indirectly increases the possible access times, and this breaks the rule.

Assume that a user is holding a treaty which allows him to perform a `read` action exactly once. Then the challenge is how to prevent him from 'validly' performing more than one `read` by making a duplicate of the treaty. If this treaty is a piece of data (bit-string) completely stored in an agent's memory, there is no way of preventing the holder from making a copy of it before using it for any accesses. Then the holder can access the targeted resource using the copied treaty, and keep the unused original piece so that he can do the same thing next time he wanted to do `read` on the resource. In this way this user gains an unlimited number of accesses, which clearly breaks our rule.

The duplication problem does not only arise in this way. Taking another example of the same treaty, assume the holder A sends (a copy of) his treaty to another user B, and then uses his copy of the treaty to perform the action. Later, B sends his treaty back to A, and A now has the original treaty again! This example demonstrates that, even when they are held by different users, treaties generated from the same original one must be maintained in the

same state. The two cases of causing the duplication problem is illustrated in Fig. 4.5

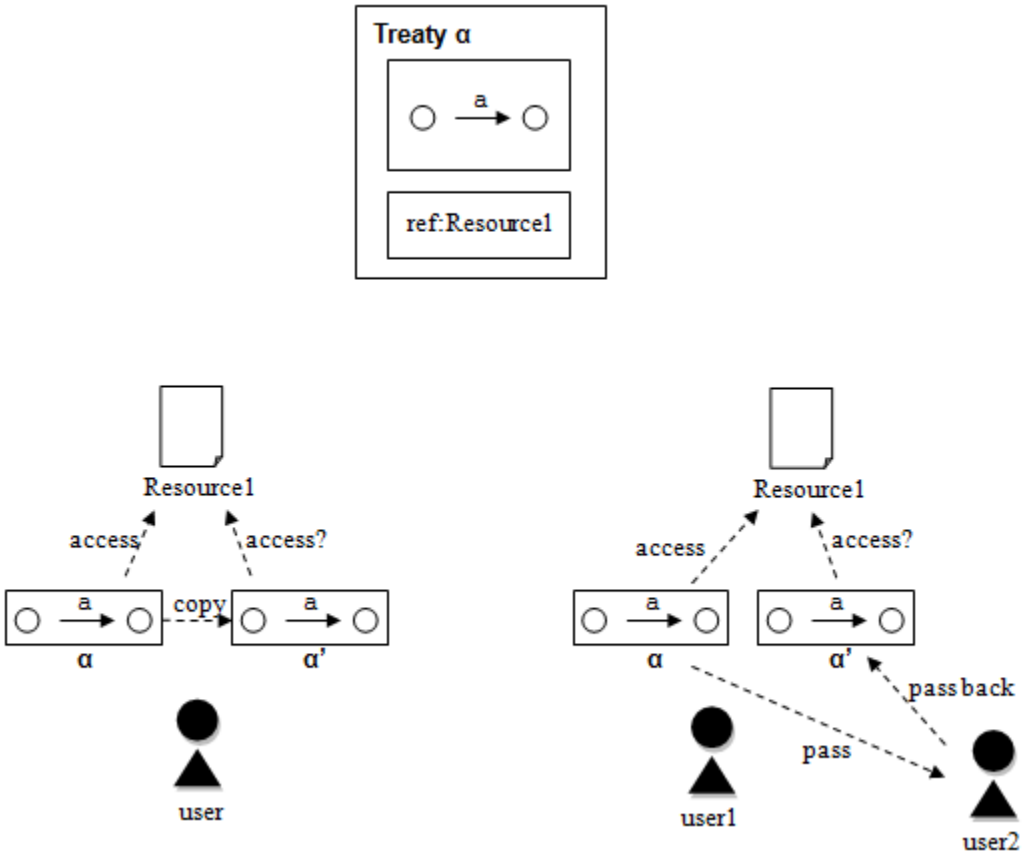


Figure 4.5: Two Cases that may Cause Duplication Problem

For similar reasons, if we were allowed to do a *concatenate* operation between a treaty and itself as being mentioned in Section 4.2, and in the treaty there are actions with limited number of executing repetitions, this will cause a duplication problem and increase the valid number of executions. This is the main reason why using binary treaty operations between treaties and themselves should not be allowed.

Moreover, the situation can be seen to be even more profound when considering treaties which have been formed by the combination operations outlined in previous sections. For instance, assume user A has received a treaty from B which allows him to do a `write` then a `read`, and has received another treaty from C that allows him to do a `write` then an `execute`, both referring to the same resource. If A were to do a *join* of the two treaties and send the result to D, and D were to request the kernel to perform a `write` action, whose treaty, B's or C's, is the correct one to be synchronized? Thus the concurrency issue also affects the design and construction of treaty operations.[3]

Generally speaking, the most important new challenges of treaties are related to states, and their management in a scalable fashion in a distributed system. This is unsurprising, as behavioural states are exactly the novel feature that treaties provide.

To solve such a problem, it seems that central control will have to act as a key part. The crucial information including the current state shall be held somewhere that is handled by the kernel, so that users cannot freely create independent copies without any restrictions. Although this may produce some centralization problems, it still grants the property of treaty propagation, and kernels do not need to identify all users, hence it is not an unacceptable cost.

4.4 Summary

In this chapter the concept of treaties has been introduced. Treaties are access control components that aim to support behaviour control in distributed

[3]This could be solved if we can distinguish the two `write` actions. If they are generated from the same action, then there is no need for concern — both treaties of B and C should be synchronized. If the two `write` are distinct, then D should tell the kernel which `write` he wants to perform.

environments. They use behaviour descriptors to capture the features of sequences, branches and terminations. The behaviour descriptors can show different states that the represented behaviour may be in. There are four basic characteristics that treaties hold, which are:

- refer to objects;
- provide access control information;
- not strictly bound to holders;
- can show different states in behaviours at different times.

The components that fit these characteristics can be regarded as treaties. The definitions of the components in a treaty system are given in the chapter.

Treaties have associated operations that allow the holder of treaties to refine the behaviours described by treaties. Five initial operations have been suggested which are *restrict, concatenate, intersect, difference* and *join*. There is a fundamental rule that treaty operations must obey: an agent cannot increase the totality of behaviours defined by the treaties it holds. We have used the idea of behaviour sets to show that the initially suggested operations will not break this rule, and this work was also provided in previous published paper [77].

The introduction of state has brought a new issue, called the *duplication problem*. This problem is caused by making copies of treaties. Treaties may have actions that are limited to certain access repetitions, and to make a copy without any protection could increase the number of accesses and exceed the allowance, which means the breakdown of access control. Any treaty system must provide some kind of solution to the duplication problem to ensure its integrity.

In this chapter only the abstract concepts of treaty systems have been given, without any details of representing and implementing actual systems.

These issues will be explained and illustrated in following chapters.

Chapter 5

Representations of Treaties

The concept of Treaties introduced in Chapter 4 is quite abstract. It only states the basic characteristics of treaties, and some implicitly defined operations. Any system or service that meets the requirements and enforces the rule of treaties can be considered as a treaty system. There, nothing of how to organize a treaty system was explained, and in practice treaties can be represented in many ways, and there are a number of options for each treaty component. In this chapter we will extend the abstract ideas of Treaties and show how to convert these ideas into practice. The various choices or options for the structure of treaties will need to be defined. Each option has its strengths and weaknesses, so service providers can choose to combine different kinds of representation depending on their requirements. We call a type of structurally defined Treaty a *Treaty Representation*. Different representations of Treaties may have different characteristics which match different cases. The representations of treaties may also affect the ways of solving the duplication problem. These options and structures for representing treaty systems will be examined in this chapter.

5.1 Object Referencing

Options for the style of referencing concern the number of resources that one treaty can refer to, and the relation and structures of these references. In the abstract concept of Treaties mentioned in the previous chapter, each treaty can only refer to one object, but it is valid to allow treaties referring to multiple objects to upgrade the functionality of treaties. Hence there are a number of different options. For example, we can choose to let each treaty refer to only one object, as in the capability approach. This is a simple choice that provides straightforward resource references, and no additional facilities are needed to identify mappings between actions and objects. Conversely, this choice means that it is not possible to operate on multiple objects, whether simultaneously or non-simultaneously. If a user wants to do the same action to many objects of the same type, he would need to pass treaties for each of these objects separately which, in a large-scale distributed environment, would limit efficiency. Possible applications to deploy this kind of treaty referencing style would be systems where objects are independent and separately accessed, such as in an office environment, having printers, scanners, telephones and a clock with alarm. Assuming that none of these machines needs to interact with others, then the treaties used to access them would be separately dealt as well.

Another choice would be to allow treaties to refer to multiple objects, but hold the behaviour descriptor separately (this would be similar to a set of single-reference treaties). It would then be possible to deal with actions on multiple target objects, but since behaviour descriptors are independent, they are not capable of describing sequences of actions among different objects. This choice is particularly suitable for applications that have a number of objects of a same type. A good example when this referencing style could be useful is the administration of a university. It holds data for a large

number of students, and the profile of each student is of the same type. The administration may want to change or update information for single or groups of students, but would not need to do these in a particular order, and thus no sequences among these student objects are required.

The third choice is to let a treaty be a 'mix'. These treaties' behaviour descriptors would include actions on multiple objects, making it possible to describe sequences over different objects. Clearly this is the most general, and powerful option, as behaviours specified by the other two options can only refer to a unique object. The disadvantage of this choice is that each action in a treaty's behaviour specification would have to identify the target object, so the complexity of the structure would increase greatly. An example where this referencing style could be applied is: assuming a user needs to transfer a file from a public source to a local storage, but he would like to have a check whether for any harmful contents are in the file before it is transfered to the local storage. Treaties that grant this process will need to access three targets of public source, the guard that provides the security check and the local storage, and these actions are needed to be in sequence: first get the file from the source, then go through the check, and finally save into the storage. Only mixed-referencing of treaties can do this.

5.2 Behaviour Descriptors

Behaviour descriptors are required to define specifications of behaviours. Behaviours are defined as a group of actions that are combined in a certain order and pattern, where an action is a single operation on a resource. Any logic, mechanism or language that is able to represent behaviours of this form can be a candidate in the behaviour descriptor part.

One obvious candidate is the finite-state machine (FSM), which is a widely-understood concept. If FSMs were chosen to represent treaty speci-

fications, then transitions in an FSM would represent actions. The FSM is ideal for representing repetitions, sequences, branches and so on.

Regular expressions [22] would also be a convenient way of representing treaty representations. As regular expressions, actions are represented by characters, boolean '|' for branching and '*' for repetition. These two options will be discussed further and their implementations are introduced in Chapter 6.

We may also use Process Algebras to represent behaviours, as Process Algebras are designed for representing interactions, behaviours and communications for independent processes, and have algebraic laws for process operations.

Choices of behaviour descriptors can be made depending on different applications, as different options result in different effects.

5.2.1 Finite-State Machines

Using finite-state machines to represent the behaviour descriptors in treaties, states are explicit. A behaviour descriptor represented by an FSM will have a finite number of states (at least one), and a finite number of transitions that represent accesses or actions. Each transition links two states (which do not need to be distinct), one denotes in which state this action can happen, and the other denotes which state the behaviour descriptor will be in after this action happens. In an FSM representation of behaviour descriptors, the *current state* is also maintained. This indicates the state of the holder of this treaty by pointing to one of the states in the FSM represented behaviour descriptor. As it has been mentioned, the FSMs used as behaviour descriptors will not have any 'accepting states', as the access control scheme only cares about whether the user's required action is valid or not, but not when the

actions should stop.[1]

The advantage of using an FSM representation is the precisely defined states. They are directly contained as components in the behaviour descriptors, so it is possible to add information to states to provide more control. For example, we could mark some essential states, and the kernel would get immediate notification when these states were reached. For example, imagine in a house there are a number of electronic products, and there are treaties controlling the power of these machines. States in treaties represents the number of machines that are on. When there are too many machines turned on, the fuse would blow. Hence when the specified state of the critical point is reached, the kernel will be notified, and it can produce an alert telling the danger. This idea also benefits the treaty combinator operation of *follow* which will be introduced in Section 5.4.

5.2.2 Regular Expressions

In the Regular Expression (RE) representation, accesses or actions are represented by literal characters, or words, and we will use *concatenation*, *alternation* and *Kleene star* to represent aspects of behaviours. There are no explicit components that represent states in RE behaviour models. The current state pointer in RE representations can be modelled by the 'remaining' expressions, and changes when the actions are performed and the expressions are processed. For instance, the current state of an RE represented behaviour can be '`(read.write)|(execute)`' that means the holder can do an `execute`, or a `read` then `write`, and after an action of `read` is performed, the remaining expression and the current state is '`write`.'

Compared with the FSM representation, RE representation is easier for human reading and writing/typing, especially when constructing the be-

[1] In other words, every state in a treaty is an accepting state.

haviour model directly from a line of input. The main difference between the two representations is the appearance of explicit state components. The RE representation may have limits due the lack of identifiable states, as the current state in the RE representation is denoted by the remaining expressions. It is possible that there are two remaining expressions that are exactly the same, but they came from two different paths (for example, '`a.b`' that is obtained from '`x.a.b`' where `x` had been executed, and '`a.b`' that is obtained from '`y.a.b`' where `y` had been executed) and should have different prompts to users and kernels — in the RE representation it cannot distinguish the two states (while the FSM representation can, as in that representation the states are explicit). Despite this limitation, the RE representation is still a good choice if the application does not require such functionalities. Besides, no state components means simpler structures, and may allow treaties in this representation to have smaller space requirements and faster processing speed.

5.2.3 Process Algebra

Process Algebra (or Process Calculus) is the concept that is used for 'the study of the behaviour of parallel or distributed systems by algebraic means' [3]. A process algebra is usually used for formally modelling concurrent systems where multiple processes exist and interact. It provides the description for the consequences, communications and synchronizations for those processes or agents, in algebraic ways. As the algebra laws are involved, a process algebra is particularly applicable for analyzing and reasoning the behaviours of processes in distributed systems. The most famous process algebras are CSP [32], CCS [51], ACP [7] and π-calculus [50].

Take CSP as an example. The *Communicating Sequential Process* is a formal language for describing interactions in concurrent system, and is par-

ticularly used for providing a way to model behaviours in distrbuted environment. If we use CSP notions, a simple *read-once* action can be expressed as '$read \rightarrow Stop$', and a process of consecutive actions is made by having event (action) appended to the head, e.g. '$write \rightarrow (read \rightarrow Stop)$'. Sequences of processes uses symbol ';', hence '$P; Q$' denotes 'behaves as P until it terminates, then behaves as Q'. The branch or choice aspect in CSP is denoted by '$\sqcap$' (internal choice) and '$\Box$' (external choice), and they are nondeterministic and deterministic respectively. In context of access control oriented branches, the deterministic '$\Box$' seems more suitable. The process of choosing between *left* or *right* is expressed by '$(left \rightarrow Stop)\Box(right \rightarrow Stop)$'. The loop behaviours can be defined by recursive processes, e.g. '$UPDATE = read \rightarrow (write \rightarrow UPDATE)$'. More tutorials for using CSP can be found in Davies's work [16].

CSP and Process Algebras are widely studied in many other researches as an independent topic. Since they have relatively complete and complex structures, it will take much more effort in learning and applying CSP in this project. Therefore we will not look in deep for this option in the thesis.

5.3 Storage Location of Treaties

The storage location of treaties directly affects the solution to the duplication problem. As has been mentioned, if treaties are completely held in users' (private) memory, it is possible that they could copy a treaty and gain unauthorized behaviours. We can choose to store treaties in a different location to avoid this problem. Otherwise, treaties can be left in users' devices, but the critical data of treaties still needs to be protected by kernels.

A simple solution is to store treaties centrally. There would then be a central storage for all treaties in the service provided, and users would only hold references to these treaties. When users needed to access resources,

they would present the references to the kernel, and the kernel would check the corresponding treaties to allow or deny this access. Since only references would be held by users, it would no longer be necessary to manage how users copy and propagate them. However the side effect of this option is obvious: the centralization management significantly restricts the scalability of systems, and the performance will heavily depend on the storage's processing speed, bandwidth, etc. It also becomes fatal if the central storage fails, where all accesses will be denied.

The option of allowing users to hold treaties together with their current state has the highest scalability, which is very attractive in distributed systems. However, it has been shown in Section 4.3 that this could be dangerous, as it might cause the duplication problem and destroy the behaviour control provided by treaties. It is still possible to distribute (the major part of) treaties to users though, providing that the critical information of these treaties are in the control of kernels. It is clear that only the current state of treaties needs to be protected from duplication or interference. Consequently a promising compromise approach is to have this state managed by the kernel, but the reference and behavioural specification parts made available for (safe) user management.

5.3.1 Centrally Stored Treaties and References

A very simple way of solving the duplication problem is to take treaties away from users completely. In this case, users cannot access treaties directly any more, and thus are not able to make copies of treaties or send them to other users. Treaties will now be held in somewhere that can only be accessed by kernels. In return, users will only have references to those treaties. In this approach, together with the behaviour descriptor and the current state and the reference to the target object, there will be a fourth part in treaties:

a unique identifier for each treaty. The identifier will also be contained in treaty references held by users. Users can copy and send these references as many times as they like, as treaties are still kept unique on the kernel side. Fig. 5.1 shows an example of the structure of a treaty system that applies the reprensentation of centrally stored treaties.

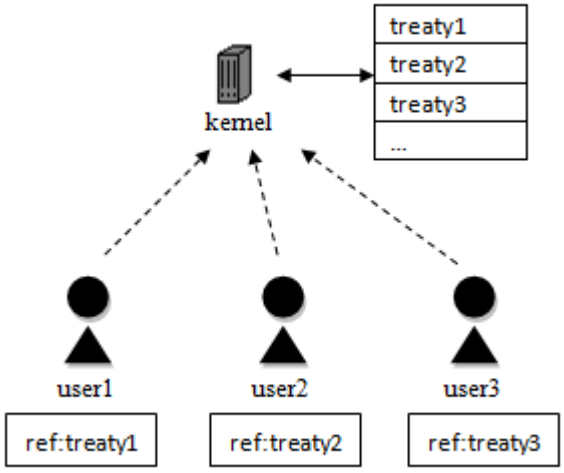


Figure 5.1: Structure of a Centrally Stored Treaty System

The process for preparing behavioural access control in such a model would be:

- A user process sends a request to the kernel, to create an object.

- The kernel creates the object, and creates a 'complete treaty' (unlimited accesses to all available action types for the object). This treaty will be stored in the treaty container that is managed by kernels, and the reference to this treaty, with the unique treaty identifier, will be given to the creator of the object.

- Then the creator can refine the behaviour descriptor by telling the kernel how he will use treaty operations.
- The kernel then constructs a new treaty and stores it in the container, and sends the new reference back.
- The creator can now send copies of this reference to other users, so that they share the access rights to the created object.

