Secure Linking: a Framework for Trusted Software Components (Extended Version)

Eunyoung Lee
Department of Computer Science
Princeton University
elee@cs.princeton.edu

Andrew W. Appel
Department of Computer Science
Princeton University
appel@cs.princeton.edu

ABSTRACT

Deciding whether or not to trust a foreign software component is important for the safety of a host which imports the component. The decision can be made in a static situation where the main concern is building a large software system out of many components from different parties, as well as in a dynamic situation where the main concern is executing codes transferred via networks. Recently some linkers have started concerning about trusting foreign code at the component level, and type checking and digital signing are the main ways of achieving the goal.

We propose a way of linking software components with certified assurances (called properties) and proof-carrying authentication. We have developed a framework for secure linking systems, which consists of the logic expressing the linking procedure and the prover finding proofs for proof-carrying authentication. The framework is general and expressive enough to represent other existing linking systems and to help different linking systems interoperate.

1. INTRODUCTION

When a software system of large scale is built, the system is analyzed carefully before being implemented and divided into several subsystems, each of which is relatively independent of other subsystems. The relative independence of software components makes it possible for each component to be implemented separately and to communicate with other components only through its interface, regardless of its inner implementation.

One of the advantages of restricting software components to communicate only through interfaces is software reuse. A large software system can be built out of independent, self-contained software components. But when a programmer uses a third-party software component as a building block of her system, she doesn't want the code she imports to break her system.

The most widely used methods for ensuring safe linking are type checking and code signing. Checking the type of the interfaces between two software components ensures that two components agree on the types they are using. Although type checking is quite strong and easy to use, it doesn't guarantee that the code will behave in an expected way.

Code signing ensures that someone trustworthy trusts the code. Currently a digital signature on codes only shows that a trusted party signed the code, but it doesn't show that the code he writes/or inspects is trustworthy. Even though a code is written by a trusted programmer, the code may do harmful things to a host by the programmer's mistake.

To solve this problem, stronger guarantees are needed. A code consumer wants to specify that a component must have certain properties to be linked safely and securely to its system. At the same time, a code provider writing/or supplying the component needs a method of proving that it has the properties required by the host. We have developed a logical framework for secure linking based on Proof-Carrying Authentication (PCA). PCA is an authentication framework based on higher-order logic [2]. Unlike traditional distributed authentication frameworks, a code provider is responsible for proving her right to access some shared or protected resources, and a code consumer only verifies the proof handed in by the client. Since PCA uses higher-order logic as a basic logic, the framework is more general and more flexible than its predecessors. Users can define operators on top of the basic logic of PCA. The semantics of each operator is expressed by the underlying basic logic, and inference rules on those operators are proved as lemmas.

In our framework, a code consumer announces its linking policy to protect itself from malicious code from outside. The policy can include certain properties required by the code consumer for system safety, such as software component names, a valid hash of codes, version information of software components, etc. To link and to execute a component in the code consumer, the provider of the component should submit a proof that the component has the properties specified in the code consumer's linking policy. The proof is formed by the basic logic and inference rules of the framework. After being submitted, the proof is checked by a small trusted proof checker in the code consumer, and if verified, the component is allowed to be linked to other components in the code consumer.

The breakdown of the paper is as follows: In section 2, we mention some work related to our research. In section 3, we explain the key concept of our proposing linking procedure with a typical situation when a linking decision should be made. In section 4 and 5, we present the user interface languages of our framework and the basic logic, using the example in section 3. In section 6, we show that the tactical prover of our framework is sound and complete. In section 7, we demonstrate that our linking logic is expressive and general enough to describe the existing linking systems with it.

2. RELATED WORK

2.1 PCA

The idea of Proof-Carrying Authentication was introduced by Appel and Felten [2]. PCA is a distributed authentication/authorization framework based on proof-carrying mechanism. PCA is different from previously existing authentication frameworks in two ways:

- it uses a higher-order logic that makes PCA more general and more flexible, and
- a code consumer needs not execute a complicated decision procedure to grant the client's request. A code provider is responsible for proving her capability of access.

Authentication frameworks and protocols have been described using formal logic [1], for example in the Taos distributed operating system [12]. Taos has a logic of authentication on top of propositional calculus, which are proved to be sound.

In Taos, a code consumer given an access request has to decide whether to grant the request. Wobber et al. [12] chose to implement only a decidable subset of their authentication logic since they want the decision procedure of granting a request to be decidable. Decidable logic are weaker than general logics, so this makes the authentication logic less flexible. To make an application-specific inference logic, some application-specific rules should be added to a given set of basic inference rules and the soundness of the whole logic should be proved again.

PCA gains more flexibility by allowing quantification over predicates. Therefore the authentication framework has only one set of inference rules and all application-specific rules are proved as lemmas. To describe an application-specific security policy, a programmer only needs to pick an application-specific decidable subset of the underlying general logic of PCA.

Since all the application-specific logics are expressed using the same general inference rules, they can interoperate with each other easily. This makes PCA more general than other previous authentication logic. However, finding a proof for a request is not always possible because higher-order logic is not decidable. To get around this problem, PCA puts the burden of constructing proofs on the client and on the contrary the server simply checks that proof. This is in analogy with proof-carrying code [9]. Even in an undecidable logic, proof checking (not proving!) can be simple and efficient. Bauer, Schneider and Felten developed an infrastructure for distributed authorization based on the ideas of PCA [5].

2.2 Component models

A fundamental principle of software engineering is to divide a large-scale program into relatively independent and small subprograms. With this strategy, communication between the subprograms and seamless integration of them become important in order to protect each subprogram from malicious attack or inadvertent misuse of other subprograms, and eventually in order to make the large program work. Traditionally, each program protects itself while communicating with other programs through abstract data types (ADT) or information hiding.

Some languages, for example Standard ML and its associated Compilation Manager (CM) [6], support more facilities than simple ADTs by making it possible to structure modules hierarchically. Rather than having a flat and simple space of modules, CM makes it possible to build a hierarchy of modules by describing the relationship between them with its own descriptive language. Furthermore, it provides the facility of restricting view of modules and the facility to be able to see modules through a richer interface. Within the module hierarchy, modules in lower levels can communicate across more expressive interfaces, and modules in higher levels can enforce more restrictive ones. Bauer, Appel and Felten extended the Java package mechanism and developed a linking system supporting hierarchical modularity similar to that of Standard ML and CM [4].

2.3 Assemblies of the .NET framework

The .NET framework is a computing platform developed by Microsoft targeting the highly distributed environment of the Internet [11]. The main components of the .NET framework are Common Language Runtime and its class library.

Common Language Runtime (CLR) is responsible for execution-time management such as memory management, thread management and remote procedure calls. It could be compared to the Java Virtual Machine [8]. CLR provides its own intermediate language called Common Intermediate Language (CIL) and a code must be written in this intermediate language to be executed on CLR. CLR also implements a strict type- and code-verification infrastructure called Common Type System (CTS) and supports Just-In-Time (JIT) compiling for enhancing performance.

The .NET framework class library is a collection of reusable classes, or types, that tightly integrate with the common language runtime. It provides a variety of classes, from the basic classes for graphical user interface to the more sophisticated classes and tools for development and consumption of Web services supporting the standards such as SOAP (standard object access protocol), XML (an extensible mark-up language).

2.3.1 Assemblies

An assembly is the logical unit of a program executable on the common language runtime. It is also a unit of security, a unit of type and a unit of version within the .NET framework, as well as a deployment unit during runtime execution. An assembly consists of four elements: the assembly manifest, type metadata, CIL code, and a set of resources (such as .bmp or .jpg files).

The assembly manifest contains a collection of data that describes how the elements in the assembly relate to each other. This metadata includes the name of the assembly, version number, its cultural background (such as language), list of all files in the assembly, type reference information, and information on referenced assemblies. Assembly developers can add or change some information in the assembly manifest by putting assembly attributes declaratively in their source codes. But an assembly manifest is created automatically by compilers or programming tools supporting the .NET framework. The .NET framework provides the CIL disassembler to view CIL information in a file. If the file being examined is an assembly, this information can include the assembly's attributes, as well as references to other modules and assemblies.

2.3.2 Versioning

An assembly is used as the unit of linking deployment, and execution in the .NET framework. When a developer build an application, the main assembly is linked to other reference assemblies, each of which is identified by using the information such as the name of the assembly, the version number, the cultural information, etc.. However, sometimes the developer wants the application to run against a newer version of an assembly. The .NET framework supports redirecting assembly versions through configuration files. Configuration files and decision procedures for version redirection will be discussed in detail later.

The .NET framework also supports side-by-side execution. This is the ability to run multiple versions of assemblies of the same name simultaneously. Support for side-by-side storage and execution of different versions of the same assembly is an integral part of versioning, and is built into a part of the runtime. Treating an assembly's version number as part of its identity enables the runtime to store multiple versions of assemblies under the same name and distinguish them at run time.

3. EXAMPLE

In this section, we explain the design of our framework with a simple, but realistic example. Suppose that a principal Alice has a component named `compiler`. Alice could be a programmer who wrote the component, or she could be a customer who bought the component from a third-party developer. Now, Alice wants to send the component `compiler` to another principal named Bob, and execute `compiler` after linking to other components in his system. Because Bob wants to keep his system secure and cannot trust the safety of the component Alice provides, he requires Alice to prove that her software component is safe to be linked to other components in his system. To protect his system from all untrusted components from outside, Bob tells his security policy to Alice. A linking policy usually consists of three parts: the description of software components Bob provides (call it a *library*), a list of useful properties Bob requires for outside software components, and the names of authorities trusted by Bob.

3.1 Properties

A code consumer requires a software component to have some preannounced properties in order for the component to be linked to other components in his system. A *property* of a component is an assertion of expected behavior from the component. There are many useful properties which help protect systems from malicious outside codes, such as "this software component is type-checked," "this software component never accesses outside of the memory which is assigned to it," "this software component doesn't read any information from or write any information to the file system," or "this software component doesn't produce any arithmetic overflow or underflow."

In our example, suppose that Bob requires that every component from outside should have a property called `prp_type_checked`, that means that Bob will allow only a type-checked component to be linked to his system. Now Alice must prove her component `compiler` has the property `prp_type_checked` in order to link the component to Bob's system.

3.2 Property authorities

Some properties, like the property of being type checked, can be guaranteed by a trusted compiler, while others cannot be proven easily. These properties, however, may be accepted as true if a software component has assurances made by a trusted third-party authority. The trusted authority can make assurances resulting from a software audit or some other verification processes for software engineering. Such assurances are usually encoded as digitally signed statements. We call those authorities *property authorities.* After verifying the digital signatures on the statements, the property statements made by trusted property authorities are accepted as true and the components are considered to have the properties assured in those statements.

In our example, as shown in Figure 1, Charlie is a property authority who examines a software component and determines if the component has the property `prp_type_checked`. Since Bob announced that he trusts Charlie as a property authority of `prp_type_checked`, Alice must get a digitally signed assurance from Charlie that the component `compiler` has the property `prp_type_checked`.

3.3 Key authorities

Since all certificates from property authorities comes with digital signatures, a code consumer must know what the signers' public keys are in order to check the validity of the digital signatures. The code consumer must at least know who can provide the right public keys for verifying the signatures and what her public key is. These authorities are called *key authorities* (also known as certificate authorities). Key authorities are responsible for guaranteeing the bindings between a principal name and a public key. Key certificates signed by a key authority are verified with her already known public key. Hence, it is usual for the key certificates to form a chain of trust.

Diane, in Figure 1, is a key authority who ensures the bindings between a principal and the principal's public key. Therefore, when Bob gets a digitally signed property certificate

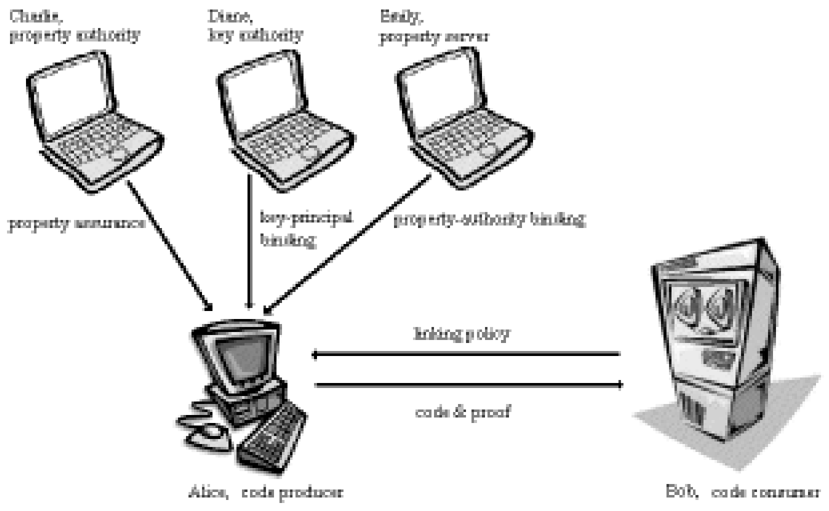

Figure 1: An example of secure linking

from Charlie, Bob doesn't need to know in advance Charlie's public key to verify the signature of that certificate. Instead Bob asks Diane for Charlie's key certificate (or Bob might ask Alice to get Charlie's key certificate from Diane to complete her proof); he gets Charlie's public key after verifying the key certificate with Diane's public key.

3.4 Property servers

Just as Bob doesn't have to know all the public keys of principals, he doesn't have to know all the bindings between properties and property authorities. Instead Bob announces that he trusts a principal as someone who will let him know the property-authority bindings. Therefore, Bob doesn't enumerate the names of property authorities for every required property in his linking policy, and it relieves him from modifying and reannouncing the linking policy whenever a property-authority binding changes.