The process of performing an action on an object would be:

- A user needs to send a request together with the treaty reference.
- Upon receiving the request, the kernel then uses the identifier contained in the reference to locate the proper treaty, and checks whether the requested action is valid according to the behaviour descriptor in the treaty.
- If it is permitted, the action will be performed on the target object, otherwise the kernel will send a denying message implying that the action is not valid at this time.

This model is simple, but it has a weakness. In real cases it is quite possible that users do not know the current valid actions regarding the behaviour descriptors. And without access to the treaties, they will have to either *a*) send access requests without any knowledge of treaties or, *b*) send a query asking what types of actions are available at the moment. In both cases, the number of messages used for communication will increase.

However it needs to be mentioned that, in the case of option *b*, the returned 'available' types of actions might not be granted when the users try accessing objects. This is because there is another user who also has a reference to the same treaty, and he made an action using this treaty after the kernel returns the available action set and before the first user makes the

accessing request. This is the non-deterministic problem that many concurrent systems may suffer, where processes share common resources and may disturb each other. A possible solution is to include locking scheme, and this will be discussed in future work part of Section 8.2.

5.3.2 User-held Treaties and Treaty Entries

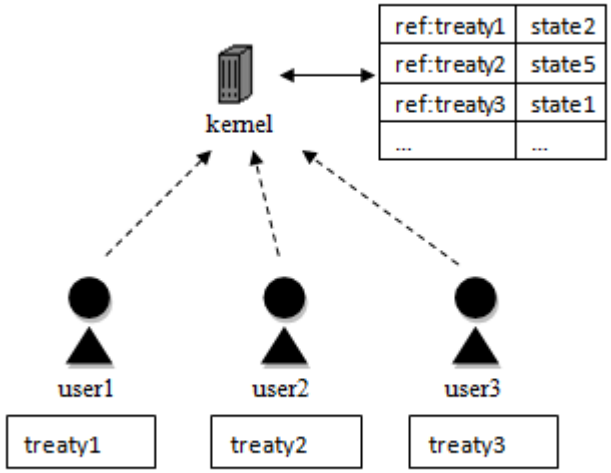


Figure 5.2: Structure of a Treaty System with Entry List

In contrast to the previous implementation which locates treaties on the kernel side, this model lets users hold treaties, and kernels only keep a list containing a number of entries, each of which stores the information for a treaty. Treaties still have their unique identifiers, and they are contained in entries as well, so that kernels can link them to treaties. Each entry consists of two parts: in addition to the treaty identifier, there is the current state of the related treaty. Although users can make copies and distribute treaties, these copies still have the same identifier and current state, which prevents

the duplication problem happening. This approach also indicates that for a particular behaviour descriptor, all states are distinguishable. Each state is tagged to show the identity that makes the state different from other states in this behaviour descriptor. Fig. 5.2 shows an example of the structure of a treaty system that uses the entry list option.

To initiate the creation process:

- A user sends a request for creating a new object.
- The kernel creates the object and the complete treaty for the user, and an entry containing the identifier and the initial state for the treaty.
- The entry is added to the entry list, and the complete treaty is sent to the creator of the object.
- The creator can use treaty operations to refine the complete treaty by using treaty combinator operations.
- Before these refined treaties can be sent to other users, they must be first sent to the kernel for registration.
- The kernel assigns identifiers to these treaties, and records them by adding corresponding entries to the list.
- Now these treaties are ready for propagation.

If the user does not make the registration of their newly refined treaties to kernels, the kernels will not hold any information of these treaties. When users use these unregistered treaties to access objects, kernels simply deny these requests.

The process to access objects in this approach:

- A user needs to send the request with the correct treaty.

- When receiving the treaty in the request, the kernel first looks for its entry, and then checks whether the current state represented in the treaty is the same as represented in the entry.

- If the two states are the same, and the requested action is valid in this state, the kernel will update both states in the treaty and the entry to the new state according to the performed action, and return the treaty.

- If the two states are different, the kernel will synchronize them by updating the current state in the treaty with the state in the entry, and return this treaty to the user, notifying that this treaty has been updated.

Such an approach reduces the communication messages to some extent, as users can check the *possible* current state from treaties they are holding. Though there may still be cases where users get outdated information when the real state in the entry held by kernels has been changed by other users, it is suitable for applications which have few users sharing objects. This is also because when multiple users share the same object, the copy each of them holds is a treaty. Larger number of users holding treaties means larger consumption for overall space/memory, compared to references held by users in the previous approach.

5.4 Operations and Executions

Treaty operations need not only be restricted to the five types given in Chapter 4. Any useful operations could be included in treaty systems, as developers and service providers can introduce new treaty operations to fit their own requirement, as long as they do not break the rule. Also, the way of executing commands and operations can also be varied.

5.4.1 More Operations

Here some possible additional treaty operations are suggested that could be appended to treaty systems. There are certainly more available choices, but these ones may be commonly used.

Interleave A frequently seen situation would be that a user may receive treaties from multiple sources and would therefore hold a number of treaties. It could be a mess if he need to perform an action, and tried to find the correct treaty that will provide the corresponding permission from the treaties he is holding. Hence it is more convenient if there were an operation that combines treaties into one, with the result allowing the holder to do all behaviours from the original treaties in parallel. This cannot be realized by the *join* operation, which would only allow one to choose one of the behaviours from the original treaties but not in parallel. This interleave idea has been mentioned in Section 4.2 indeed, and is the concept of *shuffle*. The shuffle idea was used for the largest behaviour set that the result of any treaty operations should not go beyond.

Obviously the result of *interleave* fits the rule of treaty operations, as this operation produces the maximum behaviour set from the two operand treaties, which is the equal set of the upper bound set.

Follow The *follow* operation is similar to the *concatenate* operation except that, given the behaviours in two original treaties, *concatenate* allows behaviours from the second operand treaty to act after *any* behaviours from the first operand treaty, while the *follow* operation only allows behaviours from the second operand treaty to act when behaviours from the first operand treaty have reached a certain point. For example, given two treaties, α: (`read.write`) and β: (`execute`), if we do a

concatenate operation to α and β, the behaviour set of the resulting treaty would be:

$\{\epsilon$, `read`, `execute`, `read.write`, `read.execute`, `read.write.execute`$\}$

Now let us label one of the states in α to make it a 'reach point' state. A reach point state is the state in which we can start performing behaviours from the second operand treaty of the *follow* operation. In treaties resulting from the *follow* operation, the behaviours from the second operand treaty will not be available to perform (unlike *concatenate* where behaviours from the second treaty can start at any time), until the behaviours from the first operand treaty has reached the reach point state. If we have defined a reach point state for α after the `write` action, and we now do a *follow* operation to α and β, the behaviour set of the resulting treaty would be:

$\{\ \epsilon$, `read, read.write, read.write.execute` $\}$

We use $\bullet$ to denote the *follow* operation, and the formal definition for the *follow* operation between treaty α and β at state s in α is given by ($b_1 \triangleright s$ denotes the behaviour b_1 ends at state s):

$$[\![\alpha \bullet \beta\ ]\!] = \{\ x \mid x = (b_1 \triangleright s).b_2 \text{ where } b_1 \triangleright s \in [\![\alpha\ ]\!] \text{ and } b_2 \in [\![\beta\ ]\!]\}$$

This operation also fits the rule of treaty operations, as it is a special type of *concatenate*. That is,

$$[\![\alpha \bullet \beta\ ]\!] \subseteq [\![\alpha \odot \beta\ ]\!], \text{ thus } [\![\alpha \bullet \beta\ ]\!] \subseteq [\![\alpha \cdot \beta\ ]\!]$$

We may find the operation of *follow* more often used when we want to refine behaviour models, as it can explicitly define the connecting point between the two behaviour models. However to realize this operation, the option chosen for behaviour descriptors must provide the functionality to support specifying the reach point. In the FSM representation of behaviour descriptors, we can simply use state identifiers to do this.

In the RE representation, we can only set the reach point as the 'finish' of the expression, where the behaviours in the expression have reached their ends.

Loop This is the operation for creating loops of actions so that actions in the refined behaviours can be potentially infinitely performed. In RE representation style, if we have a treaty α: (`read.write`), the result of *loop* operation is simply (`(read.write)*`). However this operation must be specified carefully, because it may easily produce unlimited actions and break the rule of behaviour control. The *loop* operation can only be valid if the holder of the treaty already has a treaty that provides the loop of actions which one wants to contain in the created loop. At this stage of the research, we only allow this right of creating loops with *loop* operation to open to users who possess the complete treaty that allows them to do anything to the object with unlimited number of repetitions.

5.4.2 Command Executions

We may also want the system to have different semantics for executing the same command. For instance, assume there is a treaty that has references to multiple targeted objects, such that all of them are allowed a `read` action by this treaty. When presented with this treaty, a `read` request could mean: to do a `read` on *one* of the objects, or to *some* of the objects, or *all* of the objects. Developers can have different definitions for the executions of the `read` command to realize the options above, depending on their preferences. For instance, if a university wanted to change the *grade* information for all of the students (since normally all students get upgraded at the same time), it could choose to make an `upgrade` action that is executed to all objects of student data simultaneously. However it might be the case that some of the

students fail to pass examinations and will not get an upgrade. Hence here the university needs to perform a `degrade` action to that group of student after the `upgrade` is made. Actions executed to a single object are also required, such as a `ChangeInfo` action can be performed if there is a student pointing out that his record in the university system is wrongly inputted, and this action should affect the student information object of only this student.

5.5 An Example of Organized Treaty Representation

In this section a representation of treaties will be proposed. It is called the 'Two-layered Referring FSM Representation'. This representation of treaty systems has fixed choices to the options mentioned in previous sections to give an complete example. In particular, it proposes a new option by combining options given in Section 5.3, and provides a different solution for the duplication problem.

As indicated by the name, the representation chooses finite-state machines as the behaviour descriptor, hence each time a valid action is performed, an explicit state transition happens in relevant treaties. The five treaty combinator operations of *join, follow, intersect, difference* and *restrict* are supported by this representation, and they will operate in ways as defined. The key novelty of the Two-layered Referring FSM Representation is shown in its storage location choice. In this representation, treaties are neither stored centrally, nor completely distributed. Instead, there are two different categories of treaties appearing in the system.

The first kind is called 'source treaties'. These treaties are stored on the kernel side, and each one can only refer to one object. Source treaties can be accessed by kernels and the owner of the referred object (the creator of

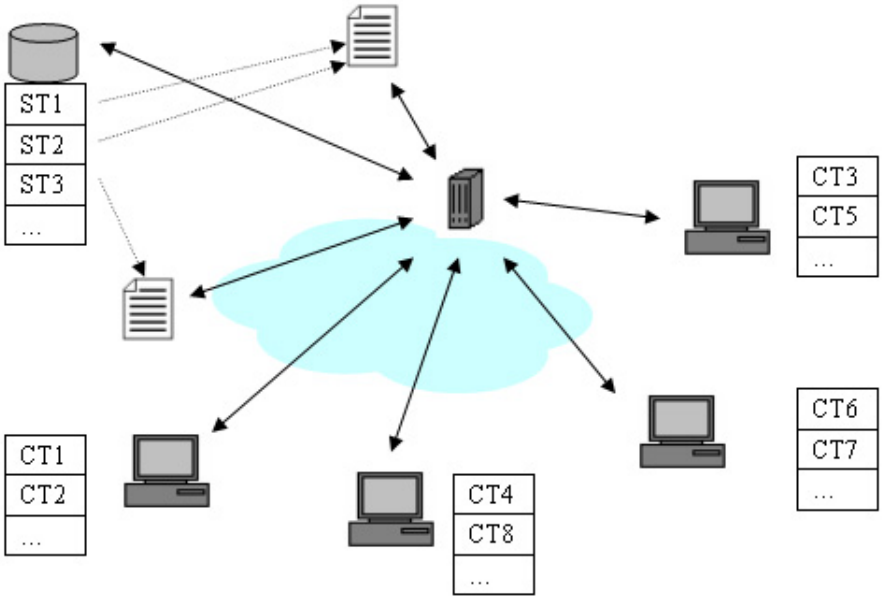


Figure 5.3: Two-Layered FSM Representation

the object and those users who are allowed to share the high privilege by the creator), but not any other users. Actions granted by the source treaties will be represented by transitions in the finite-state machine acting as the behaviour descriptor.

The other kind of treaties in the Two-layered Referring FSM Representation is called 'combined treaties'. These treaties are generated from source treaties, and they are distributed to users that are not the creator of objects. This is how this scheme of 'Two-layered referring FSM representation' transmits the permissions to other users.[2] In combined treaties there are also finite-state machines acting as behaviour descriptors, but combined treaties do not directly refer to any objects. Instead, each transition in behaviour descriptors of combined treaties refers to one transition in a certain source treaty. When the holder of a combined treaty uses it to perform a permitted action on the targeted object, both the corresponding transition in the combined treaty and the referred transition in the source treaty will be processed. Fig. 5.3 and 5.4 illustrate the structure of this representation.

To initialize, a user will apply to the kernel to create an object. The kernel creates the object, and returns a complete source treaty granting unlimited repetitions for all actions that are available to this type of object. After this, the creator of the object can use this complete source treaty to create treaties with more limited behaviour models, using treaty operations. At this stage all treaties created by the creator of the object are source treaties. When a behaviour model is obtained as a result of a treaty operation, the creator can store it in the storage of the kernel for distribution uses. Each source treaty can be used as a model to build a combined treaty having exactly the same finite-state machine states and transitions that represent the behaviour. Each transition in the generated combined treaties refers to

[2]It is possible for the creators to send the source treaties out as well, in which case the receiver of the source treaty also becomes an owner of the created object and is allowed to refine the behaviours in the source treaty.

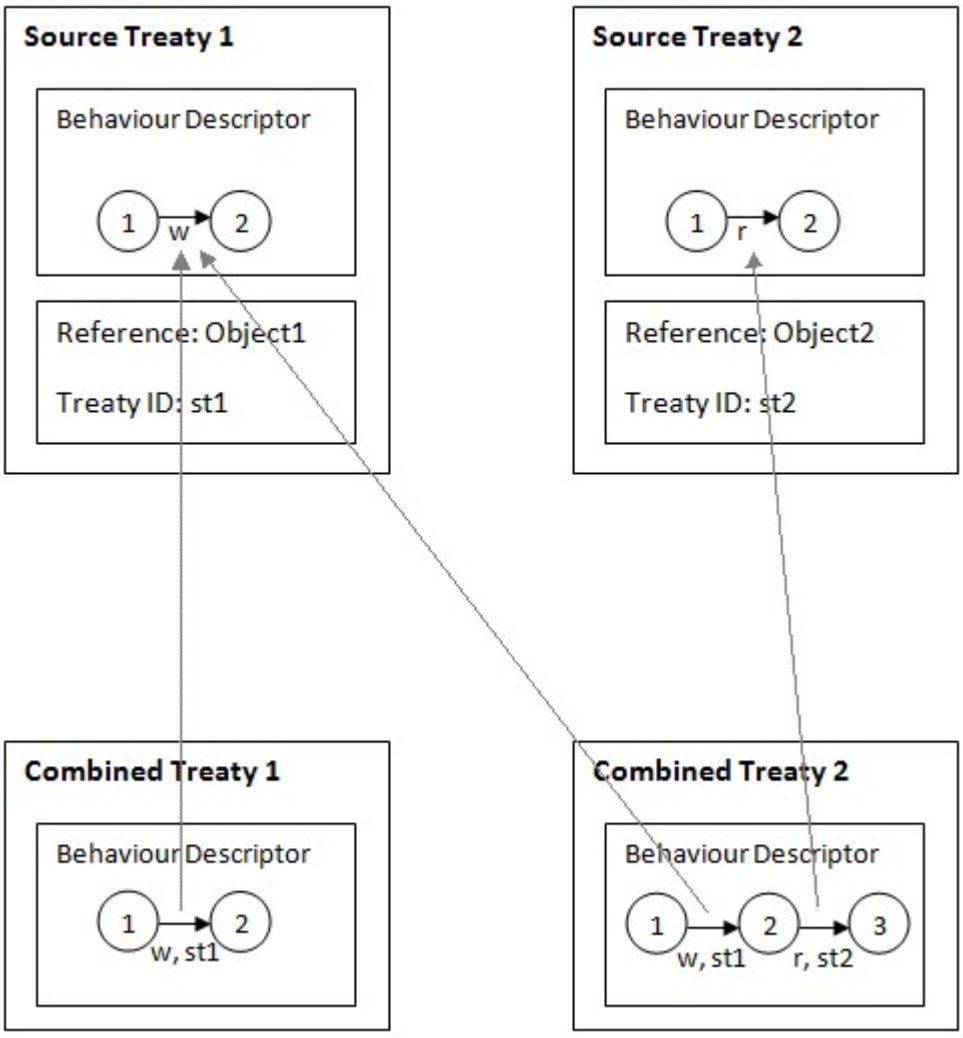


Figure 5.4: Referencing Structure between Two Categories of Treaties

the original transition in the source treaty.

After this, the creator can send out copies of combined treaties either actively or by request, and these copies are built from the source treaties this creator is holding. When other users have received multiple combined treaties, they can use treaty operations to refine the behaviour model to generate new combined treaties and send them out as well. This also indicates that there may be transitions in the same combined treaty that refer to source treaties of different objects, which means that the referencing style of the Two-layered Referring FSM Representation is mixed multiple referring (see Section 5.1).

The reason for this design is that however users make copies and propagate to duplicate combined treaties, their transitions will still refer to the transition in source treaties which are not duplicated. And after one user performed an action transition, the other users will not be able to do the same transition because the transition in the source treaty that grants this action has been processed. In this way, the duplication problem has been solved in this representation.

The five support operations will be implemented as follows:

- *restrict* limits the given type of action to a specified number for a source treaty, and for a combined treaty one can choose to restrict the given type of action only on one specified object or restrict over all objects that are contained.

- The operation of *join* will return a treaty whose current state is a union of the two original treaties, which means it will accept actions in either original treaty at the current state.

- The operation of *intersect* returns a treaty that allows sequences of actions that are valid in both original treaties. When operated over combined treaties, only transitions from the two original treaties that

refer to the same transition in a source treaty will be counted as common actions.

- *difference* does the operation in just the opposite way as *intersect*, as common actions will be removed from the resulted treaty. And the consecutive actions following these common actions will also be removed, as they are no longer reachable from the current state of the resulting treaty.

- The operation of *follow* connects the current state of the second operand to the reach point state of the first operand. To perform such a concatenation, the first operand must have had specified a reach point state.

The situation of concatenating two identical treaties will not cause the duplication problem, since when concatenating a combined treaty to itself, duplicated action requests will not be granted because of the two-layered structure, and concatenating a source treaty to itself is completely legal because operations on source treaties can be only made by the owners of the targeted object and the owners have full permissions which allow them to do any number of accesses to that object.

Note that these binary operations are only valid between two source treaties or two combined treaties, and the result treaty is of the same kind as the original treaties.

These five are the fundamental operations that are provided by the Two-layered Referring FSM Representation, but it is perfectly possible to add other treaty operations into the model. However, as usual, the new operations must obey the rule of treaty operations, and ensure that we can never get source treaties from combined treaties.

5.6 Summary

In this chapter we have discussed the representations of treaties and treaty systems. Treaties as proposed in Chapter 4 were general concepts, without any details of actual systems and applications. In the design of treaty systems there are several areas that give a number of options, and each option may have different aspects from others such that they can be prefered to be chosen in some cases. The parts that can be varied in different options are: the choice of referencing styles, the choices of behaviour descriptors, the location where treaties are stored, and treaty operations and ways of executing actions other than the basic ones that were introduced in the previous chapter. Treaties or treaty systems that have these parts defined are called *treaty representations*. Developers and service providers can chooes to combined these options and produce treaty systems according to their requirements.

- The choice of referencing styles means that one treaty can be set to refer to only one target object, or separately referring to multiple objects, or having a mixed referencing style such that treaties can define sequences of actions over different objects.

- Behaviour descriptors in treaties can be chosen to be displayed in different tools such as finite-state machines or regular expressions, or other choices that can express the aspects of behaviours, and these choices of behaviour descriptors may have effects on the performance of treaty systems.

- The location where treaties are stored can affect the memory and bandwidth occupation of those systems. We may choose to store treaties on the kernel side, or to store them in the possession of users, but in either case attention must be paid to solving the duplication problem discussed in Chapter 4.

- Developers and service providers can define new treaty operations and choose the way of executing actions in their treaty systems, as long as these decisions maintain the integrity requirements of treaty systems.

Sections 5.1 and 5.2 mainly address the inner structures of treaties, whereas Sections 5.3 and 5.4 are more focused on the structure and processing of treaty systems. Section 5.5 provides a example of a treaty system named 'Two-layered Referring FSM Representation', in which all options have been defined. The discussion of solutions to the duplication problem was also introduced in the previous published work [78].

Options to combine treaty representations have been discussed, and now we can put these into implementation. How treaties are structured and how the algorithms of treaty operations are designed will be introduced in the next chapter.

Chapter 6

Implementation

In this chapter the implementation of treaty systems will be introduced. These implementations cover the several options of treaty representations discussed in the previous chapter. The main aspect of treaties that differs from capabilities is their use of behaviour descriptors, hence this should be emphasized. Two options for behaviour descriptors, *finite-state machines* and *regular expressions*, have been chosen to be implemented and used for testing. The implementation of other aspects of treaties is also shown in the chapter.

Before any details are given, it is necessary to clarify that in this implementation, it is assumed that capabilities, vistas and treaties are unforgeable by entities that hold these access control components. The encryption or other mechanisms that protect these components from being falsified should be provided, and we presume this protection already exists. Hence no anti-forging issues will be addressed in the implementation part. Similarly, concurrency control is also out of the scope of the implementation. It is assumed there are other concurrency control mechanisms provided. There is some discussion about concurrency issues in Section 8.2.3.

6.1 Implementation Overview

The implementation platform chosen is Java which is a widely-used object-oriented language. As there are various kinds of entities in treaty systems — kernels, users and files, and treaties — the object-oriented concept is particularly suitable in this situation. Most terms that have been defined in Chapter 4 are implemented as classes in Java.

The implementation of treaties is based on two representations that differ in the choice of behaviour descriptor: finite-state machine (FSM) representation and regular expression (RE) representation. Being the essential part of treaties, behaviour descriptors affect their structure and the design of algorithms for treaty operations. The reason for choosing the two representation is that they are well-understood in computer science, and are easy to apply and present. The actual structure of finite-state machines and regular expressions in this implementation is coded by me instead of using any provided tools, as it gives a better customization and can have more adequate evaluations when having experiments. We will introduce both of the representations, including the required classes with their purposes, the fields and methods in these classes, and the treaty combinator operations.

Other components to simulate treaty systems will also need to be implemented, including kernels and users, and the objects that are used to be accessed by users. There are also classes that are used for supporting evaluations, such as simple capabilities and vistas that will be used to compare performance with treaties.

The simulation of a distributed environment will use the *SimJava* tool [35]. SimJava is a toolkit for building working models of complex systems. As its name indicates, this is a tool that is written in Java and supports other Java applications. Using SimJava a framework of entities can be produced in systems by implementing entities as subclasses of the defined classes in

SimJava packages. There are a number of built-in methods that help to simulate the communications among these entities. Functions for producing evaluation reports are also included in this toolkit.

Instead of implementing the *concatenate* operation introduced in Chapter 4, the *follow* operation introduced in Chapter 5 is chosen to be coded. This is because that when refining behaviour models in treaties, the *follow* operation is more useful and applicable than the simple *concatenate* operation, as the *follow* operation requires the 'reach point' component to connect two behaviour models, and this makes the refining of the treaties more precise. Since we allow the empty behaviour in all treaties, if we perform a *concatenate* operation, it means behaviours in the second operand can start before any actions in the first operand are performed. In case of *follow*, it can set the behaviours in the second operand starts only after actions in the first operand has reach the defined point.[1]

6.2 FSMs as Behaviour Descriptors

The first implemented representation of treaty systems is chosen to use the finite-state machine representation. In this representation, the behaviour descriptors in treaties specify the steps of behaviours by a number of states, and each valid action as a transition between states. There is a pointer referring to the current state in the behaviour specified by the descriptor.

The class where object references are placed depends on referencing styles (see Section 5.1). If it is *single-object referencing*, then a treaty can have one reference which is placed in the class of Treaty. If it is *multiple-objects referencing*, then each of the behaviour models in the treaty will have an object reference. In our implementation, we choose the latter option which

[1]But remember that *follow* needs support from behaviour descriptors to specify reach points, while *concatenate* is a more general expressed operation without special requirements. Hence *concatenate* would still be the basic treaty operation.

is to place the object reference within the behaviour descriptor. Fig. 6.1 shows the structure of classes and methods in the FSM implementation in UML style.

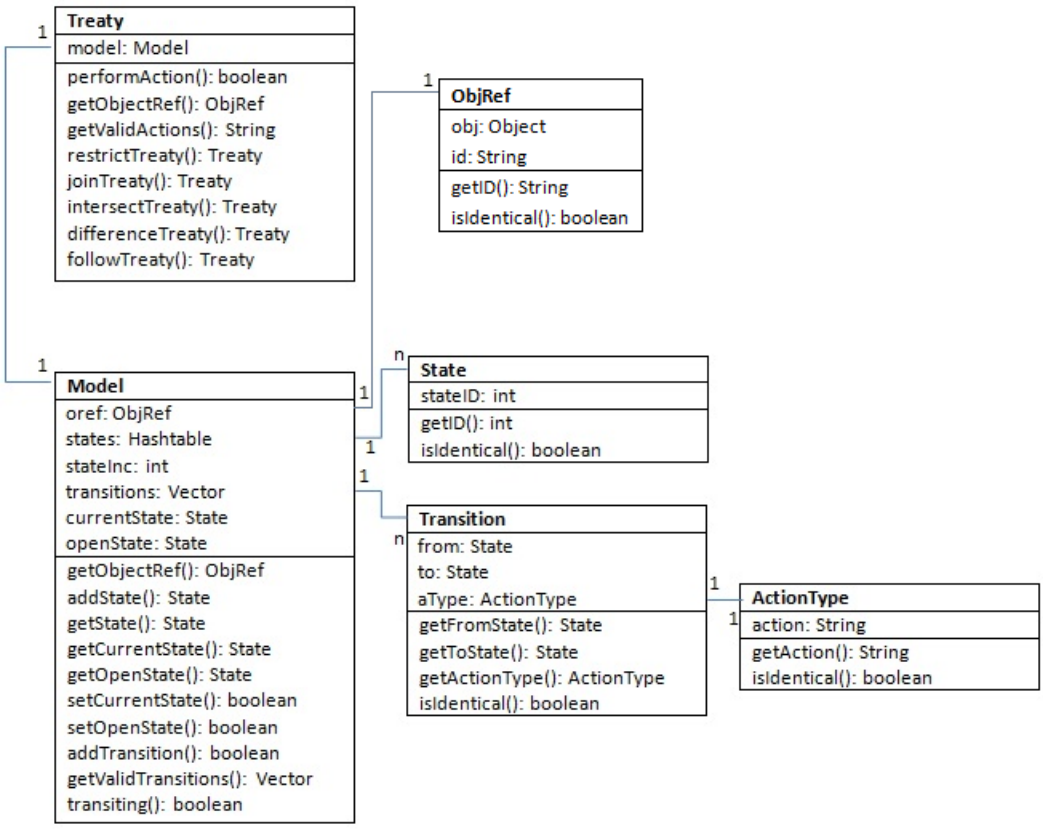


Figure 6.1: FSM Treaty classes in UML form

6.2.1 System Components and Classes

An FSM style treaty consists of several components to be implemented by classes in Java. Below these components are described:

`ActionType` This is the class that is used for different types of actions. Examples of action types could be: `read`, `write`, `vote`, etc. The field variables in this class are unique, and it is the name of the type of actions.

`Model` This class represents the behaviour descriptor part of treaties (recall that the behaviour descriptor describes behaviour models). In the FSM implementation the `Model` class contains a set of states, and a set of transitions that connect pairs of states. It also contains the current state pointer referring to one of the states in this `Model` instance. There is another state pointer called `openState` which records the reach point that connects other models when executing the *follow* operation.[2]

`ObjRef` This class is the object reference that is contained in treaties referring to the target resource. There are two field variables: the pointer to the target object, and the unique identifier which should be distinguished from any other instances of `ObjRef`. The instances of the `ObjRef` class are not only used for referring to resources, but are also available to act as treaty references when required.

`State` This specifies states in the FSM representation for behaviour descriptors. Each instance of the `State` class simply has one field variable called `stateID`, which can help when indexing states. The `stateID` only needs to be distinct in the same instance of `Model` class, but they

[2]In this implementation we assume we can only define one reach point in a behaviour model at a time. If multiple reach points are needed, they are done with *follow* operations one by one. This is not an optimized approach.

are independent from states in other `Model` instances and hence can be identical.

`Transition` An instance of the `Transition` class denotes an action applied by users and a transition between two states in the FSM representation. Three field variables are contained in this class: the `from` and `to` variables are two instances of the `State` class, and the `aType` is a variable of `ActionType` class.

`Treaty` This class integrates all classes mentioned above to implement FSM represented treaties. There are different options of referencing styles that can be chosen. For the simple case of single object referencing, an instance of Treaty contains one instance of `Model` which acts as the behaviour descriptor, having one `ObjRef` instance for the target resource and one current state pointer.

6.2.2 FSM Accessors and Access Methods

To get and set the field variables in these components, there are accessor methods where needed. For different types of components, some additional methods are defined according to their roles in treaty systems.

For `ActionType`, `ObjRef`, `State` and `Transition` classes, there are *get* accessors for all the field variables, but there are no *set* accessors as these values are fixed when the instances of these classes are created. There is also a special method called `isIdentical()`, which takes an instance of the same class as the parameter, and returns a boolean value identifying whether the two instances are the same. Note that the two instances that are recognized as identical by this method can be two distinct objects - just having the same variable values.

In the `Model` class, the current state has both *set* and *get* accessors, as the current state changes when valid actions are made. To add a state or

transition into a model, there are the `addState()` and `addTransition()` methods. The `transiting()` method takes a transition as the parameter, and checks if this transition is identical to one of the transitions contained in the model, and makes the transition happen if it is found to be valid. There a also a `cloneModel()` method which returns a separate copy of the current `Model` instance so that it can be used for another target object having the same behaviour model.

The `Treaty` class contains the *get* accessors for the `ObjRef` instance, and the method `performAction()` which grants or denies the requests of user accesses. It takes the reference of the target object and the action type of the access as the parameters, and returns a boolean value specifying whether this action has been granted or not. There is another method called `checkValidity()`, which also takes parameters of the target object and the action type of the access, but it will check all reachable states to find available transitions, and returns 0 if the action can be performed immediately, or 1 if the action cannot be granted now but may be valid at some point in future, or -1 if the action will never be granted.

6.2.3 Treaty Operations in the FSM Representation

The treaty combinator operations will interact with the behaviour specifications modelled in the `Model` instances. In the FSM style representation, the states and transitions will need to be changed, and for the binary operations combined states may need to be constructed which come from states in different treaties, and special transitions to connect them. To help in producing such states and transitions, there is a class called `StateCombiner` that is used for restructuring the behaviour specification.

In the `StateCombiner` class, there are two inner classes, which are called `MultilabelState` and `MLTransition`. Instead of containing a `stateID` like

the `State` class, `MultilabelState` use an array of integers such that one instance can label multiple states from different treaties. `MLTransition` does similar things as the `Transition` class, but variables `from` and `to` will have the type of `MultilabelState`. The `StateCombiner` class has five other methods in addition to the constructor, they are:

`addCombinedState()` takes an array of integers as the parameter and adds a `MultilabelState` instance into this `StateCombiner` instance.

`addTransition()` takes parameters of two arrays of integers to be the `from` and `to` states and the `ActionType` instance to be the action type that triggers this transition. It returns a boolean value to indicate whether this action succeeds.

`find()` is a private method taking an array of integers as the parameter and returning a boolean value indicating whether this combined state has already existed in this `StateCombiner` instance.

`visitAll()` is a recursive private method that sets a flag for all reachable `MultilabelState` instances. This is used for simplifying the model.

`toModel()` uses `visitAll()` to remove all unreachable states and then turning this `StateCombiner` instance to an instance of `Model` class.

With the help of `StateCombiner` class, these operations can be implemented. In the `Treaty` class there are methods of `restrictTreaty()`, `joinTreaty()`, `intersectTreaty()`, `differenceTreaty()` and `followTreaty()` that do the processing of treaty operations. We assume there are two operand treaties α and β, and the algorithms in pseudo-code of the five treaty operations are given below.

join - Two methods are related to the *join* operation. Their prototype in Java is defined by:

`public Treaty joinTreaty(Treaty other)`

`private void joinTraverse(Model m, State s, State l, State[] from, StateCombiner sc)`

The first method is used to do the *join* operation with another treaty, and the second method is used to be recursively called to support the first method. Here is the pseudo-code of these methods.

`public Treaty joinTreaty(Treaty other)`

```
1   IF (this.reference != other.reference)
2     RETURN NULL;
3   sc = new StateCombiner instance;
4   m1 = this.model;
5   m2 = other.model;
6   s1 = this.currentState;
7   s2 = other.currentState;
8   ss = a new combined state consisting s1 and s2;
9   add ss to sc;
10  joinTraverse(m1, m1.currentState, s1, ss, sc);
11  joinTraverse(m2, m2.currentState, s2, ss, sc);
12  get model m from sc;
13  RETURN new Treaty instance constructed with m;
```

`private void joinTraverse(Model m, State s, State l, State[] from, StateCombiner sc)`

```
1  v = set of all valid transitions the start from s;
2  FOR EACH transition t in v
3  create the combined state ss consisting l and t.to;
4    IF (ss not in sc)
5     add transition with (from, ss, t.actionType) into sc;
6     create the combined state sto consisting l and t.to;
7     joinTraverse(m, t.to, l, sto, sc);
8    ELSE
9     add transition with (from, ss, t.actionType) into sc;
```

follow - The prototype and the pseudo-code of the *follow* operation is given below. The `openState` is the identified state that acts as the 'reach point' state mentioned in Section 5.4.

`public Treaty followTreaty(Treaty other)`

```
1   IF (this.reference != other.reference)
2     RETURN NULL;
3   m1 = this.model;
4   m2 = other.model;
5   IF (m1.openState has not been set)
6     RETURN NULL;
7   sc = new StateCombiner instance;
8   FOR EACH state s in m1 and m2
9     add s into sc;
10  FOR EACH transition t in m1 and m2
11    add t into sc;
12  FOR EACH transitions t2 that (t2.from = m2.currentState)
13    add transition with (m1.openState, t2.to, t2.actionType)
      into sc;
14  get model m from sc;
15  create a new treaty tr constructed with m;
16  RETURN tr;
```

intersect - There are also two methods that are used for performing the *intersect* operation, and they are similar to those for the *join* operation. One method is used to be called from the holder of this treaty, and the other one is a recursive method to support the former method.