In our example, Bob announces that he trusts Emily as a property server. So Alice should consult Emily to find out from whom she can get the property statements for the component `compiler`.

3.5 Library

Although it is possible to write a software component to be self-contained, it is very natural for a software component to use already existing software components by importing them. A code consumer announces what components it has and what properties are exported by each of those components. At the same time, a component from a code provider declares what components it imports and what properties are required for each of them in its component description. For a foreign component to be linked safely, a secure linker checks whether or not the library of a code consumer provides all the software components that are required by the foreign component.

Suppose that the component `compiler` uses a hash table during its computation and imports a component called `hashTable` with a property `prp_efficient_search`, and Bob's library has several different components named `hashTable`. These are different from each other in terms of the properties they export. For example, one `hashTable` component doesn't export any properties verified by property authorities, while another `hashtable` components exports `prp_efficient_search`. Because the component `compiler` requires the component `hashTable` with `prp_efficient_search`, only the second `hashTable` component can be linked to the component `compiler`.

3.6 Linking decision

To decide to link a component coming from outside to other components in the system, a code consumer must verify whether or not the component provides all the required properties.

For example, Bob checks the proof from Alice with the certificates by using a trusted proof checker, and the component `compiler` to be linked to other components in his system if the proof is valid, otherwise he rejects it. Since the certificates Alice provides are digitally signed, during the proof-checking time, the validity of digital signatures are checked. The detailed verification steps are as follows:

1. verifies the digital signatures on the key certificates with Diane's public key to get Emily's and Charlie's public keys,
2. verifies the digital signatures on the property-authority binding certificates from Emily with Emily's public key obtained in the first step,
3. verifies the digital signatures on the property statements from Charlie with Charlie's public key obtained in the first step.

In what follows, we will explain our framework using the example above. Our framework is independent of program-

```
⟨componentDsc⟩
 ⟨name⟩ compiler ⟨/name⟩
 ⟨modules⟩
  ⟨item hash = “194CA77319” ⟩ compiler.class ⟨/item⟩
  ⟨item hash = “EF41900142” ⟩ regAlloc.class ⟨/item⟩
 ⟨/modules⟩
 ⟨exports⟩
  ⟨type⟩
   ⟨item⟩ class compiler⟨/item ⟩
   ⟨item⟩ interface regAlloc⟨/item⟩
  ⟨/type⟩
  ⟨property⟩ ⟨item⟩ prp_type_safety ⟨/item⟩
  ⟨/property⟩ ⟨/exports⟩
 ⟨imports⟩
  ⟨component⟩
   ⟨name⟩ hashTable ⟨/name⟩
   ⟨required⟩
    ⟨type⟩ ⟨item⟩ class hashtable ⟨/item⟩ ⟨/type⟩
    ⟨property⟩
     ⟨item⟩ prp_type_safety ⟨/item⟩
     ⟨item⟩ prp_efficient_search ⟨/item⟩
    ⟨/property⟩ ⟨/required⟩ ⟨/component⟩ ⟨/imports⟩
⟨/componentDsc⟩
```

Figure 2: A component description in XML

ming languages or programming environments; thus, the explanation about the framework is language-neutral.

4. USER INTERFACE LANGUAGES

The user interface of our framework consists of two different languages, one for describing components, and one for describing code consumers' security policies.

The user interface languages adopts the XML syntax; therefore they can be parsed by any XML parser and modified with a simple text editor. For our framework, we developed XML parsers for each language, producing our linking logic from XML files. In this section, we will explain the syntax and the semantics of the two description languages and present their syntax by way of example.

4.1 Component description language

Going back to the example in Section 3, Alice must describe the software component `compiler` to Bob's linking system in order to have it linked in his system.

Figure 4 shows the abstract syntax of the component description language, and the description of the component `compiler` in our example might look like the example in Figure 2.

4.1.1 ⟨componentDsc⟩

This is a root tag for component description. ⟨componentDsc⟩ stands for the beginning of the description and ⟨/componentDsc⟩ stands for the end of the description.

4.1.2 ⟨name⟩

This tag is used for specifying the name of the software component. A component name is a local identifier of convenience for referring the component.

```
⟨linkingPolicy⟩
 ⟨library⟩
  ⟨component⟩
   ⟨name⟩ hashTable ⟨/name⟩
   ⟨module⟩
    ⟨item hash = “9213317FCA”⟩
        hashtable1.class ⟨/item⟩ ⟨/module⟩
   ⟨exports⟩
    ⟨type⟩
     ⟨item⟩ class hashtable ⟨/item⟩
    ⟨/type⟩ ⟨/exports⟩ ⟨/component⟩

  ⟨component⟩
   ⟨name⟩ hashTable ⟨/name⟩
   ⟨module⟩
    ⟨item hash = “683EE18970”⟩
       hashtable2.class ⟨/item⟩ ⟨/module⟩
   ⟨exports⟩
    ⟨type⟩
     ⟨item⟩ class hashtable ⟨/item⟩ ⟨/type⟩
    ⟨property⟩
     ⟨item⟩ prp_hash_table ⟨/item⟩
     ⟨item⟩ prp_efficient_search ⟨/item⟩
    ⟨/property⟩ ⟨/exports⟩ ⟨/component⟩ ⟨/library⟩

 ⟨keyAuth⟩
  ⟨item⟩ Diane ⟨/item⟩ ⟨/keyAuth⟩

 ⟨propertyServer⟩
  ⟨item⟩ Emily ⟨/item⟩ ⟨/propertyServer⟩

 ⟨requiredPrps⟩
  ⟨item⟩ prp_type_safety ⟨/item⟩
 ⟨/requiredPrps⟩ ⟨/linkingPolicy⟩
```

Figure 3: A linking policy description in XML

4.1.3 ⟨modules⟩

The *module* part of the description specifies which code files compose the software component. Every entry of this part is given by its name and the hash code of it. For example, the example in Figure 2 shows that the component `compiler` consists of two code files, called *compiler.class* and *regalloc.class*, and their cryptographic hash codes are store in the component description.

Given binary files (programs or resources) and source program files, code consumers can check the integrity of those code files by recalculating their cryptographic hash codes, and comparing with the hash codes sent by code producers.

4.1.4 ⟨exports⟩

Components can export two kinds of things, type identifiers (such as class names or function names) and properties. It is very typical for a program to call functions, or to use classes or structures outside of its scope. Usually linkers find out what functions, classes, or structures are really referenced by consulting the symbol tables handed to them by compilers. In some cases, all the identifiers defined within a program are visible from the outside. With the Standard ML/Compilation Manager [6], however, a programmer can specify whether an identifier is visible from outside or not by declaring a list of visible identifiers in its group description. Similarly, with object-oriented programming languages such as Java or C++, a programmer can control the visibility of identifiers by putting access modifiers in its declaration.

In the component description language, the tag ⟨*type*⟩ is used to enumerate the identifiers visible from outside. Some identifiers are not visible from outside of a component if they are not listed in the ⟨exports⟩ part, even if they are defined within the component. The tag ⟨*property*⟩ is used to enumerate the properties a component exports. For each entry of the listed properties, a digitally signed assurance of property authority should be provided.

4.1.5 ⟨imports⟩

This part specifies what other components a component depends on. Every entry of the import part consists of a brief description of an imported component (call it an *import component description*). The import component description differs from the complete component description in that it doesn't have the module information, but contains the name of a component and exported properties. The following tags are used to build an import component description.

- ⟨**component**⟩ The tags ⟨component⟩ and ⟨/component⟩ are used to specify the beginning and the end of the import component description.
- ⟨**name**⟩**,** ⟨**exports**⟩ The syntax and semantics of these parts are the same as those of the complete component description.

4.2 Linking policy description language

The linking policy description language allows system administrators to set security policies that affect how applications run on their machines.

Figure 5 shows the abstract syntax of the language. The linking policy which Bob announces in our example might look like the example in Figure 3.

4.2.1 ⟨linkingPolicy⟩

This is a root tag for a linking policy. ⟨linkingPolicy⟩ stands for the beginning of the description and ⟨/linkingPolicy⟩ stands for the end.

4.2.2 ⟨library⟩

The tag ⟨library⟩ is used for enumerating the library component description of a code consumer, which a code provider can import. The library component description is different from the complete component description in that it doesn't have an import part. Since the library components are considered a part of a code consumer's system (and hence they are trusted), it is more important to specify what they provide to the outside of the system, than what they require from other parts in the system.

- ⟨**component**⟩ This part specifies the description of one library component. The library description begins with the tag ⟨component⟩ under the tag ⟨library⟩ and ends with the tag ⟨/component⟩.
- ⟨**name**⟩**,** ⟨**module**⟩**,** ⟨**exports**⟩ The syntax and semantics of these tags are the same as those in the complete component description.

4.2.3 ⟨keyAuth⟩

The tags ⟨keyAuth⟩ and ⟨/keyAuth⟩ are used for enumerating the names of the key authorities whom a code consumer trusts. Key authorities are responsible for providing the bindings between a principal and its public key.

4.2.4 ⟨propertyServer⟩

The tags ⟨propertyServer⟩ and ⟨/propertyServer⟩ are used for enumerating the names of the property servers whom the server trusts. As explained in Section 3, property servers are responsible for providing the bindings between a property and its property authorities.

4.2.5 ⟨requiredPrps⟩

This part is used to specify the properties a code consumer requires all the components from outside to have.

5. LINKING LOGIC

The linking logic of the secure linking framework is a higher-order logic defined on top of PCA logic [2]; therefore the semantics of each operator are expressed in terms of the underlying PCA logic, and inference rules using operators are then proved as lemmas. In this section, we explain how we translate component descriptions and linking policies described in the user interface languages into the linking logic, and how we represent the linking decision procedure in the linking logic.

5.1 Representing properties

In our framework, a code consumer announces in advance which properties it requires from a foreign component. Code providers are responsible for proving that their components have the properties required by a code consumer. Therefore, an essential part of the logic is to check if two properties match each other.

In designing the logic for property matching, we had three purposes. While implementing properties in the linking logic, we wanted:

- to check if two properties match each other easily,
- to make adding new properties simple, and
- to isolate the implementation details of properties from other parts of the linking logic.

We achieved the first goal by dividing property matching into two parts, *properties* and *property requests*. In our framework, a property request is a predicate of type *prp_req*, accepting an argument of type *property*; it returns true if the argument matches the request it implements, and returns false otherwise. The following is a general inference rule of property matching:

$$\frac{\begin{array}{l} rq = mk\_prp\_req(rprp) \\ prp\_eq(rprp, prp) \end{array}}{rq(prp)} \; prp\_match$$

ComponentDsc	::= `componentDsc` *Name Modules* $Exports^*$ $Imports^*$
Name	::= `name` *NameId*
Modules	::= `modules` $ModuleIdItem^+$
Exports	::= `exports` $Types^*$ $Properties^*$
Imports	::= `imports` $ImportComponentId^*$
ImportComponentId	::= `component` *Name* $RequiredPrps^*$
RequiredPrps	::= `required` $Types^*$ $Properties^*$
NameId	::= *String*
ModuleIdItem	::= `item` *HashCodeId ModuleName*
Modulename	::= *String*
HashCodeId	::= `hash` *String*
Types	::= `type` $TypeIdItem^+$
TypeIdItem	::= *String*
Properties	::= `property` $PropertyIdItem^+$
PropertyIdItem	::= *String*
String	::= $(0\mid 1\mid \ldots\mid a\mid b\mid \ldots\mid z\mid A\mid B\mid \ldots\mid Z\mid\ \mid \_)^+$
QuotedString	::= "*String*"

Figure 4: Abstract syntax of the component description language

LinkingPolicy	::= `linkingPolicy` $Library^*$ $PropertyServer^*$ $RequiredPrps^*$
Library	::= `library` $ExportComponentId^+$
KeyAuthority	::= `keyAuth` $ServerIdItem^+$
PropertyServer	::= `propertyServer` $ServerIdItem^+$
RequiredPrps	::= `requiredPrps` $PropertyIdtem^*$
ExportComponentId	::= `component` *Name Modules* $Exports^*$
Name	::= `name` *NameId*
Modules	::= `module` $ModuleIdItem^+$
Exports	::= `exports` $Types^*$ $Properties^*$
NameId	::= *String*
ModuleIdItem	::= `item` *Name HashCodeId*
HashCodeId	::= *String*
Types	::= `type` $TypeIdItem^+$
TypeIdItem	::= *String*
Properties	::= `property` $PropertyIdItem^+$
PropertyIdItem	::= *String*
ServerIdItem	::= *String*
String	::= $(0\mid 1\mid \ldots\mid a\mid b\mid \ldots\mid z\mid A\mid B\mid \ldots\mid Z\mid\ \mid \_)^+$
QuotedString	::= "*String*"

Figure 5: Abstract syntax of the linking policy description language

The formula constructor *mk_prp_req* accepts an argument `rprp` of type *property* and returns a predicate `rq` of type *prp_req*. Given a property `prp` as an argument, the predicate `rq` checks if `prp` matches the property `rprp`, relying on the predicate *prp_eq*. That is, the predicate `rq` returns true if and only if the result of applying `rprp` and `prp` to the predicate *prp_eq* is true. Since general property matching depends on *prp_eq*, the semantics of property matching also depend on the semantics of *prp_eq*. The predicate *prp_eq* returns true when two arguments are equal. Therefore, the predicate `preq` returns true if and only if its argument is equal to what the property `rprp` stands for.

By introducing the predicate type *prp_req* as well as the type *property*, we turn the procedure of checking property matching into simple evaluation of a predicate. Thus, the required properties in the linking policy of a code consumer are encoded in the form of *prp_req*, and a code producer proves that the set of its exported properties is a superset of the set of properties satisfying all the encoded predicates.