`public Treaty intersectTreaty(Treaty other)`

```
1  IF (this.reference != other.reference)
2    RETURN NULL;
3  sc = new StateCombiner instance;
4  m1 = this.model;
5  m2 = other.model;
6  add a combined state consisting m1.currentState and
   m2.currentState into sc;
7  intersectTraverse(m1, m1.currentState, m2, m2.currentState,
   sc);
8  get model m from sc;
9  RETURN new Treaty instance constructed with m;
```

`private void intersectTraverse(Model m1, State s1, Model m2, State s2, StateCombiner sc)`

```
1   v1 = set of all valid transitions that start from s1 in m1;
2   v2 = set of all valid transitions that start from s2 in m2;
3   FOR EACH transition t1 in v1
4     FOR EACH transition t2 in v2
5      IF (t1.actionType is the same as t2.actionType)
5       sf = combined state consisting t1.from and t2.from;
6       st = combined state consisting t1.to and t2.to;
7       IF (st not in sc)
8        add st into sc;
9        add transition with (sf, st, t1.actionType) into sc;
10       intersectTraverse(m1, t1.to, m2, t2.to, sc)
11      ELSE
12       add transition with (sf, st, t1.actionType) into sc;
```

difference - Again, two methods are used for performing the *difference* operation, one method is used to be called from the holder of this treaty, and the other one is a recursive method to support the former method.

```
public Treaty differenceTreaty(Treaty other)
```

```
 1  IF (this.reference != other.reference)
 2    RETURN this;
 3  sc = new StateCombiner instance;
 4  m1 = this.model;
 5  m2 = other.model;
 6  add a combined state consisting m1.currentState into sc;
 7  v1 = set of all valid transitions that start from s1 in m1;
 8  v2 = set of all valid transitions that start from s2 in m2;
 9  FOR EACH transition t1 in v1
10    FOR EACH transition t2 in v2
11     IF (t1.actionType is the same as t2.actionType)
12      same = true;
13    IF (same != true)
14     sf = combined state consisting t1.from;
15     st = combined state consisting t1.to;
16     IF (st not in sc)
17      add st into sc;
18      add transition with (sf, st, t1.actionType) into sc;
19      differenceTraverse(m1, t1.to, sc);
20     ELSE
21      add transition with (sf, st, t1.actionType) into sc;
22  get model m from sc;
23  RETURN new Treaty instance containing m;
```

```
private void differenceTraverse(Model m, State s, StateCombiner
sc)
```

```
 1  v = set of all valid transitions that start from s in m;
 2  FOR EACH transition t in v
 3    sf = combined state consisting t.from;
 4    st = combined state consisting t.to;
 5    IF (st not in sc)
 6     add st into sc;
 7     add transition with (sf, st, t.actionType) into sc;
 8     differenceTraverse(m, t.to, sc);
 9    ELSE
10     add transition with (sf, st, t.actionType) into sc;
```

restrict - Two methods for the *restrict* operation:

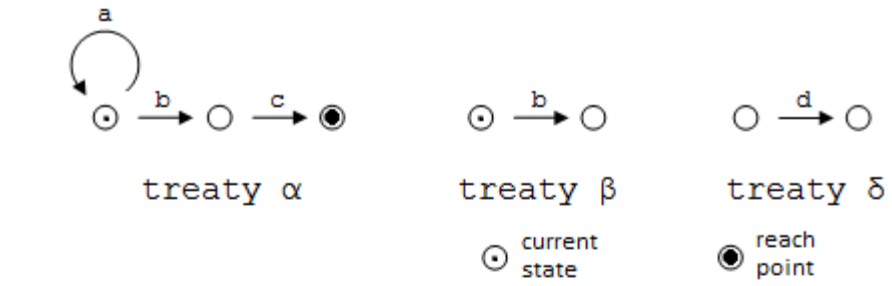
```
public Treaty restrictTreaty(ActionType at, int times)
```

```
1   IF (times < 0)
2     RETURN NULL;
3   sc = new StateCombiner instance;
4   levels = size (times+1) array of states with distinct id;
5   m = this.model;
6   add a combined state consisting levels[0] and
    m.currentState into sc;
7   FOR EACH state s in m
8     FOR 0 to (levels.length - 1)
9      add a combined state consisting levels[i] and s into sc;
10  visited = new empty collection of arrays of states;
11  restrictTraverse(m.currentState, sc, at, times, 0,
    visited);
12  get model m from sc;
13  RETURN new Treaty instance containing m;
```

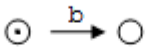
```
private void restrictTraverse(State cs, StateCombiner sc,
ActionType at, int times, int level, Vector<State[]> visited)
```

```
1   FOR EACH array of states v in visited
2     IF (v[0].id == level and v[1] is the same as cs)
3      RETURN;
4   sLevel = state with id of level;
5   thisCS = array of states consisting sLevel and cs;
6   FOR EACH transition t that (t.from = cs)
7     IF (t.actionType is the same as at)
8      IF (times > 0)
9       sNext = state with id of (level+1);
10      ss = combined state consist of sNext and t.to;
11      add transition with (thisCS, ss, at) into sc;
12      restrictTraverse(t.to, sc, at, times-1, level+1,
        visited);
13    ELSE
14     ss = combined state consist of sLevel and t.to;
15     add transition with (thisCS, ss, at) into sc;
16      restrictTraverse(t.to, sc, at, times, level, visited);
```

Fig. 6.2 and Fig. 6.3 gives a more direct view of how treaty operations in the FSM implementation are performed.

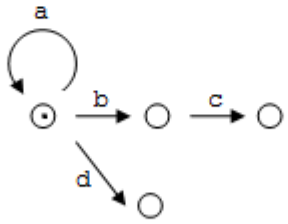


(1) α ⊓ β:



(2) α − β:

(3) α ⊔ δ:



(4) α • δ:

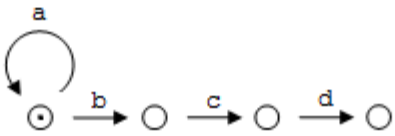


Figure 6.2: Illustration of Binary Treaty Operations

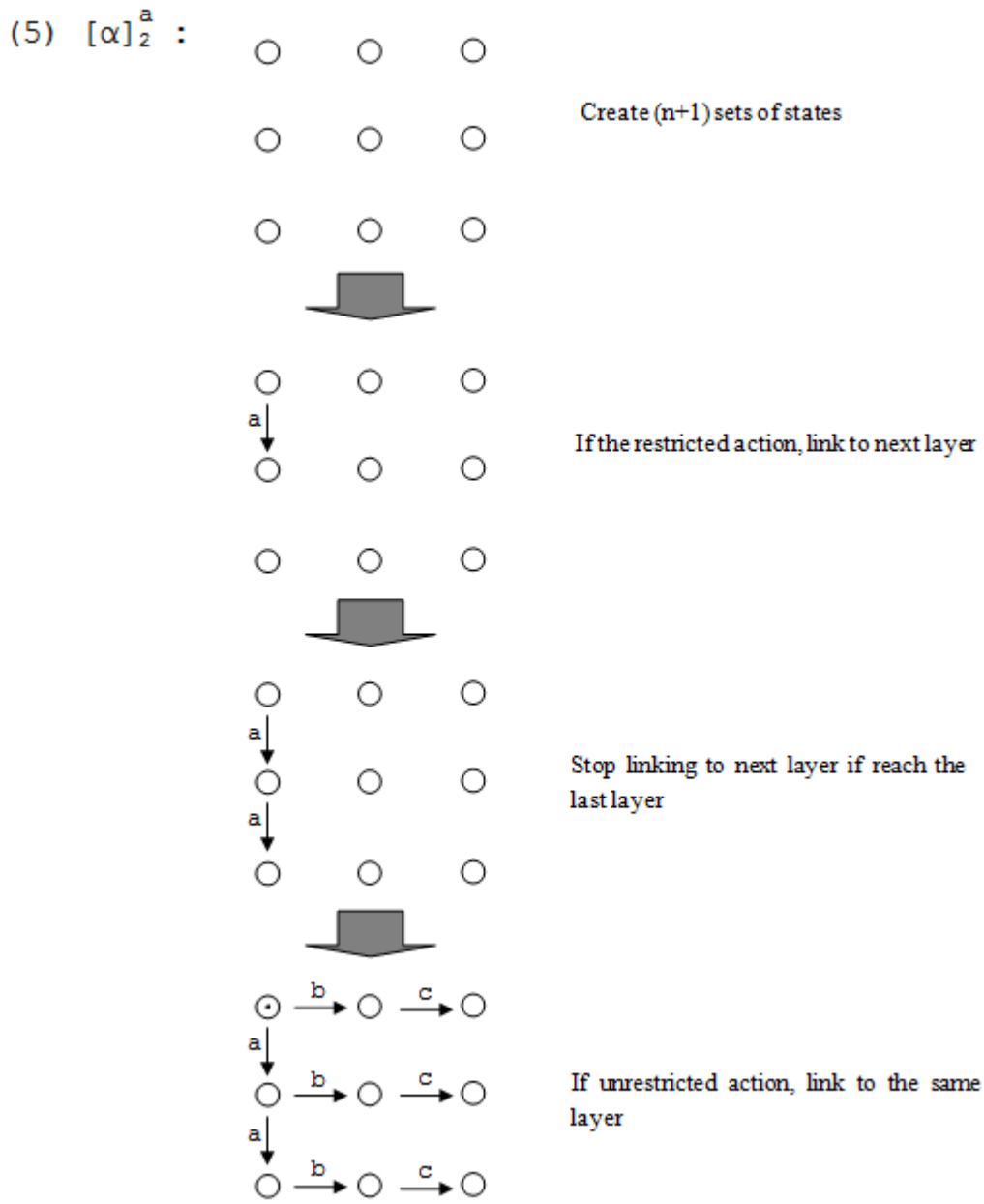


Figure 6.3: Process of the Restrict Operation

6.3 Regular Expressions as Behaviour Descriptors

In the regular expression (RE) implementation, the behaviour descriptors are represented in a different style. These no longer explicitly defined states, and the actions are represented by expressions, which can be grouped to form more complex expressions. The process of performing an action is like matching regular expressions.

There are also treaty combinator operations in the regular expression representation of treaties. Some of these operations can be realized quite straight forwardly, because of the structure of regular expressions. This is one of the advantages that the regular expression representation brings. Fig 6.4 shows the structure of classes and methods in the RE implementation in UML style.

6.3.1 Classes and Field Values

There are some components in the RE representation that are exactly the same as in the FSM representation of treaty systems. The main difference is that we use expressions to replace the `Model` class. The classes and their field values are as follows:

`ActionType` **and** `ObjRef` - same as those in the FSM representation. Used to model the type of actions and the reference to a particular object.

`Exp` - this is the class that represents expressions. However it is abstract. There is no constructor in this class and all instances of expressions are constructed by its subclasses. In class `Exp` there is the field value named `expType`, which indicates which type (defined by subclasses) this expression actually is. There are also four static final constant values

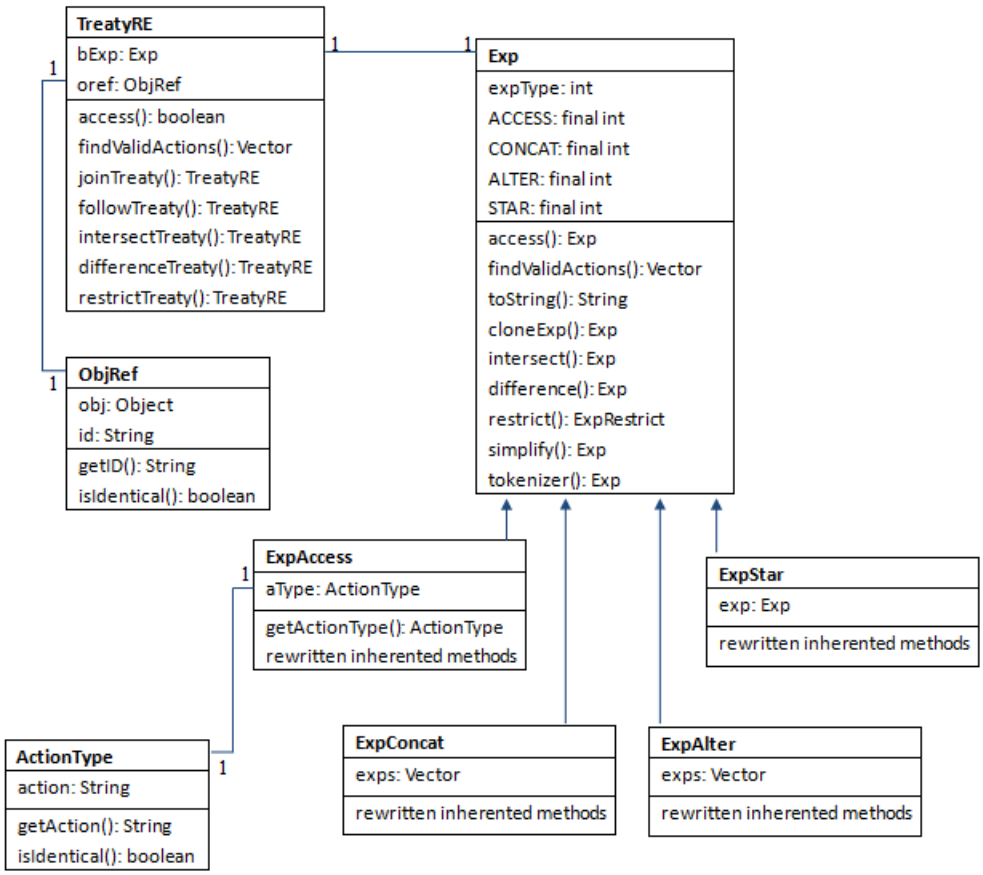


Figure 6.4: RE Treaty classes in UML form

called `ACCESS, CONCAT, ALTER` and `STAR` and they represent the types of expressions.

`ExpAccess` - a subclass of `Exp`. This class represents atomic expressions that only contain one action. The only field value is named `aType` which indicates the action type.

`ExpConcat` - a subclass of `Exp`. The instances of this class consist of a sequence of expressions. In other words these expressions are concatenated as a chain. The field value `exps` is a `Vector` instance which contains a number of expressions in sequences. These actions can only be performed consecutively.

`ExpAlter` - a subclass of `Exp`. This class also has `exps` which is a `Vector` instance of expressions as the field value, but these expressions are alternatives. One can perform any of the expressions in the vector.

`ExpStar` - a subclass of `Exp`. This class acts as the Kleene star in regular expressions. The field value `exp` is an expression for a certain type of `Exp`. Instances of `ExpStar` mean one can perform the expression zero or more times.

`TreatyRE` - the treaty in RE representation. Two field values are contained here, one is named `bExp` which is declared in the `Exp` class so that it represents the behaviour descriptor, and the other is named `oref` which is declared in the `ObjRef` class and refers to the target object.

6.3.2 RE Accessors and Access Methods

In the RE representation, fewer accessors are needed than for the FSM representation. In class `Exp` and its subclasses, there is the `getActionType()` method which returns the type of the expression instances. No *set* method

for expressions are provided as the type of an expression is fixed when it is constructed. There are no public ways to access inner expressions contained in the `Exp` instances, but they can be accessed in the scope of `protected` declaration.

The `access()` method is available for all `Exp` classes. It takes an instance of `ActionType` as the parameter, and returns an expression that results from this action. If the action is not valid, the returned expression has exactly the same Java object reference as the original expression, hence the entity that calls the `access()` method can simply use '==' comparison to know whether the access is granted or denied.

There is a `findValidActions()` method in all `Exp` classes that helps entities to know what are the available actions. This method returns a `vector` instance of `ActionType`.

Another two methods that act as key part in `Exp` classes are `simplify()` and `tokenizer()`. The `simplify()` method does simplification on the specified expression. There are simplification rules that are only relevant to this implementation of RE representation. These rules are:

1. If the inner expression in an `ExpStar` instance is also an instance of `ExpStar`, then they are combined into one, e.g. $(a^*)^* = a^*$.

2. If an inner expression in an `ExpConcat` instance is also an instance of `ExpConcat`, then the inner sequence of expressions will join the sequence of the outer one, e.g. $a.(b.c).d = a.b.c.d$.

3. If an inner expression in an `ExpAlter` instance is also an instance of `ExpAlter`, then the inner expressions will join the outer ones to be chosen together, e.g. $a|(b|c)|d = a|b|c|d$.

4. In the `ExpAlter` class, the expressions will be simplified to be deter-

ministic, e.g. $(a.b)|(a^*) = a.(b|(a^*))$.[3]

The `tokenizer()` method will read a `String` in a particular format and return an instance of `Exp`. In case the input is not valid, it returns `null`.

In the `TreatyRE` class, there are `getObjRef()` and `getExp()` methods as the accessors for the field values. There are also `access()` and `findValidActions()` methods that communicate with the expression contained in the treaty. The `access()` method in the `TreatyRE` class is different from the one in the `Exp` classes as it also takes an `ObjRef` instance to execute, and returns a boolean value to indicate the result of this access.

There is an issue that needs to be carefully considered in the implementation of RE represented treaties, which happens when there is an `ExpConcat` object which is a string of expressions (`Exp` objects), and the first expression (or first several expressions) is/are of `ExpStar` type. In normal cases when we want to get the set of valid actions at the current state, only the first expression needs to be examined, but in cases where Kleene star expressions are at the head of the concatenations, it is necessary to examine the following expressions as well, as the Kleene star expressions are allowed to 'skip' executing and pass to the next expression. For example, in an expression $(a^*).b$, the valid action at the moment is not only a, but b as well. The issue appears in cases where the `findValidActions()` or the `access()` methods are called.

Hence in the implementation when we need to get the set of valid actions, and the first several sub-expressions in the instance of `ExpConcat` is of type `ExpStar`, we also need to check the following sub-expressions, which involves the `while` loops.

[3]Note that these two expressions are not equal in the context of pure regular expressions. But this simplification is valid in RE represented treaties, as we only care about the traces of behaviours described by the behaviour descriptors. The behaviour sets provided by the two expressions are the same.

6.3.3 Treaty Operations in the RE Representation

In the RE representation of treaties, some of the operations become very easy to realize. The behaviour descriptor part of the RE representation is formed by combining expressions, and treaty operations are used for combining the behaviour specifications in treaties. Hence there are operations that can directly work on the expressions in these RE represented treaties.

join - The *join* operation works to perform a 'boolean-OR' like process. To make a *join* operation in RE representations, we can simply create an instance of `ExpAlter` from the expressions in the two original treaties.

follow - Similar to *join*, it can be simply made by creating an instance of `ExpConcat` from the expressions in the two original treaties.

intersect - The main difficulty in designing the algorithm of intersection in RE is how to deal with loops (Kleene stars). If we simply take common actions from two expressions without any solutions for loops, it may become non-terminating. In this implementation the loop issue is solved by keeping track of processed common actions and the remaining patterns, and creating `ExpStar` objects immediately when repeat patterns have been found.

```
public TreatyRE intersectTreaty(TreatyRE other)
```

```
1  IF (this.reference != other.reference)
2    RETURN NULL;
3  e1 = this.exp;
4  e2 = other.exp;
5  v1 = new empty Exp collection;
6  v2 = new empty Exp collection;
7  trace = new empty ActionType collection;
8  e = Exp instance resulted from e1.intersect(e2, v1, v2,
   trace);
9  RETURN new TreatyRE instance constructed with e;
```

public Exp intersect(Exp other, Vector<Exp> trace1, Vector<Exp> trace2, Vector<ActionType> traceA)

```
1   v = new empty ActionType collection;
2   v1 = all valid actionType at this.currentState;
3   v2 = all valid actionType at other.currentState;
4   FOR EACH actionType a1 in v1 and actionType a2 in v2
5     IF (a1 is the same as a2)
6      add a1 into v;
5   IF (v.size == 0)
6     RETURN NULL;
7   result = new empty Exp collection;
8   depth = trace1.size;
9   FOR EACH actionType a in v
10    e1 = Exp instance resulted from this.exp.access(a);
11    e2 = Exp instance resulted from other.exp.access(a);
12    IF (e1 == NULL and e2 == NULL)
13     add a new ExpAccess instance constructed with a into
    result;
14    ELSE
15     remove all elements whose index greater than depth in
    trace1, trance2 and traceA;
16     FOR i = 0 to trace1.size
17      IF (e1 is the same as trace1[i] and e2 is the same as
    trace2[i])
18       add new ExpStar instance constructed with traceA[0] to
    traceA[i] into result;
19       found = true;
20     IF (found != true)
21      add e1 into trace1;
22      add e2 into trace2;
23      add a into traceA;
24      tail = Exp instance resulted by e1.intersect(e2,
    trace1, trace2, traceA)
25      IF (tail != NULL)
26       add new ExpConcat instance constructed with new
    ExpAccess instance of a and tail into result;
27      ELSE add new ExpAccess instance constructed with a into
    result;
28  RETURN new ExpAlter instance constructed with all elements
    in result;
```

difference - The operation of *difference* between two RE represented treaties is given by subtracting any valid actions in the second operand treaty from the valid actions (and their followings) in the first operand treaty.