As shown in Section 3.1, there exist many different kinds of properties. Therefore it is useful to make it possible to add different properties and property requests to the framework without changing the other parts of the linking logic. Abstracting the types *property* and *prp_req* helps achieve this goal.

Users of our secure linking framework can add new properties to the framework easily. We will explain step by step how a user can define some basic properties and their property requests with an example. Figure 6 shows the inference rules for component names.

First, the user defines a property for component names. Every property has a unique name of type *prp_kind* used as a tag making itself distinguishable from other properties. Suppose that a new property kind *prp_component_name* is defined for the component name property. Second, the user designs and implements a formula constructor for the property. The formula constructor encodes the information of the property, which can be used when checking property matching. For the property *prp_component_name*, the formula constructor *mk_prp_component_name* accepts an identifier, the name of a component, and encodes this information into the returned property.

After that, the user can complete the work by adding predicates of type *prp_req*, which are used to decide whether or not a property matches the encoded property. Note that more than one predicate of type *prp_req* can be defined for a property. By giving proper semantics to the predicates of *prp_req*, the user can have several different semantics for property matching. This makes the property matching more flexible. In case of the property *prp_component_name*, the user might want to check if a software component has a component name without caring what the exact component name is. In the other case, however, the user might want to check if a component has the exact name he wants.

Two different ways of checking one property can be accomplished by providing two different property requests for one property. The inference rule *name_exists* in Figure 6 is used to check if a property given as an argument is a *prp_component_name* property, building proofs of *rq_component_name_exists*. It doesn't matter what component name the argument has.

$$\frac{prp\_kind(p, prp\_component\_name)}{rq\_component\_name\_exists(p)}\; name\_exists$$

$$\frac{\begin{array}{l} p\_name\_request = mk\_rq\_component\_name(p) \\ prp\_eq(p, q) \\ prp\_kind(q, prp\_component\_name) \end{array}}{p\_name\_request(q)}\; name\_match$$

Figure 6: Rules for component names

The inference rule *name_match* is used for exact matching. The predicate *mk_component_name_request* builds a property request `p_name_requests` from a property `p` whose kind is *prp_component_name*. Then the property request `p_name_requests` returns true if and only if a property `q` given as argument is of kind *prp_component_name* and equal to the encoded property `p`.

The last goal of our design is to separate the linking decision part from type *property* and *prp_req* implementation. By separating these concerns, we increased the scalability and flexibility of the framework.

While making a linking decision, property matching happens in two places: a secure linker checks if a software component exports all the required properties of a code consumer, and if a code consumer's library satisfies a foreign component's importing requests. Figure 7 shows the inference rules for these two cases. The inference rule *has_prps_i* is used to check if a set `prpset` of properties contains all properties which makes all the property requests in a set `rqset` of property requests. The inference rule *xprt_prps_i* checks if a component description `cdsc` exports all the required properties `rqset` of a code consumer, building proofs of the predicate *export_required_prps*. The predicates *export_required_prps* returns true if the set `xprtset` of exported properties satisfies the predicate *has_property* with the set `rqset`. The inference rule *stsf_imprt_i* checks if a library `libList` exports all properties `imprtList` of a code producer's importing property requests. The predicate *satisfy_import_rq* returns true if for every member `iset` in the list `libList`, there exists a property set `eset` which is a member of `libList`, and makes the predicate *has_property* true with the set `iset`.

As shown in the rules in Figure 7, the semantics of these decision predicates, *export_required_prps* and *satisfy_import_rq*, depend on the predicate *has_property*. Therefore, the predicate *has_property* is the only part concerned with the implementation of property matching. Furthermore, even the predicate *has_property* checks if a property satisfies a given property predicate. In other words, the predicate *has_property* depends only on the abstract part of types *property* and *prp_req*, not on the details of their implementation.

5.2 Translating description languages

As mentioned earlier, the XML parsers of the interface languages parse component descriptions and linking policies

$$\frac{\forall rq.(rq \in rqset \Rightarrow \exists p.rq(p))}{has\_property(rqset, prpset)}\ has\_prps\_i$$

$$\frac{\begin{array}{l}\forall iset.(list\_member(imprtList, iset) \Rightarrow \\ \exists eset.(list\_member(libList, eset) \wedge \\ has\_property(iset, eset)))\end{array}}{satisfy\_import\_rq(imprtList, libList)}\ stsf\_imprt\_i$$

$$\frac{\begin{array}{l}xprtset = component\_dsc\_prps(cdsc) \\ has\_property(rqset, xprtset)\end{array}}{export\_required\_prps(rqset, cdsc)}\ xprt\_prps\_i$$

Figure 7: Rules for property matching

into the linking logic. In this section, we will explain how to represent the descriptions in the linking logic.

5.2.1 Component description

A component description is usually turned into formulas in the linking logic. The cryptographic hash codes of modules are verified, and digitally signed certificates from key authorities, property authorities, or property servers are converted into axioms in the logic after verifying their signatures.

A component description is encoded in the form of a set of properties and a list of sets of property requests. The name of a component, the set of exported type identifiers, and the set of exported properties are all treated as properties and encoded into instances of type *property*. These make up the export part of the component.

The list of sets of property requests corresponds to the import part of a component. Each set of property requests stands for one imported component, and includes the property requests for required properties such as a component name. Since a software component usually imports more than one component, the sets of property requests for imported components form a list. By using a list rather than using a set, we can handle some cases in which the order of importing components is critical in making a linking decision.

Component descriptions can be combined by using the formula constructor *cdsc_combine*, which accepts two terms of type *component_dsc* and returns a term of type *component_dsc*. The export part of a combined component description is a union set of the export property sets of two inputs, and the import part of it is a concatenated list of the import lists of two inputs.

In addition, the linking logic provides a binary relation *sub_cdsc*, which determines one argument is a sub-component of the other. Intuitively, a component description `cdsc_A` is a sub-component of a component description `cdsc_B` if the export part of `cdsc_A` is a subset of the export part of `cdsc_B`, and the import part of `cdsc_A` is a sublist of the import part of `cdsc_B`.

Figure 8 shows the inference rules of *cdsc_combine* and *sub_cdsc*. The rules *sub_cdsc_combine1* and *sub_cdsc_combine2* show that an input component description becomes a sub-component of the resulting combined component description. The rules *sub_cdsc_refl* and *sub_cdsc_trans* imply that the relation *sub_cdsc* is reflexive and transitive.

It is very useful to make it possible to combine component descriptions, especially when considering digitally-signed certificates from property authorities. In our framework, a component description with modules is required to be digitally signed by property authorities to prove its safety during the linking time. Property authorities announce what properties they can guarantee in the form of a set of *property requests*. When signing, it is reasonable for a property authority to want to sign only on the properties he can guarantee, rather than sign on all the properties a component description exports. For example, a trusted compiler can guarantee that modules with a given component description are type-safe, but doesn't want to, or is not able to, guarantee any other properties. This is the reason why we include the formula constructor *cdsc_combine* and the binary relation *sub_cdsc*. After collecting component descriptions assured by property authorities, a code producer combines the small descriptions, and builds a complete component description It frees the property authorities from a burden of assuring more properties of a component description than they want.

5.2.2 Linking policy description

The linking policy is translated into two forms in the linking logic, axioms and formulas. The names of property servers and of key authorities are turned into axioms.

The library components are encoded as two lists in the linking logic. The first list consists of sets of modules. Each element of the list corresponds to the set of modules making up a library component. The second list consists of sets of properties. Each elements of the second list stands for the set of exported *properties* of a library component. The name of a library component, the exported type identifiers, and the exported properties are encoded as a set of *properties*. Since a code consumer usually provides more than one library component, they are put into a list. The linking logic at linking time requires that every element in the library component description list must be a proper description of the corresponding element in the library component module list.

At the same time, linking policies specify the properties required by code consumers. Each required property is translated into a corresponding predicate of type *prp_req*. To-

$$\frac{}{subset(cdsc\_exp(A), cdsc\_exp(cdsc\_combine(A,B)))}\ cdsc\_combine\_export1$$

$$\frac{}{subset(cdsc\_exp(B), cdsc\_exp(cdsc\_combine(A,B)))}\ cdsc\_combine\_export2$$

$$\frac{}{sublist(cdsc\_imp(A), cdsc\_imp(cdsc\_combine(A,B)))}\ cdsc\_combine\_import1$$

$$\frac{}{sublist(cdsc\_imp(B), cdsc\_imp(cdsc\_combine(A,B)))}\ cdsc\_combine\_import2$$

$$\frac{}{sub\_cdsc(A, cdsc\_combine(A,B))}\ sub\_cdsc\_combine1$$

$$\frac{}{sub\_cdsc(B, cdsc\_combine(A,B))}\ sub\_cdsc\_combine2$$

$$\frac{}{sub\_cdsc(A,A)}\ sub\_cdsc\_refl$$

$$\frac{sub\_cdsc(A,B) \quad sub\_cdsc(B,C)}{sub\_cdsc(A,C)}\ sub\_cdsc\_trans$$

Figure 8: Semantics of cdsc_combine and sub_cdsc

gether they form a set. For example, if a code consumer requires every foreign component to have a name, it is turned into the predicate *rq_component_name_exists* of Figure 6.

5.3 Authenticating certificates

We use PCA logic [2] in order to encode authorities (key authorities, property authorities or property servers), and to verify certificates from the authorities. PCA logic provides some primitives of verifying digital signatures and of representing authorities, such as *signed*, *says*, or *controls*.

The following is the inference rule used for verifying a certificate from a property authority.

$$\frac{\begin{array}{l} key\_authority(ca) \\ keybind(caKey, ca) \\ property\_authority(ma) \\ signed(caKey, keybind(maKey, ma)) \\ signed(maKey, module\_dsc(m, dsc)) \end{array}}{valid\_sig\_component\_dsc(ma, m, dsc)}\ valid\_sig\_cdsc$$

By the rule *valid_sig_cdsc*, the predicate *valid_sig_component_dsc* holds; in other words, a secure linker believes that a property authority `ma` assures that a component description `dsc` describes a set `m` of modules if the following premises hold: first, there should exist a key authority `ca`, and its public key `caKey` trusted by a code consumer. Second, a property authority `ma` should be trusted as a valid property authority. Third, there should exist a key certificate signed by `caKey` assuring the public key of `ma` is `maKey`. Finally, a code producer provides a certificate from the property authority `ma` signed by `maKey` assuring `m` is described by `dsc`.

If *valid_sig_component_dsc* holds, then the certificate from the authority `ma` is verified and believed true.

5.4 Making a linking decision

To link a component to other components of a code consumer, a code producer must show that her component exports all the properties required by the code consumer. This can be done by showing that the set of modules of the component and its component description satisfy the predicate *ok_to_link* with the linking policy specified by the code consumer.

The inference rule *ok_to_link_i* shows what steps a secure linker should take to make a linking decision. The following is the inference rule for linking decision.

$$\frac{\begin{array}{l} signed\_component\_dsc(m, dsc, prqset) \\ provides\_enough\_prps(dsc, lib, libdsc) \\ exports\_required\_prps(prqset, dsc) \end{array}}{ok\_to\_link(m, dsc, lib, libdsc, prqset)}\ ok\_to\_link\_i$$

First, the linker examines if the given logical description `dsc` of a component really represents the given set of modules `m`; if so, the code producer can prove that the predicate *signed_component_dsc* holds. Second, the linker examines if the component can obtain all the imported components from the library `lib` and `libdsc` of the code consumer; if so, the code producer can prove that the predicate *provides_enough_prps* holds. Last, the linker examines if the component description exports all the required properties `prqset`; if so, the code producer can prove that the predicate *exports_required_prps* holds.

If the component description and modules satisfy the above three conditions, in other words, if the code producer can prove that those three predicates hold, linking is allowed; otherwise, it is denied. All the decision steps are addressed in the linking logic as operators and lemmas on top of PCA logic, and all the lemmas are proved.

Given a proof from a code producer, a code consumer must be able to verify the validity of the proof. Our framework is built on the Twelf logical framework [10]. The Twelf system is one of the implementations of the logical framework LF [7], which allows the specification of logics. Since our linking logic is written on top of PCA, an object logic of LF, every term in our linking logic boils down to a term in the underlying LF logic. Therefore, the proof provided by a code producer is encoded as an LF term. The type of the term is the statement of the proof; the body of the term is the proof's derivation.

By the Curry-Howard isomorphism, checking the correctness of deriving the term that represents a proof is equivalent to type checking of the term. If the term is well typed, then the derivation is correct; hence, a code producer has succeeded in proving the proposition. If the proof of a code producer is checked, a secure linker of our framework will allow the component with the proof to be linked.

6. TACTICAL PROVER

We have developed a tactical prover for our linking logic. The prover is a logic program running on the Twelf logical framework [10]. The goal to be proved is encoded as the statement of a theorem, and axioms that are likely to be helpful in proving the theorem are added as assumptions. The prover generates a derivation of the theorem; this is the proof that a code provider must send to a code consumer.

Our tactical prover consists of 30 tacticals and 58 tactics: tactics, reducing goals to subgoals, and tacticals, providing primitives for combining tactics into larger ones that can give multiple proof-steps.

6.1 Soundness

By showing the soundness of logic, it is guaranteed that every formula that is provable (or derivable) by the prover (consisting of axioms and inference rules) is true in the logic.

Appel and Felty [3] showed that using a dependently typed programming language can yield a partial correctness guarantee for a theorem prover: if it type-checks, then any proof (or subproof) that it builds will be valid. Twelf is such a higher-order dependently typed logic programming language, and our prover is easily seen to be sound by the method of Appel & Felty.

Although a dependent type system is very useful for showing the soundness of a prover, it doesn't guarantee that the prover is complete. Next, we show the completeness of our prover.