`public TreatyRE differenceTreaty(TreatyRE other)`

```
1   IF (this.reference != other.reference)
2     RETURN this;
3   cloned = an Exp object cloned from this.exp;
4   v1 = all valid actionType at current state of cloned;
5   v2 = all valid actionType at current state of other.exp;
6   FOR EACH actionType a1 in v1
7     FOR EACH actionType a2 in v2
8      IF (a1 is the same as a2)
9       result = this.exp.difference(a1);
10  RETURN new TreatyRE instance constructed with result;
```

`public Exp difference(ActionType at)`

```
1   SWITCH (expType of this.exp)
2     CASE (ExpAccess):
3      IF (this.exp.actionType is the same as at)
4       RETURN NULL;
5      ELSE RETURN this;
6     CASE (ExpConcat):
7      IF (this.exps[0].difference(at) != this.exps[0])
8       RETURN NULL;
9      ELSE RETURN this;
10    CASE (ExpAlter):
11     FOR EACH Exp e in this.exps
12      IF (e.difference(at) != e)
13       remove e from exps;
14     IF (elements removed from exps)
15      RETURN new ExpAlter instance containing remaining
    expressions;
16     ELSE RETURN this;
17    CASE (ExpStar):
18     diffed = this.exp.difference(at);
19     IF (diffed == this.exp)
20      RETURN this;
21     ELSE IF (diffed == NULL)
22      RETURN NULL;
23     ELSE
24      RETURN new ExpStar instance constructed with diffed;
```

restrict - The problem with implementing the restrict operation in the RE representation is that: if we apply the *restrict* to the expression with an `ExpAlter` instance followed by other expressions, and the sub-expressions in `ExpAlter` instance return different remaining number of actions, then when we keep doing the *restrict* to the following sub-expressions, we cannot decide which number to use. To solve this, there is a support class called `ExpRestrict`, which contains two `Vector` instances, one stores the expressions that were applied with the *restrict* operation, and the other stores the remaining number of actions after the respective expressions in the former vector are restricted.

```
public TreatyRE restrictTreaty(ActionType at, int times)
```

```
1  result = ExpRestrict instance from this.exp.restrict(at,
   times);
2  ea = new ExpAlter instance constructed with elements in
   result.exps;
3  RETURN new TreatyRE instance constructed with ea;
```

```
public ExpRestrict restrict(ActionType at, int times)
```

```
1   SWITCH (expType of this.exp)
2     CASE (ExpAccess):
3      es = new empty Exp collection;
4      is = new empty integer collection;
5      IF (this.actionType is the same as at)
6       IF (times > 0)
7        add this into es;
8        add (times-1) into is;
9      ELSE
10      add this into es;
11      add times into is;
12     RETURN new ExpRestrict instance with (es, is);
13    CASE (ExpConcat):
14     current = new ExpRestrict instance constructed with (at,
    times);
15     FOR EACH Exp e in this.exps
16      tempRes = new empty ExpRestrict instance;
17      FOR i = 0 to current.exps.size
18       result = e.restrict(current.exps[i],
    current.remain[i]);
```

```
19       add all pairs in result into tempRes;
20      current = tempRes;
21     RETURN current;
22    CASE (ExpAlter):
23     result = new empty ExpRestrict instance;
24     FOR EACH Exp e in this.exps
25      tempRes = e.restrict(at, times);
26      add all pairs in tempRes into result;
27     RETURN result;
28    CASE (ExpStar):
29     tempRes = this.exp.restrict(at, times);
30     result = new empty ExpRestrict instance;
31     FOR i = 0 to tempRes.exps.size
32      IF (tempRes.remain[i] == times)
33       add (new ExpStar instance with tempRes.exps[i], times)
    into result;
34     ELSE
35      next = (new ExpStar instance with
    this.exp).restrict(at, tempRes.remain[i]);
36      For j = 0 to next.exps.size
37       ea = new ExpConcat instance constructed with
    (tempRes.exps[i] and next.exps[j]);
38       add (ea, next.remain[j]) into result;
39     RETURN result;
```

It is important to note that, after the treaty operations, these newly formed expressions must be simplified before use. But it is not necessary to call the `simplify()` method each time an operation is performed. To produce a required behaviour specification it usually takes several steps of treaty operations, and the `simplify()` method can be called only once, after these operations have been done.

6.4 Other Components

To build the system for evaluation, there are several other components to be implemented. The SimJava tool was used for simulation of a distributed environment with a number of kernels and users or processes.

6.4.1 Classes and Field Values

`Kernel` - simulating kernels or middleware or servers in the system. This class extends the `Sim_entity` class in SimJava, and acts as an entity that can send and receive messages with other entities. The class includes field values of `inport` and `outports`. They are instances and collections of `Sim_port` which is the port provided by SimJava used for message communications. The `Sim_entity` class also contains a `name` field that is used to identify instances, and `Kernel` inherent this value.

`User` - simulating the users or processes who send action requests in the system. This is also a subclass of `Sim_entity`, with a `name` String as its identifier. In instances of `User`, there is only one `outport` and one `inport` of `Sim_port`, as a user will only communicate with one kernel that processes requests. When the system is running, the `inport` of a user is bound to one of the outports of its kernel, and the outport of the user is bound to the inport of that kernel. There is another field value named `treaties` of type `Vector`, and which stores all the treaties this user entity holds.

`Request` - the class that simulates the access request from a user to a kernel. The field values it contains are: a String of `name` as the identifier of the sender of the request; an instance of `ObjRef` indicating the target object of this access; an instance of `ActionType` indicating the type of action in this request; and a treaty that the user used for granting this access.

`Capability` - the class used to model the Capability approach of access control. This is used to compare with the treaty approach. There is an `ObjRef` instance indicating the target object, and a `Vector` instance named `rights` indicating the types of permissions granted by this ca-

pability instance.

`Vista` - the class used for model the Vista approach of access control. This is also used for comparison with treaties. Each instance contains a `Vector` instance storing a number of `Visibility` instances.

`Visibility` - represents the visibilities in the Vista approach. There are two field values of a `ObjRef` instance referring to the target object and a `ActionType` instance indicating the names to be accesses in the target object.

Other Classes - These classes are used for simulating objects and resources to be accessed. They depend on applications hence there are no fixed definitions of classes. But they will contain the `ObjRef` instance as the identifier of the object, and the corresponding access methods for those access types.

Here are simple examples in pseudo-code to show how kernels behave when receiving messages from users in capability, vista and treaty systems.

kernels behaving in capability systems

```
1   m = message received from user u;
2   SWITCH type of m;
3     CASE CREATE:
4      create object o according to u's requirement
       specified in m;
5      create capability c refer to o;
6      send c to u;
7     CASE ACCESS:
8      a = required action specified in m;
9      c' = capability attached in m;
10     IF a is valid by c';
11      perform the access action;
12      send result to u;
13     ELSE send denying message to u;
```

kernels behaving in vista systems

```
1   m = message received from user u;
2   SWITCH type of m;
3     CASE CREATE:
4      create object o according to u's requirement
       specified in m;
5      create vista v refer to o;
6      send v to u;
7     CASE REFINE:
8      obtain vistas v1 and v2 attached in m;
9      perform the vista operation specified in m
       to v1 and v2 to get v3;
10     send v3 to u;
11    CASE ACCESS:
12     a = required action specified in m;
13     v' = vista attached in m;
14     IF a is valid by v';
15      perform the access action;
16      send result to u;
17     ELSE send denying message to u;
```

kernels behaving in treaty systems

```
1   m = message received from user u;
2   SWITCH type of m;
3     CASE CREATE:
4      create object o according to u's requirement
       specified in m;
5      create complete treaty t refer to o;
6      send t to u;
7     CASE REFINE:
8      obtain treaties t1 (and t2) attached in m;
9      perform the treaty operation specified in m
       to t1 (and t2) to get t3;
10     send t3 to u;
11    CASE ACCESS:
12     a = required action specified in m;
13     t' = treaty attached in m;
14     IF a is valid by t';
15      perform the access action;
16      update the current state in t';
17      send result and updated t' to u;
18     ELSE send denying message to u;
```

6.4.2 Methods in Classes

There are only a few methods in these classes as we only need to simulate the relevant part when running the system, hence functionalities that exceed the scope of treaty systems are abandoned.

In the `Capability` class, there is the `getObjRef()` when capabilities are used for accessing. The `access()` method is certainly implemented to deal with the checking validity process. The other method in this class is called `reduce()` which takes an instance of `ActionType`, and removes it from the set of rights.

In the `Vista` class, there is the `addVisibility()` that adds a `Visibility` instance into this `Vista` instance. The `checkVisibility()` method checks whether a visibility is in this instance. Four methods `sumVista()`, `differenceVista()`, `intersectVista()` and `productVista()` represent the vista combinator operations.

In the `Kernel` class, there is a method named `addPort()`. This method is performed when a new user joins the kernel, and the corresponding port communicating with the newcomer is created. The `body()` method overrides the method inherented from `Sim_entity`, and this is the main body to process communicating messages and events.

The `User` class is similar to the `Kernel` class as they are both subclasses of `Sim_entity`, and there is the `body()` method for processing communications. No `addPort()` method is contained in this class as in our simple system model, we define that one user needs to connect to only one kernel, and the port has already been declared as a field value.

6.5 Summary

We have introduced the implementation of treaty systems. The implementations described were split into two main parts: one is based on the FSM representation of the behaviour descriptors in treaties, and the other is about the RE representation. The two representations have clearly different ways of describing behaviour models, hence they also affect how treaty operations act in their representations.

In the FSM representation each treaty contains one behaviour model that refers to a particular object. In the model there is a collection of states and a collection of transitions. Each transition has a label for the action type that causes this transition to happen. There is a special class called `StateCombiner` which supports the simplification of the FSM representations of the model and many treaty operations. To perform an action, the holder of the treaty needs to specify the object reference and the action type of the access together with the treaty. A successful access will cause the current state pointer in the model to change its target.

In the RE representation, we also let one treaty refer to one object, and the regular expressions that act as the behaviour descriptors are organized in a tree structure. Expressions are divided into four categories of single actions, concatenations, alternations and Kleene stars (loops), the latter three kinds contains other expressions. In this way expression trees are constructed. To perform an action, the holder of the treaty also needs to specify the object reference and the action type of the access. This shows that FSM and RE representations only differ in the structure of behaviour descriptors, and they can be used in exactly the same way. A successful action may cause the expression tree to change its shape.

The implementation of other components in the system such as kernels and users are also introduced in this chapter. With these components and

the implemented treaties we can carry out experiments to simulate the treaty systems and get evaluation results. These will be explained in the next chapter.

Chapter 7

Evaluation

Since treaties are proposed mainly for behaviour control in distributed environments, the evaluation for this project will cover the following aspects of treaty systems: the correct behaviour control ability, time for kernel(s) to deal with requests, and resources in systems that apply the treaty approach. Being an escalation of a security mechanism, the correct behaviour control ability is still essential as if it leads to fault behaving, the escalation would become useless. The time cost would be a major concern from the users' point of view as we do not want the access request to be executed slowly. The resource occupation is also one of the major concerns, from kernels, users and bandwidth's point of view. Hence the evaluations in this chapter concentrate on how time and space consumption changes when different factors change. The evaluations also show how the treaty combinator operations perform, and the comparisons between FSM represented treaties and RE represented treaties.

Most data shown in figures are average values resulted from multiple repetitions of experiments. The error bar in graphs are the standard deviation of those experiments. For data that are shown in more than one figure for different observations, only the figure that is more appropriate (may be clearer

for view or better comparison) is chosen for displaying the error bar.

7.1 Evaluation of Behaviour Control

Behaviour control is the major functionality that treaties provide, compared to access control by capabilities. Therefore, to ensure that treaties will give permissions and grant accesses correctly, especially in cases where the duplication problem may appear, it is essential to examine the correctness of accesses with treaties, within a large number of requests and user actions that may lead to the duplication problem. These user actions include using copied treaties to try to gain more permissions, and propagation of treaties among users letting multiple users hold the same treaty to request the same actions.

A system has been built to test the behaviours controlled by treaties. There are a large number of users that behave legally, and a small number of malicious users who try to perform actions that are not allowed by the system. These illegal behaviours may contain action types that do not exist in the set of valid actions for the given resource type, or repeatedly performing those actions which only allowed a limited number of times. The proportion of malicious users to overall users will be varied to examine whether any illegal behaviours are granted by the systems. Table 7.1 illustrates the result for this evaluation. We count the number of valid actions that are rejected by the kernel, the invalid actions that are accepted by the kernel and the actions which are allowed to perform only limited numbers of times but are requested more times than the limitation. The proportion of malicious users has changed from 1% to 100%, and the fault number of all three categories are 0, which means there are no fault behaviours allowed by the treaty system in our test. This experiment result does not guarantee the implemented system is faultless, but it shows that the treaty system behaves correctly and satisfies

Malicious users in the population	Valid actions being denied	Invalid actions being allowed	Limited actions being over-performed
1%	0	0	0
10%	0	0	0
50%	0	0	0
100%	0	0	0

Table 7.1: Behaviour Control Evaluation when Malicious Users Exist

the security requirements of behaviour control to some extent.

7.2 Evaluation of Time Costs

The time taken for verification is also an important criterion for evaluating treaty systems. Although treaties provide more powerful controls than simple capabilities, users will not be happy if the response time for their requests becomes too long. We need to make comparisons between the performance of treaties and capabilities, particularly on time and resource consumption. It is unavoidable that the verification time for treaties is longer than the verification time for capabilities, as it takes more steps to check permissions in treaties. Moreover, the time for verification should be significantly less than the time for accessing files, for both treaties and capabilities.

As the structures of behaviour descriptors in finite-state machine representation and regular expression representation are different, the action processing and treaty operations also have different execution methods. Hence the efficiency of the two representations is worthy of evaluation. We will build the evaluation system for FSM-style treaties and RE-style treaties under the same conditions, and calculate and compare the creation time, refinement time and execution time for these two representations. We will also compare the operating times for each type of treaty operations.

7.2.1 Access Time for Capabilities, Vistas and Treaties

In this evaluation we will test and measure the access times for all the access control components that have been introduced in this work: Capabilities, Vistas, Treaties of FSM representation and Treaties of RE representation. To keep a fair contest, all the four candidates are implemented in simple structures with only their necessary components and no additional functionalities described in Chapter 5 are applied, and they all grant the same permissions. In the implemented capabilities there are only single references to the target object and a set of action types that a capability grants. In the implemented vistas there are a set of visibilities, each of which contains one reference to the same target object and one action type for accessing the object. The simple treaties in this part of the evaluation take no account of the duplication problems, and they are only single-object referencing. The FSM represented treaties only have one state, and a number of transitions that represent those action types. The RE represented treaties contain a Kleene star expression over an alternation expression, which consists of all valid action types.

Since each single access only takes a very short time that is hard to observe, to get a clearer view on how the time consumption of using the four candidates to access objects grows, we do the test a large number of times and evaluate the overall time for each candidate. Fig. 7.1 shows the time consumption of the access time using Capabilities, Vistas, FSM represented treaties and RE represented treaties, where there is only one valid action type, and the request for access is granted (i.e. the requested action is the same as the valid action type in these access control components). Fig. 7.2 shows the time consumption of the access time using the four control components with only one action type, but this time we try an invalid action (which is different from the valid action type in the components). Finally Fig. 7.3 shows the time consumption of the access time using the four control components with ten valid action types.

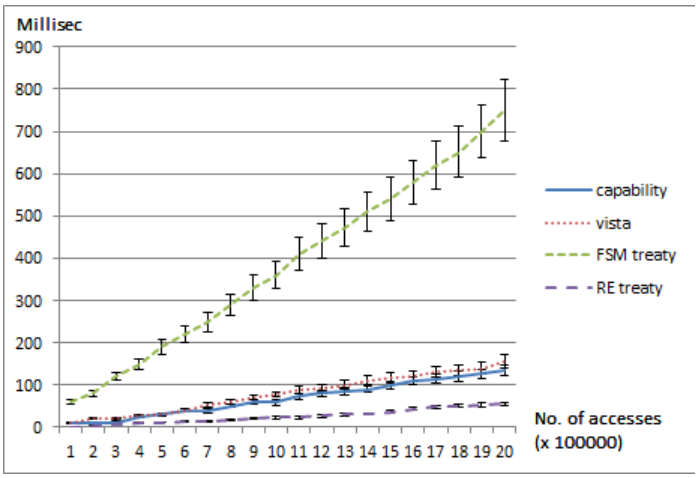


Figure 7.1: Times for successful accesses, for Capabilities, Vistas, FSM Treaties and RE Treaties with a Single Action

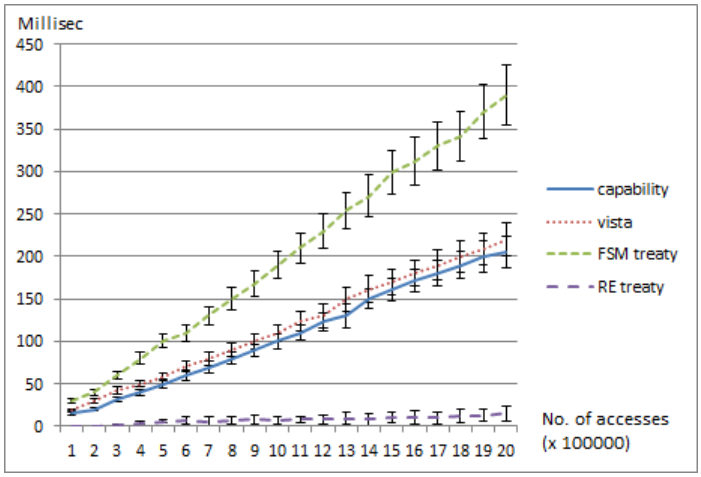


Figure 7.2: Times for failed accesses, for Capabilities, Vistas, FSM Treaties and RE Treaties with a Single Action

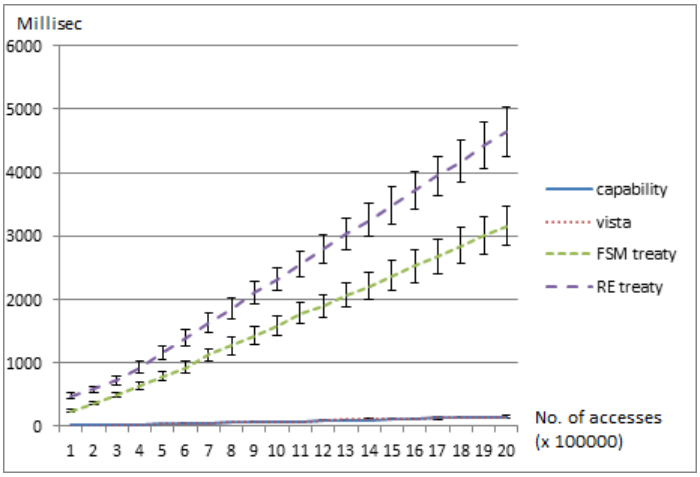


Figure 7.3: Times for successful accesses, for Capabilities, Vistas, FSM Treaties and RE Treaties with 10 Actions

From these observation we see that the time consumption for using capabilities and visibilities are almost the same, while vistas take sightly more. In cases with fewer actions, the RE representation takes even less time than capabilities to grant or deny a request, and the time for the FSM represented treaties is the most of the four. However, when the number of action types increases, the time cost of using treaties in RE representation increases significantly, and it becomes higher than the FSM representation. The capability and vista approaches have a steady time cost for accesses, where the number of action types does not have a big effect on them.