6.2 Completeness

In this section, we will prove that our tactical prover always terminates and that it is complete. By proving the termination of the prover, we can guarantee that the prover returns a result, regardless of the input. By proving the completeness of the prover, we can make sure that the prover finds a derivation for every true formula formed by the linking logic.

The tactical prover in our framework consists of 30 tacticals and their related tactics. The 30 tacticals are categorized into three groups:

- a tactical which is used only for finding a proof from a list of assumptions. Assumptions are given by code consumers or authorities (key authorities, property authorities or property servers), and believed true.
- tacticals which are used for finding a proof of a formula without assumptions, relying only on the linking logic.
- tacticals which are used for finding a proof of a formula, relying on the assumptions as well as the linking logic.

As mentioned earlier, if certificates from authorities are verified, they are considered true, and translated into axioms in the linking logic. These axioms make up a list of assumptions for the prover. Therefore, the prover must be able to search the assumptions and find proofs from the assumptions. The tactical in the first group is used for that purpose.

On the other hand, some goals can be proved successfully by the tactics without relying on assumptions. In this case, the prover doesn't have to maintain a list of assumptions or search the list; hence, simpler tacticals in the second group are used to find the proofs.

The tacticals in the last group are more comprehensive, in the sense that they use the tactical in the first group as well as the tacticals in the second group in order to prove their goal.

The tacticals and related tactics of each group are shown in Appendix A. We related each formula constructor in the linking logic to one tactical in the prover. That is, for a theorem formed by a given constructor, the proof of the theorem is derivable only by the related tactical in the prover. For example, as shown in Appendix A, the formula constructor *ok_to_link* is related to the tactical *findproof*. Therefore, any theorem formed by *ok_to_link* is derivable only by using the tactical *findproof* in the prover.

In the following sections, first we prove the termination of the prover, then its completeness.

6.2.1 Proving termination

Proving the termination of the prover guarantees that the prover always returns a result for any given input. If the given input is derivable, then the prover will return the derivation; otherwise, it reports failure. The termination of the whole prover depends on the termination of the tacticals it consists of, and the termination of each tactical depends on the termination of the tactics using the tactical.

The ultimate goal of the prover is to find the derivation of a theorem formed by the formula constructor *ok_to_link*, if there exists such a derivation. There is only one tactical *findproof* with which the prover can find the derivation of a theorem of *ok_to_link*, and only the tactic *ok_to_link_pf* uses the tactical *findproof*. That means the prover always begins proving by applying the tactic *ok_to_link_pf* to a given input goal. The prover uses the other tacticals or tactics to prove subgoals built recursively along the way.

Thus, if the tactic *ok_to_link_pf* terminates when applied to any given input, then all calls to the tactical *findproof* must terminate. This implies the prover always terminates. The termination of the prover can thus be restated as the following.

THEOREM 1. *The tactic ok_to_link_pf terminates for any* *given input.*

We will briefly describe the proof of Theorem 1 here; the full proof is provided in Appendix B.

PROOF. Applying a tactic reduces a goal into zero or more subgoals. Therefore, the termination of a tactic depends on the termination of the subsequent calls to other tacticals.

To prove that applying every input theorem to the tactic *ok_to_link_pf* terminates, we have to show that all its subsequent calls to other tacticals terminate. This can be shown by proving one by one from the bottom up that all relevant tacticals (or their related tactics) in the prover terminate.

Proving that an individual tactical (or its related tactics) in the prover terminates results in one of the following 3 cases:

- A tactic terminates if it doesn't make any subsequent calls to other tacticals.

  The tactics in this category simply return the derivation of input goals and trivially terminate. Tactics such as *set_emptyset_pf*, *set_equiv_validper_pf*, and *list_is_nil_nil_pf* fall into this class.

- A tactic terminates if every subsequent call to other tacticals terminates.

  The tactics in this category make subsequent calls for proving the subgoals, but the calls are not recursive. Therefore the termination of this kind of tactic depends solely on the termination of the tacticals subsequently called for proving subgoals. Tactics such as *ok_to_link_pf*, *valid_cdsc_pf*, and *valid_sig_auth_pf* fall into this class.

- A tactic terminates if every recursive call to the original tactical terminates.

  The tactics in this category make recursive calls for proving subgoals, using the same tactical called for proving a goal. By showing that the logical formulas for the subgoals are always subterms of the logical formula of the original goal, we can prove that a tactic of this kind terminates. We use induction on the structure of the logical formulas for goals and subgoals. Tactics such as *prv_prps_cdsc_combine_pf*, *set_union1_pf*, and *set_union2_pf* fall into this class.

□

6.2.2 Proving the completeness

Since Gödel's incompleteness theorem, it has been known that a general higher-order logic is not complete. It is, however, still possible to show that our tactical prover is complete, because we are considering only a subset of a general higher-order logic.

As explained in the previous section, the ultimate goal of the prover is to find a derivation of a theorem formed by *ok_to_link* if it is true; if not, the prover must report failure. We also pointed out that the tactic *ok_to_link_pf* is the only tactic applicable to theorems formed by *ok_to_link*. Hence, the completeness of the prover can be stated as the following theorem.

THEOREM 2. *The tactic ok_to_link_pf finds a derivation for every true theorem formed by ok_to_link.*

We will outline the proof of Theorem 2 here; the full proof is provided in Appendix C.

PROOF. The completeness of a tactic depends on the completeness of the subsequent calls to other tacticals for proving subgoals.

To prove that applying every true input formula to the tactic *ok_to_link_pf* always results in a correct derivation, we have to show that each subsequent call with a subgoal always returns a correct derivation. We can show that by showing every relevant tactical in the prover is complete from bottom up one by one.

We designed the prover in order that proving theorems is syntax-directed; in other words, for a theorem, the formula constructors used in the theorem determine the tactical to be used when proving the theorem. In Appendix A, every tactical is listed with the related formula constructors.

For example, the formula constructor *ok_to_link* is related to the tactical *findproof*, so theorems built from *ok_to_link* are proved by applying *findproof*. In the same way, the formula constructors *singleton* and *set_union* are related to the tactical *fp_set*, so theorems built from *singleton* and/or *set_union* are proved by applying the tactical *fp_set*. In case of the tactical *fp_signed_keybind*, two formula constructors, *says* and *keybind*, are related to this tactical. That means the tactical *fp_signed_keybind* is used only for proving theorems which use both of those formula constructors.

By saying that a tactical is complete, we intend for the tactical to always find a derivation of any true formula built by the related formula constructors.

Proving that an individual tactical in the prover is complete is possible by showing all the tactics calling the tactical are complete, and this comes down to one of the following three cases:

- Tactics using the tactical *search_assmp*

  The tactical *search_assmp* is used for finding a proof from a list of assumptions, where each of the assumptions is an axiom considered true by the prover. Proofs of theorems built from some formula constructors such as *key_authority*, *signed*, and *library_dsc* can be found only with the tactical *search_assmp*.

  Since the prover regards only the proofs in the list as true, the truth of a given theorem depends completely on whether or not a matching proof is in the assumption list. Therefore proving the completeness of the tactical *search_assmp* and its related tactics, in turn, proves that the prover always finds a proof if it is in a given list of assumptions. This is proved by induction on the length of the assumption list.

- Tactics calling other tacticals for proving subgoals

  The tactics in this category make subsequent calls for subgoals, but the calls are not recursive. Therefore the completeness of this kind of tactics solely depends on the completeness of the tacticals used for the subsequent calls.

  Tactics such as *ok_to_link_pf*, *valid_cdsc_pf*, and *valid_sig_auth_pf* fall into this class.

- Tactics calling the original tactical for proving subgoals

  The tactics in this category make recursive calls for proving subgoals using the same tactical it uses for proving a goal.

  We can prove that a tactic of this kind is complete by showing that the logical formulas for the subgoals are always subterms of the logical formula of the original goal, and by using induction on the structure of the logical formulas for goals and subgoals.

  Tactics such as *prv_prps_cdsc_combine_pf*, *set_union1_pf*, and *set_union2_pf* fall into this class.

□

7. CASE STUDY

In this section, we will show our linking logic is general and expressive enough to address the linking decision procedure of the .NET framework.

7.1 Version redirection in .NET

As mentioned earlier, the runtime system of .NET requires every assembly to specify its name, its culture information, its version number, the list of all the files in the assembly, and type reference information. This information is stored in the assembly's manifest. It is quite straightforward to represent this information in the linking logic as exported properties of a component description explained in Section 5.

In addition, the runtime of the .NET framework allows developers or system administrators to specify the version of an assembly to be used for linking. This enables assemblies of the same name to co-exist within one system, and enables developers or system administrators to use a different version of an assembly of the same name at linking time. In this section, we explain how this feature can be added to our framework without significant modification of the linking logic.

Users can specify assembly version redirection in configuration files at different levels. Application configuration files, machine configuration files, and publisher policy files are used to redirect one version of assembly to another with the same name. Configuration files allow developers to change runtime settings without having to recompile applications, and administrators to set policies that affect how applications run on their machines. Since configuration files are written in XML, it is convenient to edit and parse them. An example of a configuration file redirecting assembly versions is shown in Figure 9. To redirect assembly versions in the configuration file, the information for each assembly that developers want to redirect is put inside the ⟨dependentAssembly⟩ tag and the information identifying the assembly is inside the ⟨assemblyidentity⟩ tag.

The original version of an assembly and the versions of dependent assemblies are recorded automatically in the assembly's manifest at compile time. A linker decides which version of an assembly is to be linked with the following steps: First, the linker checks the original assembly reference to determine what version was originally used. Second, it checks all available configuration files to find applicable version policies in a sequence of machine configuration files, publisher policy files and application configuration files. Lastly, it determines the correct assembly version that should be linked to the calling assembly, from the information of the original assembly reference and any redirection specified in the configuration files.

7.1.1 An example

We will present how to apply version policy in detail with the example of Section 3. Suppose that a code consumer is looking for a component (in fact, an assembly in .NET) `hashTable` to link it to the component `compiler` from a code producer. Originally the component `compiler` was built on the component `hashTable` of version 1.5.0.0 and this information is stored in the assembly manifest of `compiler`. Also suppose that there exists a machine configuration file, whose content is shown in Figure 9. To decide what version of component `hashTable` will be used, first of all the code consumer must determine what is the original version of `hashTable` on which the component `compiler` was built. Then the code consumer consults the machine configuration file to see if it contains any version redirection for the component `hashTable`. Since the machine configuration file contains a version redirection for the component `hashTable` shown in Figure 9, now the code consumer checks if the original version of `hashTable` (in this case, 1.5.0.0) is within the range affected by the redirection. The original version 1.5.0.0 of `hashTable` lies in the specified range in the machine configuration file (between 1.0.0.0 and 1.9.9.0), and thus this version redirection affects the linking decision this time. Therefore, the code consumer (or a secure linker of the code consumer) must find another `hashTable` component whose version is the new version specified in the machine configuration file (in this case, 2.0.0.0).

If there exists no machine configuration file or if no redirection in the machine configuration file is effective, the code consumer consults a publisher policy file and then an application configuration file in the same way as stated above. If there is no effective version redirection, the component `compiler` is linked to `hashTable` on which it was initially built.