The time cost of successfully accessing using treaties is apparently higher than the time cost of failure to access by treaties. This shows that in treaty approaches, when a valid action is made, extra time is needed for changing the current state in behaviour models in treaties.

This evaluation is purely about the validation time for the candidates. It is worth mentioning that the significance of the time difference between treaties and capabilities/vistas can be much reduced if we include the prop-

agation time of messages, which means there will be another value added to the validation time, and in most cases the propagation time is much bigger than the validation, and hence makes the difference of time consumption between treaties and capabilities/vistas discountable.

7.2.2 Time Cost for Stages of Using Treaties

We examine the time consumption for each stage of using treaties in this part: creating treaties, refining treaties and accessing with treaties. Treaties are initially created to be complete treaties which contain all the types of action available to the created object, and with an unlimited number of repetitions. We call this the 'creation' stage. The 'refinement' stage is the part when creators of objects use treaty combinator operations to refine behaviour models to get new treaties. In the 'access' stage users can use treaties distributed by the creators of objects to access those objects.

We will test the time consumption of the three stages in a use-case. The use-case of creating voting treaties was selected. In this case, the kernel will originally create a complete treaty containing action types of `vote` and `check`, and refine it to fit the required specification using treaty operations (as shown in Fig. 7.4). Then a behaviour of '`check-vote-check`' will be performed using this refined treaty.

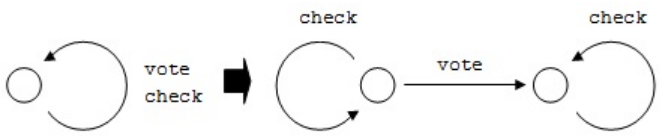


Figure 7.4: Refining a Complete Treaty

Fig. 7.5 and Fig. 7.6 show the time consumption for each of these steps in the FSM and RE representations. Since it takes a very short time to complete such a process for one treaty, we did the test for a number of times

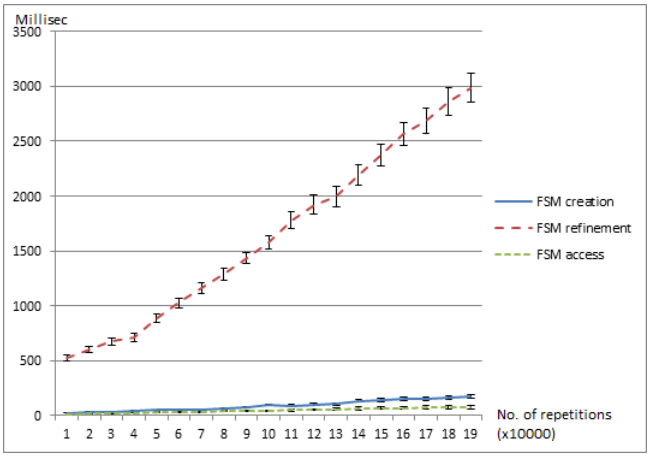


Figure 7.5: Time Consumption for Creation, Refinement and Access Stages in FSM Representation Treaties

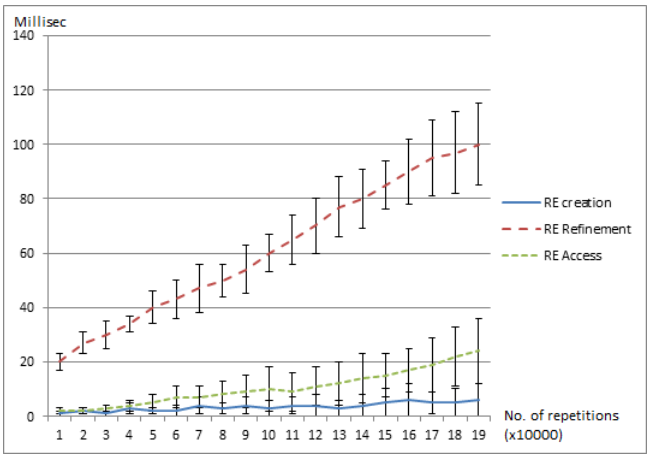


Figure 7.6: Time Consumption for Creation, Refinement and Access Stages in RE Representation Treaties

and observed the total time. The horizontal axis represents the number of processes that were performed, and the vertical axis represents the total time, in milliseconds. From the result we see that the creation and access of treaties takes similar times and is relatively low, while the refinement takes most time of the process. However it should be noted that the refinement of treaties is usually an 'infrequent' job that only happens a few times, compared with accesses which will happen frequently during a long period of time.

Fig. 7.7 shows the time consumption of these stages in both representations to give a comparison view. We can see in this figure, the time consumption in the FSM representation is much greater than the time consumption in the RE representation. The time consumption is greatly affected by the use of classes implementing the `Collection` interface in Java. In the implementation of the FSM representation, states and transitions are implemented as concrete objects as they were addressed in Section 6.2, and they are stored in collections (in our case, instances of the `Vector` class). On the other hand, in the RE representation there are no collections used in some types of expression classes. It takes much more time to access the elements in these collections using their indexes (positions in the collections) than to directly access an object with its explicit reference. This produces the difference between the performance of the FSM and RE representations.

However, `Vector` is a list-type of collection, as it stores elements as a chain of objects, and accesses them using an index. There is the alternative of using hash-type collections, which use chosen keys to access the stored elements, and hence improve the efficiency of accessing elements so decreasing the time consumption. It is possible that the time consumption of RE treaties in Fig. 7.3 would be reduced greatly and become lower then that of FSM treaties. But there are two things that need to be mentioned: first, using hash tables requires each stored element to have an identifiable key, which could mean adding extra data for elements that have no identifiable attributes;

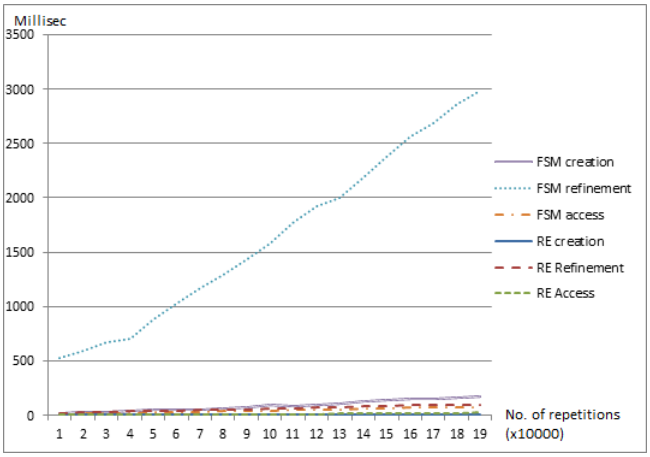


Figure 7.7: Time Consumption for Creation, Refinement and Access Stages in both Representation

second, there are places where we cannot change list-type collections to hash-type collections, e.g. the chain of sub-expressions that are connected in the relationship of concatenation in the RE representation.

7.2.3 Evaluating Treaty Combinator Operations

In this part we will evaluate the time consumption of treaty operations for both FSM and RE representations. We do the evaluation not on the simple number of run times for each operation, but instead, we change the number of actions in treaties, in other words, the evaluation is based on the complexity of the treaties' behaviour models.

Fig. 7.8 shows the time comsumption of operations in the FSM represented treaties. In each evaluation the number of actions in the behaviour model used for the test is increased by 20, and we run the operation for 20 times. From the figure it can be seen that in FSM represented treaties, the time for all the five operations does not show big differences, which means the efficiency for FSM operations is quite steady.

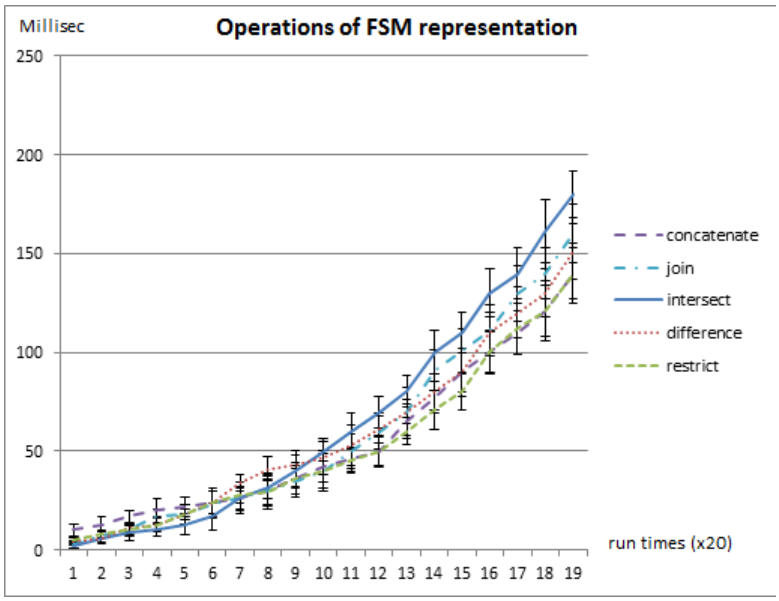


Figure 7.8: Time Consumption for Operations in FSM represented Treaties

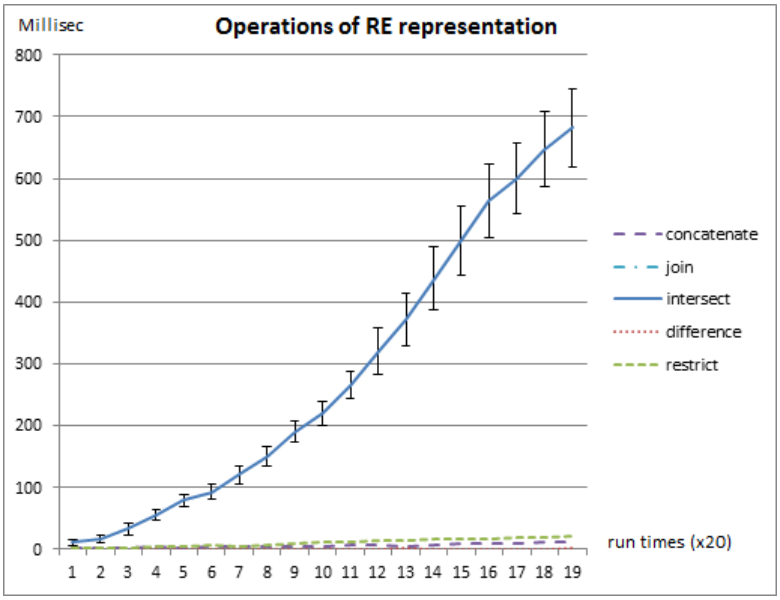


Figure 7.9: Time Consumption for Operations in RE represented Treaties

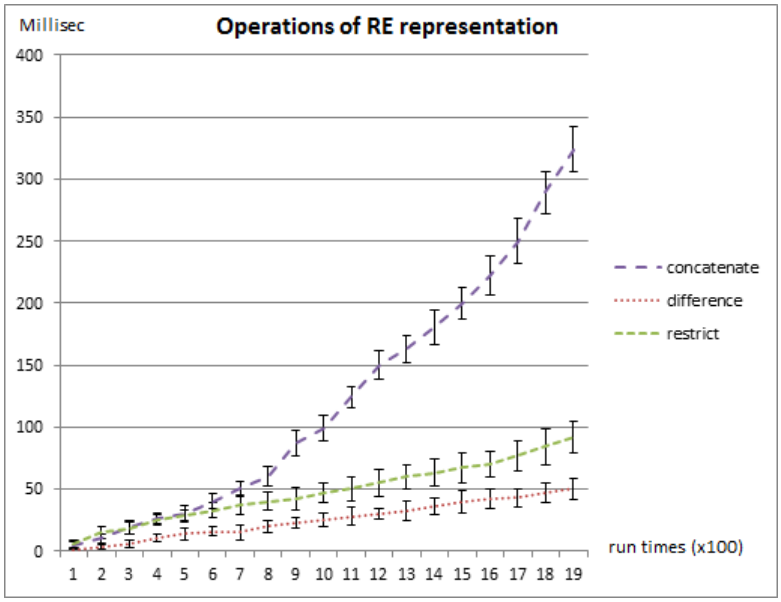


Figure 7.10: Time Consumption for Operations in RE represented Treaties

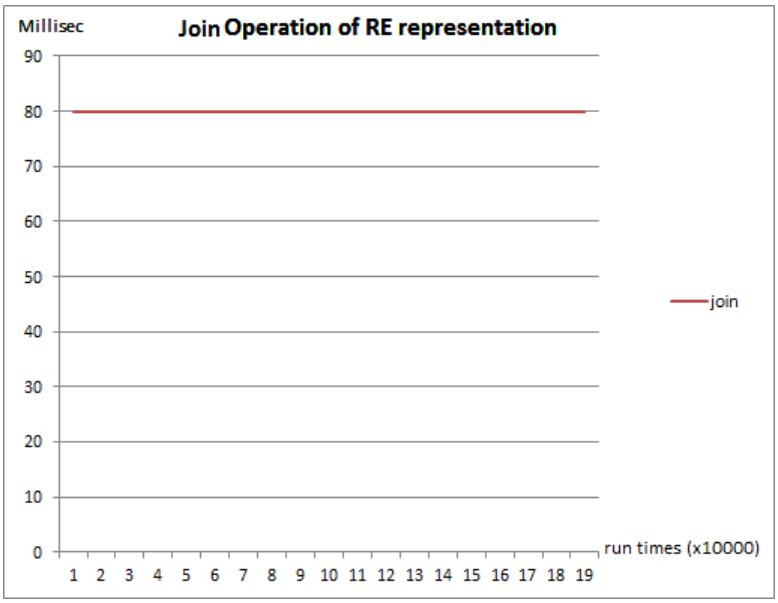


Figure 7.11: Time Consumption for *Join* in RE represented Treaties

Fig. 7.9 shows the time comsumption of operations in RE represented treaties. We use same factors as those used in the FSM evaluations, in which the number of actions is increased by 20 and 20 runs of tests are evaluated. This figure shows that the *join, follow, difference* and *restrict* operations take very little time to be performed, but the operation of *intersect* is obviously much slower. We saw in Section 6.3.3 that the algorithm for the *intersect* operation needs several collections to execute, and they are accessed frequently. Moreover, for each *intersect* operation, every sub-expression is needed to be compared with all the previous sub-expressions, all of which need to access the collections, and this gives the complexity of the algorithm nearly $O(N^2)$. This causes the *intersect* operation in the RE representation to take far more time.

Since it is to difficult to observe the trend for the *join, follow, difference* and *restrict* operations in RE representations in the resolution we used for FSM evaluations, we have to change the factor to redo the tests. Here we increase the number of actions in RE represented treaties by 100 and run the operation for 100 times in each test. Fig. 7.10 shows the time comsumption of operations *follow, difference* and *restrict*.

For the operations of *join*, we have to change the resolution again. Fig. 7.11 shows the time comsumption of operation *join*, where we increase the number of actions in RE represented treaties by 10000 and run the operation for 10000 times in each test. It shows that the time consumption for the operation of *join* in the RE representation is not affected by the complexity of the behaviour models in treaties.

Fig. 7.12 – 7.16 show the direct comparisons of the five operations between FSM and RE represented treaties using the same factor. For the operations of *join, follow, difference* and *restrict*, the RE representation performs far better than the FSM representation. The only exception is the *intersect* operation, for which the FSM representation gives a better performance.