7.1.2 Translating into our linking logic

In Section 5, we explained how to represent some entities (such as component names) as properties and property requests in our logic. In the same way, version information and redirection requests are coded as *properties* and *property requests*.

```
<configuration>
   <runtime>
      <assemblyBinding  xmlns="run:schemas-microsoft-com:asm.v1">
         <dependentAssembly>
            <assemblyIdentity name="hashTable"/>
            <bindingRedirect oldVersion = "1.0.0.0 - 1.9.9.0"
                             newVersion = "2.0.0.0"/>
         <dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

Figure 9: A configuration file of the .NET framework

$$\frac{\begin{array}{l}\neg isempty(vPolicy)\\ inrange(vPolicy.range, originalVer)\end{array}}{ver\_policy\_effective(vPolicy, orginalVer)}\; vp\_effective$$

$$\frac{\begin{array}{l}ver\_policy\_effective(vrq.mch, vrq.org)\\ version\_eq(vrq.mch.newVer, v)\end{array}}{vrq(v)}\; machine\_level\_redir$$

$$\frac{\begin{array}{l}\neg ver\_policy\_effective(vrq.mch, vrq.org)\\ ver\_policy\_effective(vrq.pub, vrq.org)\\ version\_eq(vrq.pub.newVer, v)\end{array}}{vrq(v)}\; publisher\_level\_redir$$

$$\frac{\begin{array}{l}\neg ver\_policy\_effective(vrq.mch, vrq.org)\\ \neg ver\_policy\_effective(vrq.pub, vrq.org)\\ ver\_policy\_effective(vrq.app, vrq.org)\\ version\_eq(vrq.app.newVer, v)\end{array}}{vrq(v)}\; app\_level\_redir$$

$$\frac{\begin{array}{l}\neg ver\_policy\_effective(vrq.mch, vrq.org)\\ \neg ver\_policy\_effective(vrq.pub, vrq.org)\\ \neg ver\_policy\_effective(vrq.app, vrq.org)\\ version\_eq(vrq.org, v)\end{array}}{vrq(v)}\; no\_redir$$

Figure 10: Inference rules for version redirection

Version information is of type *version*, an alias of type *property*, and built by using the formula constructor *mk_prp_version*. The constructor *mk_prp_version* takes four numbers as arguments, standing for a major version number, a minor version number, a build number and a revision number respectively.

A redirection request is a predicate of type *ver_req*, an alias of type *prp_req*. It takes an argument of type *version* and returns true if the argument satisfies the redirection request it implements. The constructor *mk_prp_req_version* builds a predicate of type *ver_req* out of four input arguments: the original version against which a calling assembly is built, version redirection information from an application configuration file, version redirection information from a publisher policy file, and version redirection information from a machine configuration file.

The version redirection information has the type *ver_policy*, consisting of two parts, affected old versions and a new target version. The old version part can have two alternative formats, specifying either one version, or a range of versions. The formula constructor *mk_ver_policy_simple* is used for building a term of type *ver_policy* out of an old version and a new target version. On the other hand, the formula constructor *mk_ver_policy_intv* is used for building a term of type *ver_policy* out of a range of old versions and a new target version.

To represent a range of versions, we introduce a type *ver_range*. The constructor *mk_ver_range* makes a term of type *ver_range* out of two arguments of type *version*, each of which stands for the beginning of the range and the end of the range respectively.

The linking decision procedure of the .NET framework is translated into a set of lemmas in our linking logic. The inference rules for version redirection are shown in Figure 10.

The inference rule *vp_effective* shows how to check whether or not a given version policy affects finding a target assembly. Suppose that a version `originalVer` is the version of an original assembly to be linked, and there exists a version policy `vPolicy`. If `vPolicy` is not empty and `originalVer` is within the old version range specified in `vPolicy`, then the predicate *ver_policy_effective* holds and `vPolicy` is considered to be effective.

The inference rule *machine_level_redir* shows the case when there exists a version redirection request `vreq`, and the version policy field `mch` of `vreq` is effective. If the version `v` of a target assembly matches the new version in `mch`, then `v` satisfies `vreq`, and the assembly of version `v` is used in the later phase of linking.

Other inference rules, *publisher_level_redir*, *app_level_redir* and *no_redir* show that the version redirection with configurations of low priority. Each rule requires that higher version policies are not effective as well as the version policy of interest is effective. This means a version policy would affect version redirection only when all of the version policies of higher priority were not effective.

Shown in this section, it is possible to encode a security model in our linking logic by expressing its primitives as a set of definitions, and then encoding and proving the processing rules as lemmas. Our linking logic provides a way for describing different security models in a single logic based on a well-defined higher order logic; thus, it enables those security models to interoperate in a principled way.

7.2 Key certificates

Signing codes guarantees the integrity of the codes by digital signatures and key certificates. A certificate is an electronic document used to identify some entities such as an individual, a server or a company to associate that identity with a public key.

In the security model of the .NET framework, a code producer can sign an assembly with his public key. This adds the encrypted public key and the resulting signature to its assembly manifest. Then, the .NET runtime verifies the digital signature of an assembly using the public key in its assembly manifest.

However, signing with a self-announcing public key doesn't provide any level of trust, which enforces the security model to provide a way of reasoning about the hierarchy of trust. As already explained, a code producer can get a certificate of his public key from a third-party key authority after proving his identity to the authority. Since different authorities could use different formats for their certificates, it is inevitable that a secure system must support as many formats as possible for interoperability.

In case of the security model of the .NET framework, it supports several standard public key certificate formats, including X.509. The key certificates are embedded in a specific location in an assembly manifest by the compiler at compile time and then used by a code consumer to authenticate the assembly later.

The assembly-manifest format accommodates a specific set of key certificate formats. Tying public-key infrastructure so closely to version management results in a less flexible system than expected: future users with a different PKI will not be able to take advantage of code signing in .NET. Our linking logic can separate these issues in a more modular way.

We showed already in Section 7.1 that security models of different decision policies can be translated into our linking logic and operate with each other smoothly on the level of the underlying logic. In the remaining part of this section, we will explain how to express the authentication with key certificates independent of underlying linking decision procedures.

To address principal-public key bindings, our linking framework has a formula constructor *keybind* for translating various key certificate formats into a formula in the linking logic. It takes two arguments, the name of a principal and its public key. The formula holds if and only if the second argument is the public key of the first argument.

The statement `keybind(name, pubkey)` is made out of different key certificate formats by the trusted part of a code consumer. Since translating format-specific key certificates is not part of the linking logic, a new key certificate format can be added to the linking framework later without the necessity of changing the linking logic. This increases the scalability of the linking framework. The following is the complete rule for verifying digital signatures.

$$\frac{\begin{array}{l} key\_auth(ca) \\ keybind(ca, caKey) \\ signed(caKey, keybind(pname, pkey)) \\ signed(pkey, stmt) \end{array}}{says(pname, stmt)}\ valid\_sig$$

This rule means that it is believed that the principal `pname` says `stmt` if there exists a trusted key authority `ca`, its public key `caKey`, a key certificate, saying that the public key of the principal `pname` is `pkey`, and `stmt` is signed with a key `pkey`.

Having verified the digital signatures with this inference rule, none of the remaining part of the linking procedure depends on the formulas built by *keybind* or *signed*. By introducing the format-neutral constructor *keybind* and letting only a small part of the logic use digital signature-specific constructors such as *keybind* or *signed*, we can successfully separate the logic of verifying digital signatures from the other parts of the linking logic.

8. CONCLUSION

We have developed a framework for secure linking systems based on PCA. In this scheme, the burden of proving rights to access the shared resources of a code consumer is put on a code producer rather than on the code consumer unlike in traditional distributed authentication frameworks.

In our framework, a code consumer announces its linking policy to protect its system from malicious code from outside. The policy can include properties, for example component names, valid hash code of programs, version information, and so on, which code consumers thinks important for the system safety. To link a software component to other components in a code consumer and to execute it, a provider of the component should submit a proof that the component has the properties required by the code consumer.

Our linking logic consists of basic formula constructors and inference rules based on the logic of PCA. Linking decision procedures of a code consumer, system-specific linking policies, and description of software components of code producers are translated into the linking logic. A proof of secure linking is formed out of the linking logic, and checked by a trusted proof checker in a code consumer. If the accompanying proof is valid, a software component is allowed to be linked to other components in the system of the code consumer.

In addition, adopting higher-order logic of PCA makes our linking logic general and flexible. We showed that our logic is expressive enough to encode the linking procedures of the .NET framework. Due to this expressiveness, it is possible to encode various security models into our logic, and to enable different security models to interoperate conveniently.

Trying to give a formal description of a real-world system gives us insight into the system. We found that it is better to seperate digital signature verification and its key certificate management from other linking decision to increase the scalability and interoperability of the .NET framework.

We introduced a way a code consumer can require high-level properties from outside software components, and check the exported properties of a software component with certificates from third party authorities. Our framework enhances the security of a system during linking time, giving more control to linking decisions.

9. REFERENCES

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[2] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, November 1999.

[3] A. W. Appel and A. P. Felty. Dependent types ensure partial correctness of theorem provers. *Journal of Functional Programming*, Accepted for publication.

[4] L. Bauer, A. W. Appel, and E. W. Felten. Mechanisms for secure modular programming in java. Technical Report CS-TR-603-99, Department of Computer Science, Princeton University, July 1999.

[5] L. Bauer, M. A. Schneider, and E. W. Felten. A general and flexible access-control system for the web. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[6] M. Blume and A. W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21:812–846, 1999.

[7] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40:143–184, January 1993.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addision Wesley, second edition, 1999.

[9] G. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, January 1997.

[10] F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, July 1999.

[11] D. S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 2001.

[12] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

APPENDIX

A. TACTICALS AND TACTICS

A.1 search_assmp

operators signed, keybind, key_authority, prp_server, module_authority, library_dsc

$$\frac{}{\mathit{search\_assmp}\ (A\ \mathit{by}\ P, \Gamma)\ A\ P}\ \mathit{initpf}$$

$$\frac{\mathit{search\_assmp}\ \Gamma\ A\ P}{\mathit{search\_assmp}\ (X, \Gamma)\ A\ P}\ \mathit{init2pf}$$

A.2 fp_string_valid

operator string_valid

$$\frac{}{\mathit{fp\_string\_valid}\ (\mathit{string\_valid}(\mathit{mk\_str1}(\mathit{ch})))\ \mathit{string\_valid\_mk\_str1}}\ \mathit{string\_valid\_mk\_str1\_pf}$$

$$\frac{\mathit{fp\_string\_valid}\ (\mathit{string\_valid}(\mathit{str}))\ P}{\mathit{fp\_string\_valid}\ (\mathit{string\_valid}(\mathit{mk\_str}(\mathit{ch}, \mathit{str})))\ (\mathit{string\_valid\_mk\_str}\ P)}\ \mathit{string\_valid\_mk\_str\_pf}$$

A.3 fp_ide_valid

operator ide_valid

$$\frac{\mathit{fp\_string\_valid}\ (\mathit{string\_valid}(\mathit{str}))\ P}{\mathit{fp\_ide\_valid}\ (\mathit{ide\_valid}(\mathit{mk\_ide}(\mathit{str})))\ (\mathit{ide\_valid\_mk\_ide}\ P)}\ \mathit{ide\_valid\_mk\_ide\_pf}$$

A.4 fp_emptyset

operator isempty

$$\frac{}{\mathit{fp\_emptyset}\ (\mathit{isempty}(\mathit{emptyset}))\ \mathit{set\_isempty\_emptyset}}\ \mathit{set\_empty\_pf}$$

A.5 fp_set

operators set_union, singleton

$$\frac{}{\mathit{fp\_set}\ (X{\in}\mathit{singleton}(X))\ \mathit{singleton\_elem}}\ \mathit{singleton\_refl\_pf}$$

$$\frac{\mathit{fp\_set}\ (X{\in}S1)\ P}{\mathit{fp\_set}\ (X{\in}\mathit{set\_union}(S1, S2))\ (\mathit{set\_union\_i1}\ P)}\ \mathit{set\_union1\_pf}$$

$$\frac{\mathit{fp\_set}\ (X{\in}S2)\ P}{\mathit{fp\_set}\ (X{\in}\mathit{set\_union}(S1, S2))\ (\mathit{set\_union\_i2}\ P)}\ \mathit{set\_union2\_pf}$$

A.6 fp_validper

operator validper

$$\frac{}{\mathit{fp\_validper}\ (\mathit{validper}(\mathit{module\_eq}))\ \mathit{module\_eq\_validper}}\ \mathit{module\_eq\_valdiper\_pf}$$

$$\frac{}{\mathit{fp\_validper}\ (\mathit{validper}(\mathit{set\_equiv}))\ \mathit{set\_equiv\_validper}}\ \mathit{set\_equiv\_validper\_pf}$$

A.7 fp_validper_refl

operators module_eq, set_equiv

$$\frac{}{\mathit{fp\_validper\_refl}\ (\mathit{module\_eq}(m, m))\ \mathit{module\_eq\_refl}}\ \mathit{module\_eq\_refl\_pf}$$

$$\frac{}{\mathit{fp\_validper\_refl}\ (\mathit{set\_equiv}(s, s))\ \mathit{set\_equiv\_refl}}\ \mathit{set\_equiv\_refl\_pf}$$

A.8 fp_list_nil

operator list_is_nil

$$\frac{}{\mathit{fp\_list\_nil}\ (\mathit{list\_is\_nil}(\mathit{list\_nil}))\ \mathit{list\_is\_nil\_nil}}\ \mathit{list\_is\_nil\_nil\_pf}$$

A.9 fp_list_valid

operator list_valid

$$\frac{\begin{array}{c}\mathit{fp\_validper}\ \mathit{validper}(\mathit{eq})\ P1\\ \mathit{fp\_list\_nil}\ (\mathit{list\_is\_nil}(l))\ P2\end{array}}{\mathit{fp\_list\_valid}\ (\mathit{list\_valid}(\mathit{eq},l))\ (\mathit{list\_valid\_nil}\ P1\ P2)}\ \mathit{list\_valid\_nil\_pf}$$

$$\frac{\begin{array}{c}\mathit{fp\_validper\_refl}\ (\mathit{eq}(\mathit{hd},\mathit{hd}))\ P1\\ \mathit{fp\_list\_valid}\ (\mathit{list\_valid}(\mathit{eq},\mathit{tl}))\ P2\end{array}}{\mathit{fp\_list\_valid}\ (\mathit{list\_valid}(\mathit{eq},(\mathit{list\_cons}(\mathit{eq},\mathit{list\_cons}(\mathit{eq},\mathit{hd},\mathit{tl})))))\ (\mathit{list\_valid\_cons}\ P1\ P2)}\ \mathit{list\_valid\_cons\_pf}$$

A.10 fp_prp_valid

operator prp_valid

$$\frac{\mathit{fp\_ide\_valid}\ (\mathit{ide\_valid}(\mathit{name}))\ P}{\mathit{fp\_prp\_valid}\ (\mathit{prp\_valid}(\mathit{mk\_prp}(\mathit{name})))\ (\mathit{mk\_prp\_valid}\ P)}\ \mathit{mk\_prp\_valid\_pf}$$

A.11 fp_prp_match

operators mk_prp_req, rq_component_name_exists, mk_rq_component_name, rq_hash_code_checked, rq_export_id_exsits

$$\frac{\mathit{fp\_prp\_valid}\ (\mathit{prp\_valid}(\mathit{mk\_prp}(\mathit{pname})))\ P}{\mathit{fp\_prp\_match}\ (\mathit{mk\_prp\_req}(\mathit{mk\_prp}(\mathit{pname}),\mathit{mk\_prp}(\mathit{pname})))\ (\mathit{mk\_prp\_req\_refl}\ P)}\ \mathit{mk\_req\_refl\_pf}$$

$$\frac{}{\mathit{fp\_prp\_match}\ (\mathit{rq\_component\_name\_exists}(\mathit{mk\_cname}(n)))\ \mathit{req\_mk\_cname}}\ \mathit{req\_mk\_cname\_pf}$$

$$\frac{\mathit{fp\_ide\_valid}\ (\mathit{ide\_valid}(n))\ P}{\mathit{fp\_prp\_match}\ (\mathit{mk\_rq\_component\_name}(\mathit{mk\_cname}(n),\mathit{mk\_cname}(n)))\ (\mathit{req\_mk\_cname}'\ P)}\ \mathit{req\_mk\_cname}'\mathit{\_pf}$$

$$\frac{}{\mathit{fp\_prp\_match}\ (\mathit{rq\_hashcode\_checked}(\mathit{mk\_hash\_prp}(h)))\ \mathit{req\_mk\_hash\_prp}}\ \mathit{req\_mk\_hash\_prp\_pf}$$

$$\frac{}{\mathit{fp\_prp\_match}\ (\mathit{rq\_export\_id\_exists}(\mathit{mk\_ide\_prp}(i)))\ \mathit{req\_mk\_ide\_prp}}\ \mathit{req\_mk\_ide\_prp\_pf}$$

A.12 fp_cdsc_valid

operator component_dsc_valid

$$\frac{\mathit{fp\_list\_valid}\ (\mathit{list\_valid}(\mathit{set\_equiv},\mathit{imp}))\ P}{\mathit{fp\_cdsc\_valid}\ (\mathit{component\_dsc\_valid}(\mathit{mk\_cdsc}(\mathit{exp},\mathit{imp})))\ (\mathit{mk\_cdsc\_valid}\ P)}\ \mathit{mk\_cdsc\_valid\_pf}$$

$$\frac{\begin{array}{c}\mathit{fp\_cdsc\_valid}\ (\mathit{component\_dsc\_valid}(d1))\ P1\\ \mathit{fp\_cdsc\_valid}\ (\mathit{component\_dsc\_valid}(d2))\ P2\end{array}}{\mathit{fp\_cdsc\_valid}\ (\mathit{component\_dsc\_valid}(\mathit{cdsc\_combine}(d1,d2)))\ (\mathit{cdsc\_valid}\ P1\ P2)}\ \mathit{cdsc\_combine\_valid\_pf}$$

A.13 fp_signed_keybind

operator says and keybind

$$\frac{\begin{array}{l}\mathit{search\_assmp}\ \Gamma\ (\mathit{keybind}(\mathit{ca},\mathit{caKey}))\ P1\\ \mathit{search\_assmp}\ \Gamma\ (\mathit{signed}(\mathit{caKey},\mathit{keycert}(\mathit{prn},\mathit{pkey},\mathit{pinfo})))\ P2\end{array}}{\mathit{fp\_signed\_keybind}\ \Gamma\ (\mathit{says}(\mathit{ca},\mathit{keybind}(\mathit{prn},\mathit{pkey})))\ (\mathit{signed\_keycert\_e}\ P1\ P2)}\ \mathit{keycert\_pf}$$

A.14 fp_signed_auth

operator says and prp_auth

$$\frac{\begin{array}{l}\mathit{fp\_signed\_keybind}\ \Gamma\ (\mathit{says}(\mathit{ca},(\mathit{keybind}(\mathit{auth},\mathit{authKey})))))\ P1\\ \mathit{search\_assmp}\ \Gamma\ (\mathit{signed}(\mathit{authKey},\mathit{prp\_auth}(\mathit{prn},\mathit{rset})))\ P2\\ \mathit{search\_assmp}\ \Gamma\ (\mathit{key\_authority}(\mathit{ca}))\ P3\end{array}}{\mathit{fp\_signed\_auth}\ \Gamma\ (\mathit{says}(\mathit{auth},\mathit{prp\_auth}(\mathit{prn},\mathit{rset})))\ (\mathit{signed\_stmt\_e}\ P1\ P2\ P3)}\ \mathit{prp\_auth\_pf}$$

A.15 fp_signed_dsc

operator says and module_dsc

$$\frac{\begin{array}{l} \mathit{fp\_signed\_keybind}\ \Gamma\ (\mathit{says}(\mathit{ca},(\mathit{keybind}(\mathit{auth},\mathit{authKey})))))\ P1 \\ \mathit{search\_assmp}\ \Gamma\ (\mathit{signed}(\mathit{authKey},\mathit{module\_dsc}(m,\mathit{dsc})))\ P2 \\ \mathit{search\_assmp}\ \Gamma\ (\mathit{key\_authority}(\mathit{ca}))\ P3 \end{array}}{\mathit{fp\_signed\_assmp}\ \Gamma\ (\mathit{says}(\mathit{auth},\mathit{module\_dsc}(m,\mathit{dsc})))\ (\mathit{signed\_stmt\_e}\ P1\ P2\ P3)}\ \mathit{component\_dsc\_pf}$$

A.16 fp_valid_sig_auth

operator valid_sig_prp_auth

$$\frac{\begin{array}{l} \mathit{search\_assmp}\ \Gamma\ (\mathit{key\_authority}(\mathit{ca}))\ P1 \\ \mathit{search\_assmp}\ \Gamma\ (\mathit{keybind}(\mathit{ca},\mathit{caKey}))\ P2 \\ \mathit{search\_assmp}\ \Gamma\ (\mathit{prp\_server}(\mathit{pa}))\ P3 \\ \mathit{fp\_signed\_keybind}\ \Gamma\ (\mathit{says}(\mathit{ca},\mathit{keybind}(\mathit{pa},\mathit{paKey})))\ P4 \\ \mathit{fp\_signed\_auth}\ \Gamma\ (\mathit{says}(\mathit{pa},\mathit{prp\_auth}(\mathit{ma},\mathit{prp})))\ P5 \end{array}}{\mathit{fp\_valid\_sig\_auth}\ \Gamma\ (\mathit{valid\_sig\_prp\_auth}(\mathit{ma},\mathit{prp}))\ (\mathit{vsig\_auth\_i}\ P1\ P2\ P3\ P4\ P5)}\ \mathit{valid\_sig\_auth\_pf}$$

A.17 fp_valid_sig_cdsc

operator valid_sig_component_dsc

$$\frac{\begin{array}{l} \mathit{search\_assmp}\ \Gamma\ (\mathit{key\_authority}(\mathit{ca}))\ P1 \\ \mathit{search\_assmp}\ \Gamma\ (\mathit{keybind}(\mathit{ca},\mathit{caKey}))\ P2 \\ \mathit{search\_assmp}\ \Gamma\ (\mathit{module\_authority}(\mathit{ma}))\ P3 \\ \mathit{fp\_signed\_keybind}\ \Gamma\ (\mathit{says}(\mathit{ca},(\mathit{keybind}(\mathit{ma},\mathit{maKey}))))\ P4 \\ \mathit{fp\_signed\_dsc}\ \Gamma\ (\mathit{says}(\mathit{ma},\mathit{module\_dsc}(m,\mathit{dsc})))\ P5 \end{array}}{\mathit{fp\_valid\_sig\_cdsc}\ \Gamma\ (\mathit{valid\_sig\_component\_dsc}(\mathit{ma},m,\mathit{dsc}))\ (\mathit{valid\_sig\_cdsc\_i}\ P1\ P2\ P3\ P4\ P5)}\ \mathit{valid\_sig\_cdsc\_pf}$$

A.18 fp_signed_by_ma

operator signed_by_ma

$$\frac{\begin{array}{l} \mathit{fp\_valid\_sig\_auth}\ \Gamma\ (\mathit{valid\_sig\_prp\_auth}(\mathit{ma},\mathit{rqset}))\ P2 \\ \mathit{fp\_valid\_sig\_cdsc}\ \Gamma\ (\mathit{valid\_sig\_component\_dsc}(\mathit{ma},m,\mathit{dsc}))\ P3 \\ \mathit{fp\_set}\ (\mathit{rq}{\in}\mathit{rqset})\ P1 \end{array}}{\mathit{fp\_signed\_by\_ma}\ \Gamma\ (\mathit{signed\_by\_ma}(m,\mathit{dsc},\mathit{rq}))\ (\mathit{sma\_i}\ \mathit{ma}\ \mathit{rqset}\ P1\ P2\ P3)}\ \mathit{signed\_by\_ma\_pf}$$

A.19 fp_as_cdsc_combine

operator signed_rq

$$\frac{\begin{array}{l} \mathit{fp\_cdsc\_valid}\ (\mathit{component\_dsc\_valid}(d1))\ P1 \\ \mathit{fp\_cdsc\_valid}\ (\mathit{component\_dsc\_valid}(d2))\ P2 \\ \mathit{fp\_as\_cdsc\_combine}\ \Gamma\ (\mathit{signed\_rq}(m,d1,\mathit{rq}))\ P3 \end{array}}{\mathit{fp\_as\_cdsc\_combine}\ \Gamma\ (\mathit{signed\_rq}(m,\mathit{cdsc\_combine}(d1,s2),\mathit{rq}))\ (\mathit{as\_cdsc\_cmbn\_i}1\ P1\ P2\ P3)}\ \mathit{sig\_rq\_cdsc\_combine\_i}1\mathit{\_pf}$$

$$\frac{\begin{array}{l} \mathit{fp\_cdsc\_valid}\ (\mathit{component\_dsc\_valid}(d1))\ P1 \\ \mathit{fp\_cdsc\_valid}\ (\mathit{component\_dsc\_valid}(d2))\ P2 \\ \mathit{fp\_as\_cdsc\_combine}\ \Gamma\ (\mathit{signed\_rq}(m,d2,\mathit{rq}))\ P3 \end{array}}{\mathit{fp\_as\_cdsc\_combine}\ \Gamma\ (\mathit{signed\_rq}(m,\mathit{cdsc\_combine}(d1,s2),\mathit{rq}))\ (\mathit{as\_cdsc\_cmbn\_i}2\ P1\ P2\ P3)}\ \mathit{sig\_rq\_cdsc\_combine\_i}2\mathit{\_pf}$$

$$\frac{\mathit{fp\_signed\_by\_ma}\ \Gamma\ (\mathit{signed\_by\_ma}(m,\mathit{mk\_cdsc}(\mathit{exp},\mathit{imp}),\mathit{rq}))\ P}{\mathit{fp\_as\_cdsc\_combine}\ \Gamma\ (\mathit{signed\_rq}(m,\mathit{mk\_cdsc}(\mathit{exp},\mathit{imp}),\mathit{rq}))\ (\mathit{as\_singleton}\ P)}\ \mathit{sig\_rq\_sng\_pf}$$

A.20 fp_as_set_union

operator all_signed

$$\frac{\mathit{fp\_emptyset}\ (\mathit{isempty}(s))\ P}{\mathit{fp\_as\_set\_union}\ \Gamma\ (\mathit{all\_signed}(m,\mathit{dsc},s))\ (\mathit{as\_emptyset}\ P)}\ \mathit{all\_sig\_emptyset\_pf}$$

$$\frac{\begin{array}{l} \mathit{fp\_as\_set\_union}\ \Gamma\ (\mathit{all\_signed}(m,\mathit{dsc},s1))\ P1 \\ \mathit{fp\_as\_set\_union}\ \Gamma\ (\mathit{all\_signed}(m,\mathit{dsc},s2))\ P2 \end{array}}{\mathit{fp\_as\_set\_union}\ \Gamma\ (\mathit{all\_signed}(m,\mathit{dsc},\mathit{set\_union}(s1,s2)))\ (\mathit{as\_union}\ P1\ P2)}\ \mathit{all\_sig\_set\_union\_pf}$$

$$\frac{\mathit{fp\_as\_cds\_combine}\ \Gamma\ (\mathit{signed\_rq}(m,\mathit{dsc},\mathit{rq})\ P}{\mathit{fp\_as\_set\_union}\ \Gamma\ (\mathit{all\_signed}(m,\mathit{dsc},\mathit{singleton}(\mathit{rq})))\ (\mathit{all\_signed\_i}P)}\ \mathit{all\_sig\_pf}$$

A.21 fp_valid_cdsc

operator signed_component_dsc

$$\frac{\begin{array}{l} fp\_cdsc\_valid\ (component\_dsc\_valid(dsc))\ P1 \\ fp\_as\_set\_union\ \Gamma\ (all\_signed(m, dsc, rqset))\ P2 \end{array}}{fp\_valid\_cdsc\ \Gamma\ (signed\_component\_dsc(m, dsc, rqset))\ (valid\_cdsc\_i2'\ P1\ P2)}\ valid\_cdsc\_pf$$

A.22 fp_valid_lib

operator valid_library

$$\frac{\begin{array}{l} fp\_list\_nil\ (list\_is\_nil(lib))\ P1 \\ fp\_list\_nil\ (list\_is\_nil(libdsc))\ P2 \end{array}}{fp\_valid\_lib\ \Gamma\ (valid\_library(lib, libdsc))\ (valid\_library\_nil\ P1\ P2)}\ valid\_lib\_nil\_pf$$

$$\frac{\begin{array}{l} fp\_valid\_lib\ \Gamma\ (valid\_library(libtl, libdsctl))\ P1 \\ search\_assmp\ \Gamma\ (library\_dsc(m, dsc))\ P2 \\ where\ lib = list\_cons(modules\_eq, m, libtl)\ and\ libdsc = list\_cons(set\_equiv, dsc, libdsctl) \end{array}}{fp\_valid\_lib\ \Gamma\ (valid\_library(lib, listdsc))\ (valid\_library\_cons\ P1\ P2)}\ valid\_lib\_cons\_pf$$