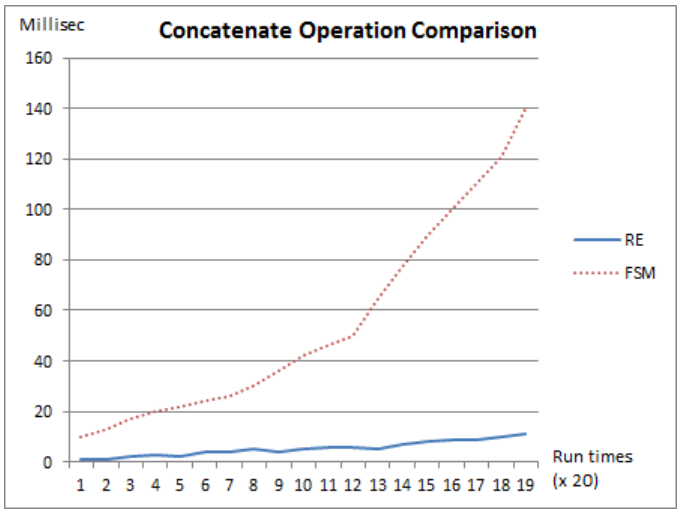


Figure 7.12: Time Consumption for Concatenate Operations

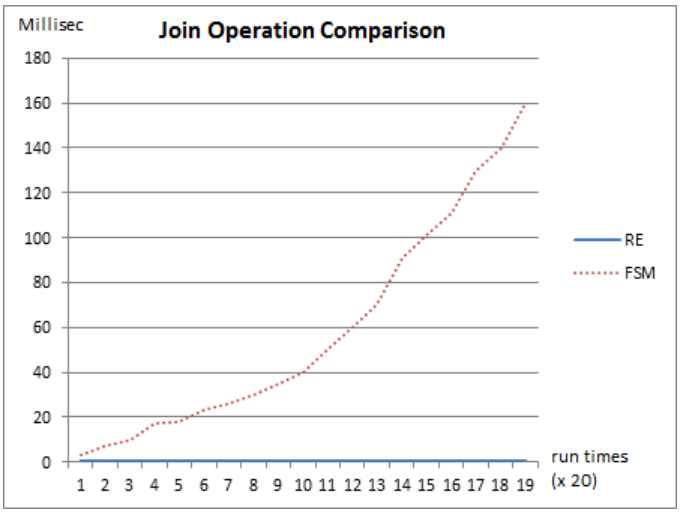


Figure 7.13: Time Consumption for Join Operations

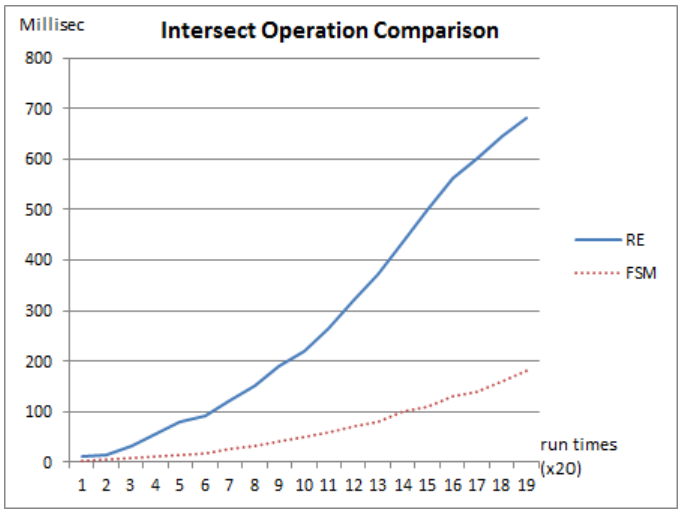


Figure 7.14: Time Consumption for Intersect Operations

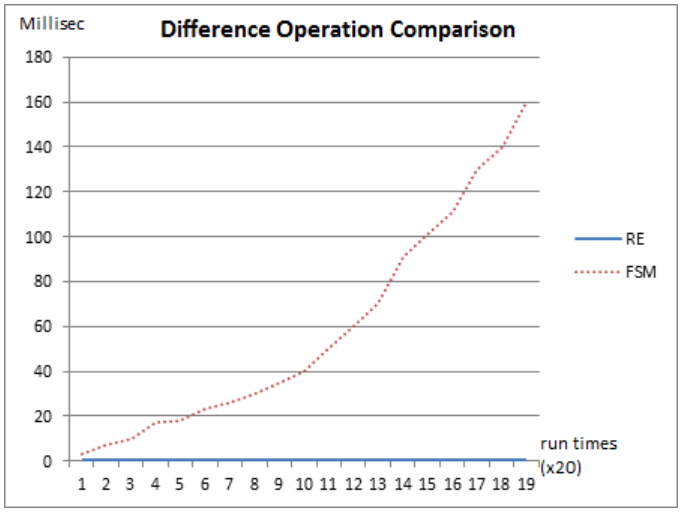


Figure 7.15: Time Consumption for Difference Operations

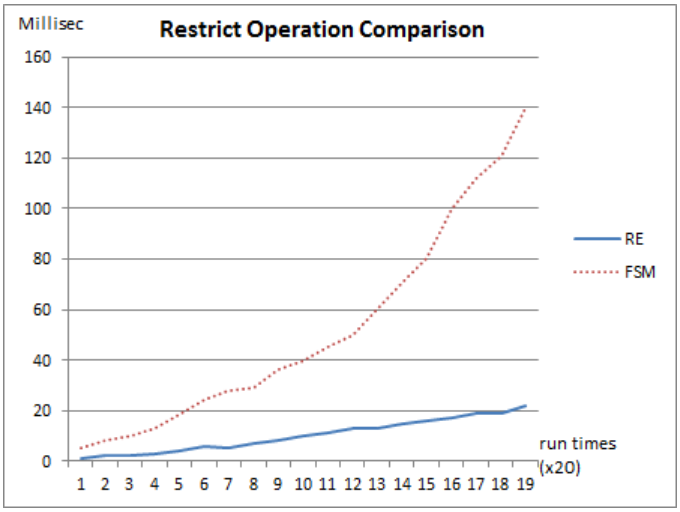


Figure 7.16: Time Consumption for Restrict Operations

Again, the collections used in the *intersect* in the RE representation are accessed more frequently than those in the FSM representation, which results in this exception. Moreover, the time consumption for the *intersect* operation of RE treaties shown in Fig. 7.14 would not be benefited from changing `Vector` collections to hash-type collections, since the algorithm of *intersect* in the RE representation relies on the sequences of actions so only list-type collections are valid.

7.3 Evaluation in a Distributed Environment

In this part of the evaluation our treaty systems are applied to a simulated distributed environment. In the evaluation, kernels and users are all entities in the system, and they can communicate with other entities using messages. With this environment we will carry out evaluations to examine the performance of models aiming to solve the duplication problem introduced in Section 4.3. We will also examine the scalability of these implementations in

the context of such environments.

7.3.1 Models towards Duplication Problem

We have proposed several models that are designed for solving the duplication problem, and they will be examined and compared for their performance with respect to time and memory consumption. The treaties and related components were implemented in Java, and they were installed in the discrete event simulation tool SimJava [35] which is also used for network and distributed environment simulation. The voting system use-case is chosen again, and the behaviour model is as shown in Fig. 7.4.

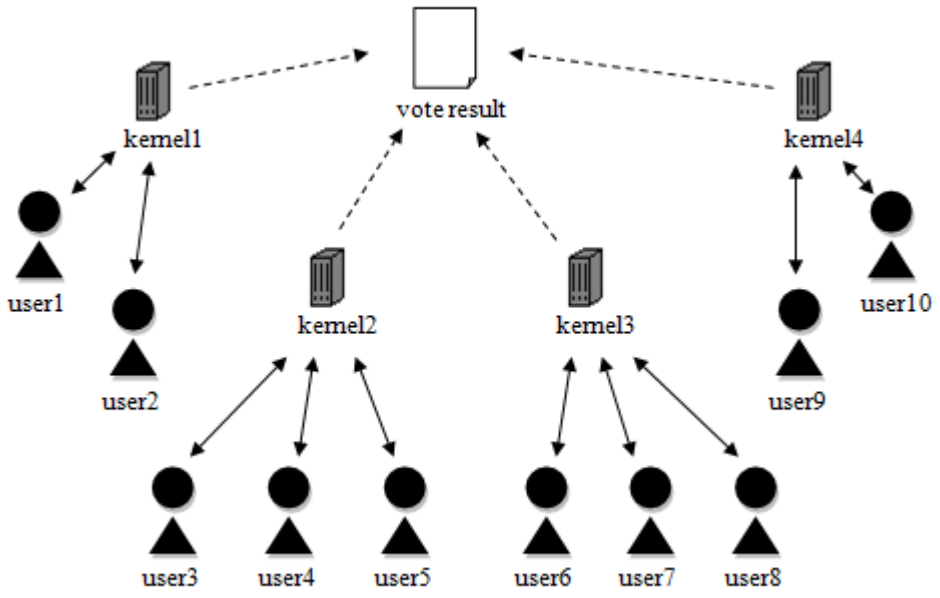


Figure 7.17: Example of Voting System Structure

An example of system structure of the evaluation is shown in Fig. 7.17. There are kernels and users whose communications include requests from

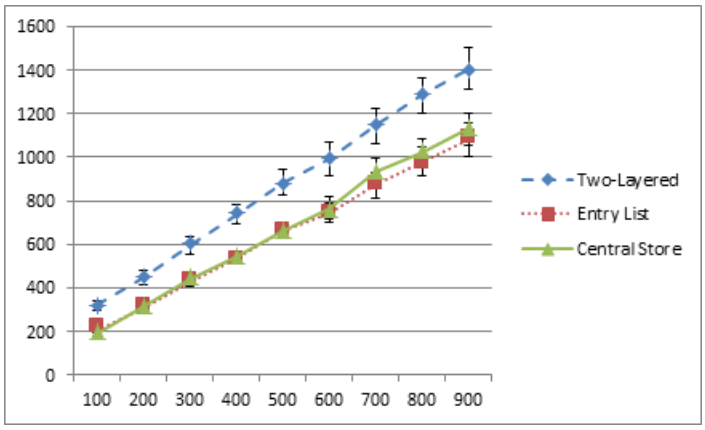


Figure 7.18: Time Consumption while Number of Users Changes

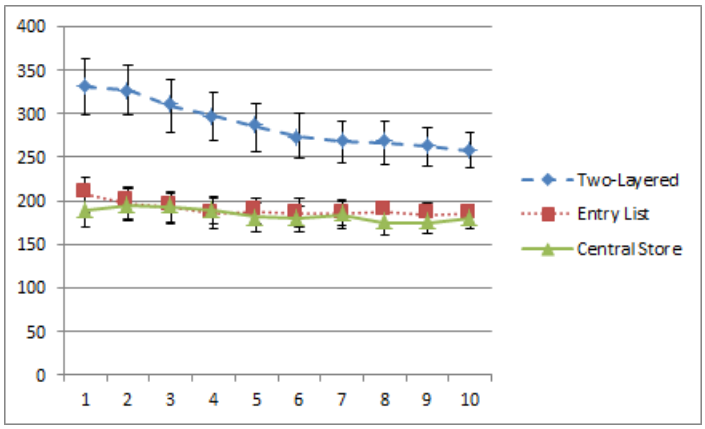


Figure 7.19: Time Consumption while Number of Kernels Changes

users and acknowledgements from kernels. There is an object counting the voting result, which can only be accessed by kernels. When processing, each user starts by sending a request for creating a treaty to the kernel. Upon reception of the treaty (or treaty reference), the user chooses to perform the `vote` action, and the kernel then changes the result in the counting object, until all requests have been made and processed. This process is repeated for models of centrally stored treaties, entry lists and Two-layered Referring FSM Representation as were proposed in Section 5.3 and 5.5, as all the three models provide solutions to the duplication problem.

Fig. 7.18 shows the time consumption when using one kernel to process different numbers of voters in our three models. The horizontal axis is the number of voters and the vertical axis is the total time consumed in milliseconds. From this it can be observed that the time spent by the central store approach and the entry list approach is almost the same, while the two-layered treaty approach takes about 40% more. Fig. 7.19 shows the time consumption when using multiple kernels to process 100 voters in the three models. The result for multiple kernels does not show a great improvement from single kernel processing, as the total amount of work processing the common voting resource by all kernels is nearly unchanged

Fig. 7.20 shows the memory consumption when having one kernel and one user, with one treaty. The vertical axis is the memory consumed in bytes. In the central store approach most of the memory consumption lies on the kernel side, while in the entry list approach the memory use is shared equally between users and kernels. The two-layered treaty approach costs more memory space than the other two. Fig. 7.21 shows the memory consumption when there are 10 kernels and 100 users with 100 treaties. We can see that in the central store approach, the kernel load is very heavy, as it still takes more memory than the sum of user space. In the entry list approach, most of the memory consumption goes with user processes. Again the two-

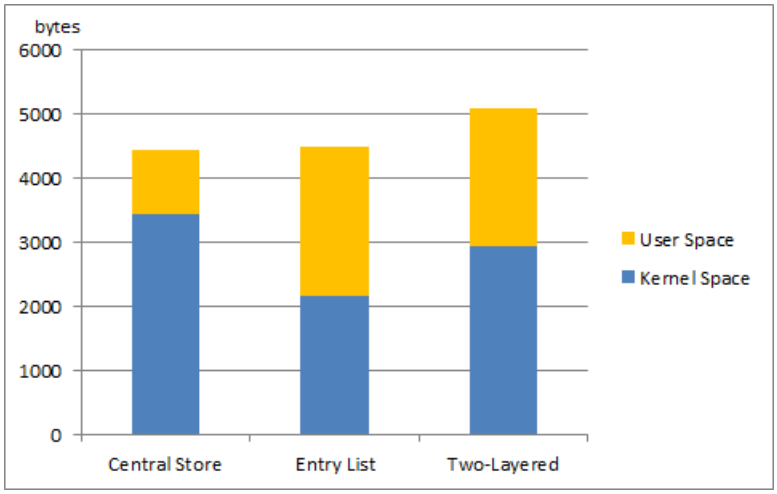


Figure 7.20: Space Consumption Comparison in Small Scaled Case

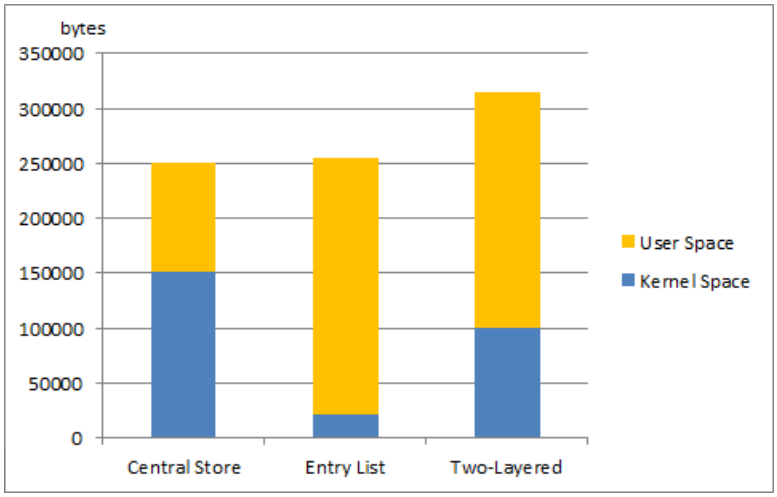


Figure 7.21: Space Consumption Comparison in Larger Scaled Case

layered treaty approach costs more memory space than the other two, but it should be noted that this approach has the extra functionality of modelling sequences among multiple objects (as discussed in Section 5.1).

7.3.2 Scalability of Treaty Systems

The scalability of treaty systems is a very important measurement in this project, as treaties are proposed to work in distributed environments where the number of users is dynamic and varies widely. However, there is a problem before any evaluation can be carried: the concept of scalability has not yet been formally defined as common knowledge. Scalability in parallel computing was originally used to describe the enhancement of processing speed when multiple processing units are involved in a computation. But as the area of parallel and distributed computing develop and applications and services in these fields keep varying, there is no single, commonly agreed way of measuring the 'scalability' for different systems.

Thus it is best to define the meaning of scalability in treaty systems to carry out any evaluations. The chosen measurement of 'scalability' shall be suitable for illustrating the performance of systems when the number of kernels and users changes in distributed environments. We define treaty system to be 'scalable' if the consumption of key measurements of the system increases linearly (or approximately linearly) while the population in the system increases linearly, which means the ratio between increased amount of consumption for resource and the increased amount of entities in the system is approximately constant.

There are two major measurements that are selected for evaluating the performance of treaty systems, they are *the additional resource on kernels* and *the processing time for requests*. The additional resource on kernels is mainly used for storing state information of treaties, as it has been explained

that there must be some information stored by kernels to solve the duplication problem. Ideally, the resource should have a linear increase with the number of behaviour models, and we can certainly call treaty systems in this situation 'scalable'. However, in practice it is always hard (or impossible) to reach the ideal case. We will need to trace the trend of increases of resources in kernels as the number of users increases (assuming each of them sets up a separate treaty).

The processing time for requests is another measurement that can reflect the performance of treaty systems in distributed environments. In this part of evaluation we would like to examine the average processing time for each request, which is calculated by having the overall processing time divided by the number of requests. We will trace changes in this average value when users and requests keep growing to evaluate the scalability on processing time. This differs from the evaluation in the previous section which compares the processing time among capabilities, vistas and treaties. In the ideal case, the average processing time for a single request should be relatively steady while the number of users changes.

Fig. 7.22 shows the average time per message when users in the system increase. We still do the comparison among the three models that aim to solve the duplication problem. The messages that are counted in this experiment includes the request from users to kernels for obtaining a treaty, the request from users to kernels for accessing the object using the treaty, and the acknowledgment message from kernels to users that reply to the former two kinds of requests. We can see from the figure that as the experiments go, there are fewer factors that disturb the evaluation result, so the average time consumption for processing each message becomes steady in later stages. This shows that when the number of users increases, the processing time for requests is not much affected, which implies good scalability in time consumption for our proposed models.

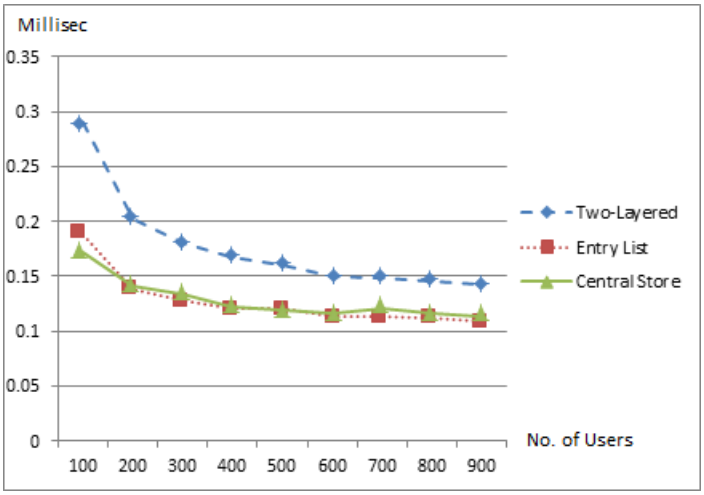


Figure 7.22: Average Time Consumption per Message

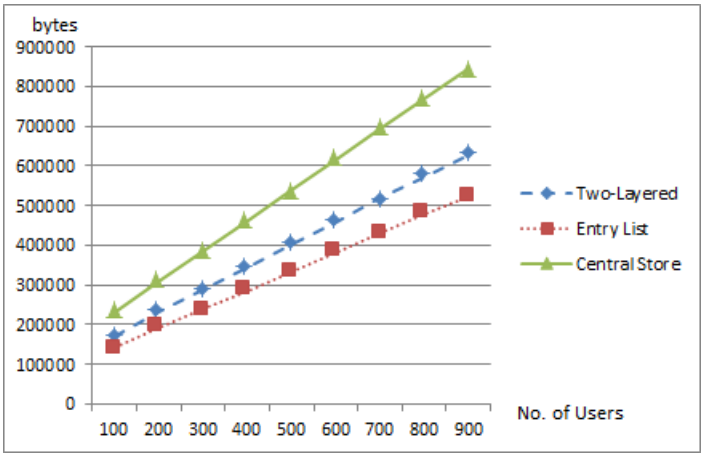


Figure 7.23: Kernel Memory Occupation

Fig. 7.23 shows the kernel space occupation when the number of users in the system increases. From the result we can see that all the three models have a linear increase in the space occupation on the kernel side. This also shows that treaty systems have a good scalability in additional resource consumption on kernels.

Note that in these approaches, the entry list approach has the lowest central resource occupation among the three, which seems to be the better choice in the context of distributed environments. But the central memory occupation is not the only consideration in such environments, and there is the other factor of bandwidth consumption. For the network load, the more the data transfered, the heavier the bandwidth consumption would be. For the three approaches, the centrally-stored treaty method would only require the entities wanting to access resources to send treaty references, which are relatively small in size. The entry list approach and the two-layered FSM treaty approach both need the treaties to be sent, hence they have much higher bandwidth consumptions. Generally speaking, the developers and service providers need to balance their trade-offs. If the application is more critical in server load, it is advised to choose the entry list approach; if the application requires lower bandwidth occupation, the centrally stored approach seems to be the better choice.

7.4 Summary

In this chapter several tests have been made and illustrated. We have shown how effectively treaties can control the behaviours by denying invalid requests from malicious users. The validation of candidates of access control and behaviour control approaches are evaluated by their validating time. Evaluations of the time consumptions for different stages when using treaties has also been given, and it was shown that the refinement stage consumes

more time than the creation of complete treaties and using treaties to access objects. However, refinement of treaties, unlike accesses, does not happen frequently.

We also made evaluations on treaty operations when the treaties have varied their number of actions. Some interesting results showed that the performance of operations in the FSM representation are almost the same, while those of the RE representations perform much better most of the time, except for the case of the *intersect* operation. The *join* operation in the RE representation is approximately constant time as it is not affected by the number of actions in the operand treaties.

These results also show that the time consumption of these implementations is greatly affected by the number and frequency of user of the collection data structures. To improve the performance of systems it is suggested to minimize using collections when possible.