A.23 fp_has_prps_set_union $'$

operator has_prp

$$\frac{fp\_has\_prps\_set\_union'\ (has\_prp(rq, p1))\ P}{fp\_has\_prps\_set\_union'\ (has\_prp(rq, set\_union(p1, p2)))\ (has\_prp\_set\_union\_i1\ P)}\ has\_prp\_union1\_pf$$

$$\frac{fp\_has\_prps\_set\_union'\ (has\_prp(rq, p2))\ P}{fp\_has\_prps\_set\_union'\ (has\_prp(rq, set\_union(p1, p2)))\ (has\_prp\_set\_union\_i2\ P)}\ has\_prp\_union2\_pf$$

$$\frac{fp\_prp\_match\ (prp\_match(rq, prp))\ P}{fp\_has\_prps\_set\_union'\ (has\_prp(rq, singleton(prp)))\ (has\_prp\_singleton\_i\ P)}\ has\_prps\_singleton\_pf$$

A.24 fp_has_prps_set_union

operator has_property

$$\frac{fp\_emptyset\ (isempty(rqset))\ P}{fp\_has\_prps\_set\_union\ (has\_property(rqset, prpset))\ (has\_proeprty\_emptyset\ P)}\ has\_prps\_emptyset\_pf$$

$$\frac{\begin{array}{l} fp\_has\_prps\_set\_union\ (has\_property(r1, prpset))\ P1 \\ fp\_has\_prps\_set\_union\ (has\_property(r2, prpset))\ P2 \end{array}}{fp\_has\_prps\_set\_union\ (has\_property(set\_union(r1, r2), prpset))\ (has\_proeprty\_set\_union\_i3\ P1\ P2)}\ has\_prps\_set\_union\_pf$$

$$\frac{fp\_has\_prps\_set\_union'\ (has\_prp(rq, prpset)\ P}{fp\_has\_prps\_set\_union\ (has\_property(singleton(rq), prpset)\ (has\_property\_i'\ P)}\ has\_prps\_pf$$

A.25 fp_exp_prps$'$

operator exp_rprp

$$\frac{fp\_exp\_prps'\ (exp\_rprp(rq, c1))\ P}{fp\_exp\_prps'\ (exp\_rprp(rq, (cdsc\_combine(c1, c2))))\ (exp\_rprp\_cdsc\_combine\_i1\ P)}\ exp\_rprp\_cdsc\_i1\_pf$$

$$\frac{fp\_exp\_prps'\ (exp\_rprp(rq, c2))\ P}{fp\_exp\_prps'\ (exp\_rprp(rq, (cdsc\_combine(c1, c2))))\ (exp\_rprp\_cdsc\_combine\_i2\ P)}\ exp\_rprp\_cdsc\_i2\_pf$$

$$\frac{fp\_has\_prps\_set\_union'\ (has\_prp(rq, exp))\ P}{fp\_exp\_prps'\ (exp\_rprp(rq, mk\_cdsc(exp, imp)))\ (exp\_rprp\_i'\ P)}\ exp\_rprp\_cdsc\_pf$$

A.26 fp_exp_prps

operator export_required_prps

$$\frac{\mathit{fp\_emptyset}\ (\mathit{isempty}(\mathit{rset}))\ P}{\mathit{fp\_exp\_prps}\ (\mathit{export\_required\_prps}(\mathit{rset}, \mathit{dsc}))\ (\mathit{exp\_rprps\_empty}\ P)}\ \mathit{export\_rprps\_empty\_pf}$$

$$\frac{\begin{array}{l}\mathit{fp\_exp\_prps}\ (\mathit{export\_required\_prps}(r1, \mathit{dsc}))\ P1\\ \mathit{fp\_exp\_prps}\ (\mathit{export\_required\_prps}(r2, \mathit{dsc}))\ P1\end{array}}{\mathit{fp\_exp\_prps}\ (\mathit{export\_required\_prps}(\mathit{set\_union}(r1, r2), \mathit{dsc}))\ (\mathit{exp\_rprps\_set\_union}\ P1\ P2)}\ \mathit{export\_rprps\_set\_union\_pf}$$

$$\frac{\mathit{fp\_exp\_prps}'\ (\mathit{exp\_rprp}(\mathit{rq}, \mathit{dsc}))\ P}{\mathit{fp\_exp\_prps}\ (\mathit{export\_required\_prps}(\mathit{singleton}(\mathit{rq}), \mathit{dsc}))\ (\mathit{exp\_rprps\_i}\ P)}\ \mathit{export\_rprps\_sng\_pf}$$

A.27 fp_st_imprt_cdsc

operator satisfy_imprt_cdsc

$$\frac{\mathit{fp\_has\_prps\_set\_union}\ (\mathit{has\_property}(\mathit{imp}, \mathit{exp}))\ P}{\mathit{fp\_st\_imprt\_cdsc}\ (\mathit{satisfy\_imprt\_cdsc}(\mathit{imp}, \mathit{list\_cons}(\mathit{set\_equiv}, \mathit{exp}, \mathit{explist})))\ (\mathit{satisfy\_imprt\_cdsc\_hd}\ P)}\ \mathit{satisfy\_imprt\_hd\_pf}$$

$$\frac{\begin{array}{l}\mathit{fp\_list\_valid}\ (\mathit{list\_valid}(\mathit{set\_equiv}, \mathit{explist}))\ P1\\ \mathit{fp\_st\_imprt\_cdsc}\ (\mathit{satisfy\_imprt\_cdsc}(\mathit{imp}, \mathit{explist}))\ P2\end{array}}{\mathit{fp\_st\_imprt\_cdsc}\ (\mathit{satisfy\_imprt\_cdsc}(\mathit{imp}, \mathit{list\_cons}(\mathit{set\_equiv}, \mathit{exp}, \mathit{explist})))\ (\mathit{satisfy\_imprt\_cdsc\_tl}\ P1\ P2)}\ \mathit{satisfy\_imprt\_tl\_pf}$$

A.28 fp_st_imprt

operator satisfy_import_req

$$\frac{\mathit{fp\_list\_nil}\ (\mathit{list\_is\_nil}(\mathit{implist}))\ P}{\mathit{fp\_st\_imprt}\ (\mathit{satisfy\_import\_req}(\mathit{implist}, \mathit{libdsc}))\ (\mathit{stsf\_imprt\_nil}\ P)}\ \mathit{satisfy\_imprt\_nil\_pf}$$

$$\frac{\begin{array}{l}\mathit{fp\_st\_imprt\_cdsc}\ (\mathit{satisfy\_imprt\_cdsc}(\mathit{imp}, \mathit{libdsc}))\ P1\\ \mathit{fp\_st\_imprt}\ (\mathit{satisfy\_import\_req}(\mathit{imptl}, \mathit{libdsc}))\ P2\\ \mathit{where}\ \mathit{implist}\ \ =\ \ \mathit{list\_cons}(\mathit{eq}, \mathit{set\_equiv}, \mathit{imp}, \mathit{imptl})\end{array}}{\mathit{fp\_st\_imprt}\ (\mathit{satisfy\_import\_req}(\mathit{implist}, \mathit{libdsc}))\ (\mathit{stsf\_imprt\_cons}\ P2\ P1)}\ \mathit{satisfy\_imprt\_cons\_pf}$$

A.29 fp_prv_prps

operator provide_enough_prps

$$\frac{\begin{array}{l}\mathit{fp\_list\_valid}\ (\mathit{list\_valid}(\mathit{set\_equiv}, \mathit{imp}))\ P1\\ \mathit{fp\_valid\_lib}\ \Gamma\ (\mathit{valid\_library}(\mathit{lib}, \mathit{libdsc}))\ P2\\ \mathit{fp\_st\_imprt}\ (\mathit{satisfy\_import\_req}(\mathit{imp}, \mathit{libdsc}))\ P3\end{array}}{\mathit{fp\_prv\_prps}\ \Gamma\ (\mathit{provide\_enough\_prps}(\mathit{mk\_dsc}(\mathit{exp}, \mathit{imp}), \mathit{lib}, \mathit{libdsc}))\ (\mathit{prv\_prps\_i}\ P1\ P2\ P3)}\ \mathit{prv\_prps\_pf}$$

$$\frac{\begin{array}{l}\mathit{fp\_cdsc\_valid}\ (\mathit{component\_dsc\_valid}(c1))\ P1\\ \mathit{fp\_cdsc\_valid}\ (\mathit{component\_dsc\_valid}(c2))\ P2\\ \mathit{fp\_prv\_prps}\ \Gamma\ (\mathit{provide\_enough\_prps}(c1, \mathit{lib}, \mathit{libdsc}))\ P3\\ \mathit{fp\_prv\_prps}\ \Gamma\ (\mathit{provide\_enough\_prps}(c2, \mathit{lib}, \mathit{libdsc}))\ P4\end{array}}{\mathit{fp\_prv\_prps}\ \Gamma\ (\mathit{provide\_enough\_prps}(\mathit{cdsc\_combine}(c1, c2), \mathit{lib}, \mathit{libdsc}))\ (\mathit{prv\_combine}\ P1\ P2\ P3\ P4)}\ \mathit{prv\_prps\_cdsc\_combine\_pf}$$

A.30 findproof

operator ok_to_link

$$\frac{\begin{array}{l}\mathit{fp\_valid\_cdsc}\ \Gamma\ (\mathit{signed\_component\_dsc}(m, \mathit{dsc}, \mathit{rqset}))\ P1\\ \mathit{fp\_prv\_prps}\ \Gamma\ (\mathit{provide\_enough\_prps}(\mathit{dsc}, \mathit{lib}, \mathit{libdsc}))\ P2\\ \mathit{fp\_exp\_prps}\ (\mathit{export\_required\_prps}(\mathit{rqset}, \mathit{dsc}))\ P3\end{array}}{\mathit{findproof}\ \Gamma\ (\mathit{ok\_to\_link}(m, \mathit{dsc}, \mathit{lib}, \mathit{libdsc}, \mathit{rqset}))\ (\mathit{ok\_to\_link\_i}\ P1\ P2\ P3)}\ \mathit{ok\_to\_link\_pf}$$

B. TERMINATION OF THE PROVER

In this section, we prove Theorem 1 by showing that all the tacticals on which the tactic *ok_to_link_pf* depends terminate. In other words, we show the termination of the call to the tactical *findproof* by showing calling the dependent tacticals, *fp_prv_prps*, *fp_valid_lib* and *fp_exp_prps* always terminate, and so do other recursively dependent tacticals.

B.1 Tactical search_assmp

The tactical *search_assmp* takes two inputs and one output: given a list of assumptions and a formula to be proven, it returns the derivation of the formula if exists. Otherwise it returns failure.

There are two tactics calling the tactical *search_assmp*, the tactic *initpf* and the tactic *init2pf*. In case of the tactic *initpf*, it always terminates since it doesn't require any further calls to other tacticals.

The termination of the tactic *init2pf* is proved by induction on the length of the assumption list. The base case is a list of length zero. Since there is no assumption to search, the tactic *init2pf* always returns failure. Hence, calling to the tactic *init2pf* terminates in the base case.

For the inductive step, suppose that calling to the tactic *init2pf* with a list Γ of length k terminates. Then calling to the tactic *init2pf* with a list $(\mathrm{X}\ ,\ \Gamma)$ of length $k+1$ is always reduced to the call to the tactic *init2pf* with a list Γ of length k. By the induction hypothesis, calling to the tactic *init2pf* with Γ of length k terminates. Therefore calling to the tactic *init2pf* with a list $(\mathrm{X}\ ,\ \Gamma)$ of length $k+1$ always terminates.

By induction, the tactic *init2pf* always terminates. Since the tactics *initpf* and *init2pf* always terminate, the tactical *search_assmp* always terminates.

B.2 Tactical fp_string_valid

Given a formula of the formula constructor *string_valid*, the tactical *fp_string_valid* returns the derivation of the formula.

The formula constructor *string_valid* accepts a formula of type *string* as input. There exist two formula constructors for type *string*:

- $\mathsf{mk\_str1} : \mathsf{char} \to \mathsf{string}$.
- $\mathsf{mk\_str} : \mathsf{char} \to \mathsf{string} \to \mathsf{string}$.

The formula constructor *mk_str1* is used to construct a formula of type *string* from a formula of type *char*. The only way to construct a bigger formula of type *string* from a formula of type *string* is using the constructor *mk_str*. The constructor *mk_str* returns a new *string* formula after adding the input *char* formula to the head of another input *string* formula.

The termination of the tactical *fp_string_valid* can be proved by induction on the structure of an input formula of the formula constructor *string_valid*. The base case occurs when the input formula is formed by the formula constructor *mk_str1*. This is the case when the tactic *string_valid_mk_str_pf* is called. The call to the tactic *string_valid_mk_str1_pf* always terminates since it does not require any further call to other tacticals.

For the inductive step, suppose that calling the tactical *fp_string_valid* with a formula $\mathsf{string\_valid(str)}$ terminates. Calling the tactical *fp_string_valid* with the formula $\mathsf{string\_valid(mk\_str(ch,\ str))}$ is always reduced to calling the tactical *fp_string_valid* with a formula $\mathsf{string\_valid(str)}$ by the tactic *string_valid_mk_str_pf.* By the inductive hypothesis, calling the tactical *fp_string_valid* with a formula $\mathsf{string\_valid(str)}$ always terminates.

Hence, calling the tactical *fp_string_valid* with a formula formed by the constructor *string_valid* always terminates by induction.

B.3 Tactical fp_ide_valid

The tactical *fp_ide_valid* returns the derivation of a formula formed by the formula constructor *ide_valid.*

The formula constructor *ide_valid* takes a formula of type *ide* as input, and there exists only one formula constructor in the linking logic which can build a formula of type *ide.* The formula constructor *mk_ide* constructs a formula of type *ide* from a formula of *string*: $\mathsf{mk\_ide} : \mathsf{string} \to \mathsf{ide}$.

For a formula of the formula constructor *ide_valid* and the formula constructor *mk_ide*, the tactic *ide_valid_mk_ide_pf* is the only matching tactic, and applying *ide_valid_mk_ide_pf* is always reduced to calling the tactical *fp_string_valid* with a formula $\mathsf{string\_valid(str)}$.

Calling the tactical *fp_string_valid* with a formula $\mathsf{string\_valid(str)}$ always terminates as proved in the previous section. Hence calling the tactical *fp_ide_valid* with a formula formed by the formula constructor *ide_valid* always terminates.

B.4 Tactical fp_emptyset

Since it doesn't spawn any further calling to other tacticals, calling the tactical *fp_emptyset* always terminates.

B.5 Tactical fp_set

Given a formula formed by the formula constructors *singleton* and *set_union*, the tactical *fp_set* returns the derivation of the formula. The formula constructor *singleton* is used to make a set of only one element and the formula constructor *set_union* is used to make a union set out of two sets.

The termination of the tactical *fp_set* is proved by induction on the structure of the input formula of the formula constructor *singleton* and *set_union.*

The base case occurs when the set consists of only one element. Since the formula constructor *singleton* is used to make a singleton set, the only applicable tactic is the tactic *singleton_refl_pf.* Since *singleton_refl_pf* does not spawn callings to other tacticals, calling the tactical *fp_set* terminates in the base case.

For the inductive step, suppose that calling the tactical *fp_set* terminates with a formula $\mathsf{X} \in \mathsf{S1}$ and a formula $\mathsf{X} \in \mathsf{S2}$. Calling the tactical *fp_set* with a formula $\mathsf{X} \in \mathsf{set\_union(S1, S2)})$ matches one of the following two different cases.

B.5.1 Tactic set_union1_pf

If the tactic *set_union1_pf* is applied to the formula, the calling is reduced to calling the tactical *fp_set* with the formula X ∈ S1. By the inductive hypothesis, it terminates.

B.5.2 Tactic set_union2_pf