The models for solving the duplication problem that were introduced in Chapter 5 were also tested in distributed fashion. The number of users and kernels in the system were varied and the evaluation of the performance for the three models demonstrated the differences among those models. It shows that the entry list representation has the lowest kernel load. Meanwhile these models are also used to evaluate the scalability of treaty system. According to the result, treaty systems show a good scalability in both time and space consumption. The overall result illustrates that treaties are feasible to be applied in the control of distributed environments.

All parts of the research have been shown in the thesis. Now we can have an overall view and point out any potential topics that could be further developed in the treaty systems in the next chapter.

Chapter 8

Conclusion

Access control is an essential concept in the security administration process. It guarantees the resources are accessed by the correct entities. Traditional access control approaches use Access Control Lists that provide centralized control, as it stores the access information with objects. However this approach is not very applicable in distributed environments, where the population of users changes frequently, and it is expensive to maintain such dynamic user lists. Capabilities provide another solution to dynamic access control. This is achieved by having capabilities as concrete components which are held by users. Users can then pass these capabilities to other users to distribute the permission for accessing objects. The kernels no longer need to store the access control information, but only need to check the validities of capabilities when they are presented by users. Hence capabilities are more suitable in distributed environments.

But capabilities are still quite simple in their structure. There are only sets of rights that grant unlimited repetitions of valid actions — there is no way of specifying subsequences of actions. In the real world there are cases where the actions must happen in particular orders, and there are cases where some actions only have a limited number of repetitions. We

call these *behaviours*. Thus, it is beneficial to investigate behavioural control approaches in distributed environments. We propose the ideas of *Vistas* and *Treaties* in this thesis which provide the behaviour control.

8.1 The Proposed Work and Contributions

Vistas are extended from the idea of capabilities. This approach is suggested because the complexity of objects in modern applications has evolved, and there may be a variety of field values and methods inside an object. In vistas there are sets of *visibilities*, each of which is a reference to the targeted object, together with the permission to access one of the field values or methods in the object. This means in a vista, there might be visibilities that refer to different objects. An important functionality that is provided in the Vista approach is the vista combinator operations that can be used to change the set of visibilities contained. In the content of this thesis, the *product* is a particularly important vista operation, since it produces a chain of actions, which provides an aspect of behaviour specifications. However there are more aspects that the idea of *behaviour* carries and vistas cannot embody all of these. Thus further development is needed, which leads to the idea of treaties.

Treaties are also an extension of capabilities and vistas, which are even better in modelling behaviour specifications. In treaties there are behaviour descriptors that can record the states of the defined behaviour model, and a pointer to the current states. The current state changes when actions are successfully performed using a treaty. There are also combinator operations in treaty systems to refine behaviour specifications while these treaties are being held by users, and this makes treaties a dynamic behaviour control approach that is suitable for distributed environments. However it needs to be kept in mind that there is a rule in treaty systems for those treaty

operations, which is: an entity cannot increase the totality of behaviours in the treaties it holds. The rule must be kept by all treaty operations. We have proven our proposed basic operations will not break this rule using the concept of behaviour set.

A new issue has been brought about by the new aspect of states in treaties, and that is called the *duplication problem*. In treaties there might be actions with a limited number of repetitions, and if there are no security mechanisms provided, to duplicate such a treaty could increase the number of allowed actions. The duplication problem may be caused by copying of treaties, passing treaties among users and doing a *concatenate* operation of a treaty to itself. Any systems that implement treaties must provide solutions to the duplication problem to avoid this issue. A number of models that aim to solve the duplication problem have been proposed in the thesis.

In treaties and treaty systems there are several parts that are not fixed, and implementation options are provided to developers and service providers such that they can choose to create a treaty system that best fits their requirements. These parts include: object referencing style, the way to represent behaviour descriptors, the location for storing treaties, additional treaty operations, and ways for executing commands. A complete example of a treaty system that has every part fixed has been presented in the thesis.

Treaties and relevant components in treaty systems has been implemented in Java, including methods for treaty combinator operations. The implemented systems were used to evaluate the performance. We compared the time consumption for accessing resources with various access control components, and tested the time consumption of different components of these implementations. The performance of treaty operations in the FSM and RE representations are compared to show the differences in the two representations. We also proposed three different models that aim to solve the duplication problem, and evaluated their time and space consumptions. They were

also used to evaluate the scalability of the implemented treaty systems. The result showed that these treaty systems scaled well, and hence are suitable for distributed environments.

The main contributions of this thesis are:

- Developing the concept of vistas and treaties, how they are structurally evolved from capabilities, and why these evolutions are beneficial.
- Investigating operations that could be used to combine vistas or treaties to form new access control and behaviour control specifications. These operations provide the functionality of letting users refine their behaviour specification after the vistas or treaties are produced, and hence further increase the dynamic control of accesses and behaviours.
- Providing a proof of the correctness of treaty operations using the idea of behaviour set to show that refining treaties with treaty operations will not destroy the safety of behaviour control.
- Pointing out the new issue called the *duplication problem* that arises with the behaviour control approaches, and suggesting possible solutions that could solve this problem. The performance of the candidates of solutions for the duplication problem were also tested and compared.
- Demonstrating that treaty implementations can have various options that developers and service providers can choose for treaties and treaty systems to satisfy their particular requirements.
- Presenting implementations of vistas and treaties. We realized the functionality of behaviour controls and the refinement of behaviour specifications using treaty operations, have evaluated the correctness of behaviour control using treaties, and have evaluated the time consumption for the components of treaty implementations. Finally we

evaluated the scalability of treaty systems to show the practicality of treaties.

8.2 Future Work

Using treaties to provide behaviour control in distributed systems is a novel approach, and there are still many areas that can benefit from further research. Here we will introduce some of these areas.

8.2.1 Behaviour Pattern Matching

In Section 4.2 we have shown the treaty combinator operation *restrict*, which can produce new behaviour models in which the occurrence of the given type of action has been limited to a certain number. This is useful when cases need to have some actions restricted in repetitions, but at this stage, it only concerns actions. From time to time we can find situations where it is not a single action that needs to be restricted, but a sequence of actions. For instance, one is allowed to smoke, and to go to the petrol station, but it is never allowed to go to the petrol station and smoke. This indicates that we may need to restrict a pattern of behaviours instead of single actions. The same can be applied to other treaty operations, such as observing the common pattern of behaviours in two treaties (the *intersect* operation), or eliminating a certain pattern of behaviour from the treaty (the *difference* operation). All these need pattern matching mechanisms.

It is difficult to perform a pattern matching in FSM represented treaties, and not an easy task for RE representations either (there are various works towards RE matching [64, 34, 4], but keep in mind that in our context it also includes sub-expression matching). But it is still worth applying some effort to finding a pattern matching mechanism for treaties with good efficiency,

due to the reasons addressed in the previous paragraph.

8.2.2 Reducing Parameters of Target Object

The efficiency of using capabilities and treaties in programming languages is also an issue that could be explored. In a treaty based system, to access any object there must be one or more attached treaties as parameters. Assuming an object-oriented language, a possible expression for accessing an object (this access can be a method to read or write the field values of the object) could be:

$$o.m(p, t);$$

where o stands for the object (in fact it is the reference to this object), m is a method or function, p some parameters and t is the treaty used to access o. Assume t is a single-object reference treaty (recall the structure of treaties: they consist of references to objects and the behaviour descriptors). Therefore we can see that in the line of code above, the reference to the object o occurs twice (as o, and in t), which causes a redundancy since the duplication of reference is unnecessary. Thus, a more elegant line of code representing the same meaning could be:

$$t.m(p);$$

There still exists other duplication in the line of code. The method m, assuming it is an operation like `read` or `write`, is the action which the holder of treaty t thought to be valid. That is, the method m is already contained in the treaty t, which means m also appears twice. At the moment it is not fair to say this is also a redundancy, since there may be more access types than m in the valid actions in t, and an indication of which action to be chosen is needed. But, can this duplication be simplified as well? The answer might

be yes, but modifications might be needed to the structure of treaties.

8.2.3 Non-deterministic Problem and Semaphore-like Scheme

As mentioned in Section 5.3, in treaty systems the non-determinism problem also appears. Since there might be multiple users requiring to perform actions on the same object, and in treaties there are sequences of actions, it is possible that the access control information being held by users is outdated. Assuming the treaty α allows its holder to do actions `a` then `b` to its targeted object o, and copies of α are held by users u_1 and u_2. u_1 believes he can do `a` to o and sends this request to the kernel, but the kernel denies his request, because u_2 had already done the `a` action to o. The kernel then tells u_1 he can do only `b` now. However, even this acknowledgment from the kernel is possibly outdated. When u_1 then asks for the action of `b`, the kernel may still reject it, because after the former acknowledgment from the kernel and before the second request from u_1, u_2 has made the request of `b` in front once again. Not providing solutions to this problem may cause a great number of invalid request and the denying acknowledgment messages, which could take up significant system resources.

We may possibly improve the situation by adapting the semaphore ideas [18, 19]. The binary semaphore can be simply applied in treaty systems by flagging the resource to be 'occupied' while a user want to access it. Taking the example in the previous paragraph, before u_1 makes the actual request of action, he can first asking the kernel 'what can I do to o'. The kernel then flags object o, and tells u_1 that the action `b` is available right now. Then u_1 can either make this valid action or waive this occupation of the resource. Whatever the choice he makes, the kernel will release the flag for o after the communication with u_1 comes to an end.

This scheme, however, faces problems that ordinary semaphores have, such as users occupying resources without releasing them. Thus further research needs to be done to find a satisfiable solution.

8.3 Coda

The vista and treaty ideas have been proposed in the thesis. They evolved step by step, aiming to provide finer descriptions of actions and better security control in distributed environments. But behaviour controls in such environments is still a topic that is deep enough for other approaches to fill. The effort illustrated in this thesis takes a small step in this direction, but has sketched an initial picture to be worked with.

Bibliography

[1] J.A. Anderson, J.L. Lewis, O.D. Saylor, Z Yu, and M Lu. *Discrete mathematics with combinatorics*. New Jersey: Prentice Hall, 2001.

[2] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. Wiley-Interscience, 2004.

[3] J.C.M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335:131–146, 2005.

[4] Sebastian Bala. Regular language matching and other decidable cases of the satisfiability problem for constraints between regular open terms. In *Proceedings of 21st Annual Symposium on Theoretical Aspects of Computer Science*, pages 596–607, 3 2004.

[5] D. E. Bell and L. J. LaPadula. Computer security model: Unified exposition and multics interpretation. Technical report, MITRE Corp., Bedford, MA, 1975.

[6] Martin Berglund, Henrik Bjorklund, and Johanna Hogberg. Recognizing shuffled languages. In *LATA2011 - Language and Automata Theory and Applications, LNCS 6638*, pages 142–154. Springer, 2011.

[7] J.A. Bergstra and J.W. Klop. Fixed point semantics in process algebra. Technical report, Mathematical Centre Amsterdam, 1982.

[8] Robert Bjornson, Nicholas Carriero, and David Gelernter. From weaving threads to untangling the web: A view of coordination from Linda's perspective. In *LNCS 1282 Coordination Languages and Models, COORDINATION '97*, pages 1–17, 1997.

[9] W.E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *7th DoD/NBS Computer Security Conference*, pages 291–293, 9 1984.

[10] Eike Born and Helmut Stiegler. Discretionary access control by means of usage conditions. *Computers & Security*, 13(5):437–450, 1994.

[11] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the Association for Computing Machinery*, 30(2):323–342, 4 1983.

[12] Michael Burrows, Martin Abadi, and Roger Michael Needham. A logic of authentication. In *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, pages 233–271, 1989.

[13] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. *SIGPLAN Notices*, 43(9):213–224, 2008.

[14] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 5 1978.

[15] D.J. Cooke and H.E. Bez. *Computer Mathematics*. Cambridge University Press, 1985.

[16] Jim Davies. Using CSP. *Refinement Techniques in Software Engineering, Lecture Notes in Computer Science Volume 3167*, pages 64–122, 2006.

[17] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 3 1966.

[18] Edsger W Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[19] Edsger W Dijkstra. Cooperating sequential processes. *The origin of concurrent programming*, pages 65–138, 2002.

[20] Samuel Eilenberg. *Automata, Languages, and Machines*. New York: Academic Press, 1974.

[21] DF Ferraiolo, R Sandhu, S Gavrila, and DR Kuhn. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4:224–274, 8 2001.

[22] Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

[23] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1 1985.

[24] David Gelernter and Lenore Zuck. On what Linda is: Formal description of Linda as a reactive system. In *LNCS 1282 Coordination Languages and Models, COORDINATION '97*, pages 187–204, 1997.

[25] David K. Gifford. Cryptographic sealing for information secrecy and authentication. *Communications of the ACM*, 25(4):274–286, 4 1982.

[26] Jay Gischer. Shuffle languages, Petri nets, and context-sensitive grammars. *Communications of the ACM*, 24(9):591–605, 1981.

[27] V.D. Gligor. Review and revocation of access privileges distributed through capabilities. *IEEE Transactions on Software Engineering*, SE-5:575–586, 1979.

[28] Li Gong. A secure identity-based capability system. In *Proceedings of 1989 IEEE Symposium on Security and Privacy*, pages 56–63, 1989.

[29] Daniele Gorla and Rosario Pugliese. Dynamic management of capabilities in a network aware coordination language. *Journal of Logic and Algebraic Programming*, 78(8):665 – 689, 2009.

[30] Philipp Haller and Martin Odersky. Capabilities for external uniqueness. Technical report, EPFL.

[31] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. In *Theoretical Computer Science*, 2008.

[32] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[33] Kohei Honda. Types for dyadic interaction. In *CONCUR'93: 4th International Conference on Concurrency Theory, LNCS715*, pages 509–523, 8 1993.

[34] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for XML. In *Proceedings of POPL '01 Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 67–80, 3 2001.

[35] F Howell and R McNab. SimJava: A discrete event simulation library for java. *Simulation Series*, 1998.

[36] Thomas J. Jech. *Set theory*. Academic Press, INC., 1978.

[37] Joanna Jedrzejowicz and Andrzej Szepietowski. Shuffle languages are in P. In *in P. Theoretical Computer Science*, 2001.

[38] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. *IEEE Transactions on Software Engineering*, SE13(2), 2 1987.

[39] P.A. Karger. Non-discretionary access control for decentralized computing systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1977.

[40] Paul A. Karger and Andrew J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 2–12, 1984.

[41] P.A. Karker. New methods for immediate revocation. In *Proceedings of 1989 IEEE Symposium onSecurity and Privacy*, pages 48–55, 5 1989.

[42] A. H. Karp, G. J. Rozas, A. Banerj, and R. Guptai. Using Split Capabilities for access control. *IEEE Software*, 20(1):42–49, 1 2003.

[43] K.Takeuchi, K.Honda, and M.Kudo. An interaction-based language and its typing system. In *PARLE'94: 6th International PARLE Conference, LNCS817*, pages 398–413, 7 1994.

[44] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 10 1973.

[45] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press, 1984.

[46] Theodore A. Linden. Operating System Structures to Support Security and Reliable Software. *ACM Comput. Surv.*, 8(4):409–445, 1976.

[47] Mark Miller, Ka-Ping Yee, Jonathan Shapiro, and Combex Inc. Capability myths demolished. Technical report, Systems Research Laboratory, Johns Hopkins University, 2003.

[48] Mark S. Miller and Jonathan Shapiro. Paradigm Regained: Abstraction Mechanisms for Access Control. In *8th Asian Computing Science Conference (ASIAN03)*, pages 224–242, 2003.

[49] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[50] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and computation*, 1992.

[51] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[52] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21:993–999, 1978.

[53] MO Rabin. *Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.

[54] Matthew Sackman and Susan Eisenbach. Session type in Haskell. 2008.

[55] RS Sandhu, EJ Coyne, HL Feinstein, and CE Youman. Role-based access control models. *Computer*, 29:38–47, 2 1996.

[56] RS Sandhu and P Samarati. Access control: principle and practice. *IEEE Communications Magazine*, 32:40–48, 9 1994.

[57] Martin Scheffler, Jan P. Springer, and Bernd Froehlich. Object-capability security in virtual environment. In *IEEE Virtual Reality 2008*, 2008.

[58] S.Gay, V.T.Vasconcelos, and A.Ravara. Session types for inter-process communication. Technical report, Department of Computing, University of Glasgow, 3 2003.

[59] J Shapiro and S Weber. Verifying the EROS confinement mechanism. In *2000 IEEE Symposium on Security and Privacy*, pages 166–176, 2000.

[60] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[61] Jonathan Strauss Shapiro. *EROS: a capability system.* PhD thesis, University of Pennsylvania, 1999.

[62] J.S. Shapiro and N. Hardy. EROS: a principle-driven operating system from the ground up. *IEEE Software*, 19:26–33, 1966.

[63] HH Shen and P Dewan. Access control for collaborative environments. In *CSCW '92 Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 51–58, 1992.

[64] R. Sidhu and V.K. Prasanna. Fast regular expression matching using FPGAs. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 227–238, 4 2001.

[65] N. I Udzir. *Capability-Based Coordination For Open Distributed Systems.* PhD thesis, University of York - Department of Computer Science, 2007.

[66] Roel van der Goot, Jonathan Schaeffer, and Gregory V. Wilson. Safer tuple spaces. In *LNCS 1282 Coordination Languages and Models, COORDINATION '97*, pages 289–301, 1997.

[67] Jan Vitek, Ciarán Bryce, and Manuel Oriol. Coordinating processes with secure spaces. *Sci. Comput. Program.*, 46(1-2):163–193, 2003.

[68] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. *A Note on Distributed Computing*. Springer Berlin Heidelberg, 1997.

[69] David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 2000.

[70] Raymond M. Wong. A comparison of secure UNIX operating systems. In *Computer Security Applications Conference, 1990., Proceedings of the Sixth Annual*, pages 322–333, 12 1990.

[71] Alan Wood. Coordination with attributes. In *LNCS 1594 Coordination Languages and Models, COORDINATION '99*, pages 21–36, 1999.

[72] Alan Wood and Yining Zhao. Vistas: Towards behavioural cloud control. In *Euro-Par 2010 Parallel Processing Workshops, LNCS 6586*, pages 689–696. Springer, 2011.

[73] Edmund Woronowicz. Relations and their basic properties. *Formalized Mathematics*, 1(1):73–88, 1990.

[74] Edmund Woronowicz. Relations defined on sets. *Formalized Mathematics*, 1(1):181–186, 1990.

[75] Sheng Yu. Regular languages. *Handbook of Formal Languages*, 1:41–110, 1997.

[76] Yining Zhao and Alan Wood. Treaties: Behaviour-controlling capabilities. In *2nd Annual International Conference on Advances in Distributed and Parallel Computing (ADPC 2011)*, pages 19–24. GSTF, 2011.

[77] Yining Zhao and Alan Wood. Behavioural sets and operations in treaty systems. In *Proc. International Conference on Control Engineering and Communication Technology*, 2012.

[78] Yining Zhao and Alan Wood. Models to solve the duplication problem in treaty systems. In *13th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'12)*, 2012.