If the tactic *set_union2_pf* is applied to the formula, the calling is reduced to to calling the tactical *fp_set* with the formula X ∈ S2. By the inductive hypothesis, it terminates. Therefore calling the tactical *fp_set* terminates in the inductive step.

By induction, calling the tactical *fp_set* with a formula of formula constructors *singleton* and *set_union* always terminates.

B.6 Tactical fp_validper

Calling the tactical *fp_validper* is axiomatic in the sense that it doesn't cause any further calling to other tacticals. Given the formula validper(module_eq), calling *fp_validper* always returns a proof and terminates. Similarly, given the formula validper(set_equiv), calling *fp_validper* always returns a proof and terminates.

B.7 Tactical fp_validper_refl

Calling the tactical *fp_validper_refl* is also axiomatic. Given the formula module_eq(m,m) and the formula set_equiv(s,s), calling *fp_validper_refl* always returns a proof and terminates.

B.8 Tactical fp_list_nil

Given the formula list_is_nil(list_nil), the tactical fp_list_nil always returns a proof without calling other tacticals by the tactic *list_is_nil_nil_pf*. Thus, calling fp_list_nil always terminates.

B.9 Tactical fp_list_valid

The tactical *fp_list_valid* returns a derivation of a formula formed by the formula constructor *list_valid*. The input of *list_valid* is a formula standing for a list for a given type T. There exist 2 formula constructors for type *list T* :

- list_nil : list T.
- list_cons : T → (list T) → (list T).

The constructor *list_nil* returns a nil list, and the constructor *list_cons* returns a new list after adding the given head element to the tail list.

The termination of the tactical *fp_list_valid* is proved by induction on the structure of the input formula of *list_valid*.

The base case occurs when the input of *list_valid* is a nil list. Only the tactic *list_valid_nil_pf* is applicable for the base case. Calling the tactical *fp_list_valid* with an input formula list_valid(eq,l) is reduced to calling the tactical *fp_validper* with a formula validper(eq) and calling the tactical *fp_list_nil* with a formula list_is_nil(l). Since calling *fp_validper* and calling *fp_list_nil* terminate as proved earlier, applying the tactic *list_valid_nil_pf* terminates. Hence, calling the tactical *fp_list_valid* terminates in the base case.

For the inductive step, suppose that calling the tactical *fp_list_valid* with a formula list_valid(eq, tl) terminates when l = list_cons(eq, hd, tl). Then calling the tactical *fp_list_valid* with the formula list_valid(eq, l) is always reduced to calling the tactical *fp_validper_refl* with a formula eq(hd, hd) and calling the tactical *fp_list_valid* with a formula list_valid(eq,tl). Calling *fp_validper_refl* always terminates as proved earlier, and calling *fp_list_valid* with a formula list_valid(eq,tl) terminates by the inductive hypothesis. So calling the tactical *fp_list_valid* terminates in the inductive step.

By induction, calling the tactical *fp_list_valid* always terminates.

B.10 Tactical fp_prp_valid

Calling the tactical *fp_prp_valid* with a formula formed by the formula constructor *prp_valid* is always reduced to calling the tactical *fp_ide_valid* with a formula formed by the formula constructor *ide_valid*. Since calling the tactical *fp_ide_valid* with a formula formed by *ide_valid* always terminates as proved earlier, calling the tactical *fp_prp_valid* with a formula formed by *prp_valid* always terminates.

B.11 Tactical fp_prp_match

The tactical *fp_prp_match* is called with five different formula constructors, *mk_prp_req*, *rq_component_name_exists*, *mk_rq_component_name*, *rq_hashcode_checked* and *rq_export_id_exists*. There exist five tactics corresponding to each formula constructor respectively.

B.11.1 Tactic mk_req_refl_pf

Calling the tactical *fp_prp_match* with a formula formed by *mk_prp_req* is always reduced to calling the tactical *fp_prp_valid* with a formula formed by *prp_valid*.

As already shown, calling *fp_prp_valid* with a formula of *prp_valid* always terminates. Therefore applying the tactical *mk_req_refl_pf* always terminates.

B.11.2 Tactic req_mk_cname_pf

Calling *fp_prp_match* with a formula formed by the formula constructor *rq_component_name_exists* doesn't require any further calls to other tacticals. Hence applying the tactic *req_mk_cname_pf* always terminates.

B.11.3 Tactic req_mk_cname′_pf

Calling *fp_prp_match* with a formula formed by *mk_rq_component_name* is always reduced to calling the tactical *fp_ide_valid* with a formula of the formula constructor *ide_valid*. As shown already, calling *fp_ide_valid* with a formula of *ide_valid* always terminates. Therefore, applying the tactic *req_mk_cname′_pf* always terminates.

B.11.4 Tactic req_mk_hash_prp_pf

Calling *fp_prp_match* with a formula formed by *rq_hash_code_checked* doesn't require any further calls to other tacticals. Hence applying the tactic *req_mk_hash_prp_pf* always terminates.

B.11.5 Tactic req_mk_ide_prp_pf

Calling *fp_prp_match* with a formula formed by *rq_export_id_exists* doesn't require any further calls to other tacticals. Hence applying the tactic *req_mk_ide_prp_pf* always terminates.

By induction, calling the tactical *fp_prp_match* always terminates.

B.12 Tactical fp_cdsc_valid

The tactical *fp_cdsc_valid* returns a derivation of a formula formed by the formula constructor *component_dsc_valid*. The input of *component_dsc_valid* is a formula of type *component_dsc*. There exist two formula constructors for type *component_dsc*:

- mk_cdsc : (set property) → (list (set property)) → component_dsc.
- cdsc_combine : component_dsc → component_dsc → component_dsc.

The constructor *mk_cdsc* builds a formula of type *component_dsc* out of a set of exported properties and a list of sets of imported properties. The constructor *cdsc_combine* returns a formula of type *component_dsc* out of two formulas of type *component_dsc*.

The termination of the tactical *fp_cdsc_valid* is proved by induction on the structure of the input formula of the formula constructor *component_dsc_valid*.

The base case occurs when an input formula of *cdsc_valid* is built by using *mk_cdsc*. Only the tactic *mk_cdsc_valid_pf* is applicable in this case. Applying *mk_cdsc_valid_pf* is always reduced to calling the tactical *fp_list_valid* with a formula formed by the formula constructor *list_valid*. As we already showed in the earlier part, calling the tactical *fp_list_valid* with a formula of *list_valid* always terminates. Hence, calling *fp_cdsc_valid* with a formula of *mk_cdsc* always terminates, and then calling *fp_csc_valid* terminates in the base case.

For the inductive step, suppose that calling *fp_cdsc_valid* with a formula cdsc_valid(d1) and calling *fp_cdsc_valid* with a formula cdsc_valid(d2) terminate. Then calling *fp_cdsc_valid* with a formula cdsc_valid(cdsc_combine(d1, d2)) is always reduced to calling *fp_cdsc_valid* with a formula cdsc_valid(d1) and calling *fp_cdsc_valid* with a formula cdsc_valid(d2). These two calls terminate by the induction hypothesis. Therefore calling *fp_csc_valid* terminates in the inductive step.

Hence, calling the tactical *fp_cdsc_valid* always terminates by induction.

B.13 Tactical fp_signed_keybind

Calling the tactical *fp_signed_keybind* with a list Γ of assumptions and a formula formed by the formula constructors *says* and *keybind* is always reduced to calling the tactical *search_assmp* with Γ and a formula of *keybind* and calling *search_assmp* with Γ and a formula of *signed*. As already shown, calling *search_assmp* with those two formula constructors always terminates. Therefore applying the tactic *keycert_pf* always terminates.

B.14 Tactical fp_signed_auth

Calling the tactical *fp_signed_auth* with a list Γ of assumptions and a formula of the formula constructors *says* and *prp_auth* is always reduced to calling the tactical *search_assmp* with Γ and a formula of *key_authority*, calling the tactical *fp_signed_keybind* with Γ and a formula of *says* and *keybind*, and calling *search_assmp* with Γ and a formula of *signed*. As already shown, calling *fp_signed_keybind* with a formula of *says* and *keybind* terminates, and calling *search_assmp* with a formula of *key_authority* or *signed* always terminates. Therefore applying the tactic *prp_auth_pf* always terminates.

B.15 Tactical fp_signed_dsc

Calling the tactical *fp_signed_assmp* with a list Γ of assumptions and a formula of the formula constructors *says* and *module_dsc* is always reduced to calling the tactical *search_assmp* with Γ and a formula of *key_authority*, calling the tactical *fp_signed_keybind* with Γ and a formula of *says* and *keybind*, and calling *search_assmp* with Γ and a formula of *signed*. As already shown, calling *fp_signed_keybind* with a formula of *says* and *keybind* always terminates, and calling *search_assmp* with a formula of *key_authority* or *signed* always terminates. Therefore applying the tactic *component_dsc_pf* always terminates.

B.16 Tactical fp_valid_sig_auth

Calling the tactical *fp_valid_sig_auth* with a list Γ of assumptions and a formula of the formula constructor *valid_sig_prp_auth* is reduced to five calls to other tacticals:

- calling the tactical *search_assmp* with Γ and a formula of *key_authority*,
- calling the tactical *search_assmp* with Γ and a formula of *keybind*,
- calling the tactical *search_assmp* with Γ and a formula of *prp_server*,
- calling the tactical *fp_signed_keybind* with Γ and a formula of *says* and *keybind*, and
- calling the tactical *fp_signed_auth* with Γ and a formula of *says* and *prp_auth*.

As proved earlier, calling the tactical *search_assmp* with a formula of *key_authority* or *prp_server*, calling the tactical *fp_signed_keybind* with a formula of *says* and *keybind*, and calling the tactical *fp_signed_auth* with a formula of *says* and *prp_auth* always terminates. Hence applying the tactic *valid_sig_auth_pf* always terminates.

B.17 Tactical fp_valid_sig_cdsc

Calling the tactical *fp_valid_sig_cdsc* with a list Γ of assumptions and a formula of the formula constructor *valid_sig_component_dsc* is reduced to following 5 calls to other tacticals:

- calling the tactical *search_assmp* with Γ and a formula of *key_authority*,
- calling the tactical *search_assmp* with Γ and a formula of *keybind*,

- calling the tactical *search_assmp* with Γ and a formula of *module_authority*,
- calling the tactical *fp_signed_keybind* with Γ and a formula of *says* and *keybind*, and
- calling the tactical *fp_signed_dsc* with Γ and a formula of *says* and *module_dsc*.

As proved earlier, calling the tactical *search_assmp* with a formula of *key_authority*, *keybind*, or *module_authority*, calling the tactical *fp_signed_keybind* with a formula of *says* and *keybind*, and calling the tactical *fp_signed_dsc* with a formula of *says* and *module_dsc* always terminates. Therefore applying the tactic *valid_sig_cdsc_pf* always terminates.

B.18 Tactical fp_signed_by_ma

Calling the tactical *fp_signed_by_ma* with a list Γ of assumptions and a formula of the formula constructor *signed_by_ma* is reduced into following calls to other tacticals:

- calling the tactical *fp_valid_sig_auth* with the list of the assumptions and a formula of *valid_sig_prp_auth*,
- calling the tactical *fp_valid_sig_cdsc* with the list of the assumptions and a formula of *valid_sig_component_dsc*, and
- calling the tactical *fp_set* with a formula of *singleton* and/or *set_union*.

As already shown, calling the tactical *fp_valid_sig_auth* with a list Γ of the assumptions and a formula of the formula constructor *valid_sig_prp_auth*, calling the tactical *fp_valid_sig_cdsc* with a list Γ of the assumptions and a formula of the formula constructor *valid_sig_component_dsc* and calling the tactical *fp_set* with a formula of the formula constructors *singleton* and *set_union* always terminates. Therefore applying the tactical *signed_by_ma_pf* always terminates.

B.19 Tactical fp_as_cdsc_combine

Given a list Γ of assumptions and a formula of the formula constructor *signed_rq*, the tactical *fp_as_cdsc_combine* returns the derivation of the formula.

The input of the formula constructor *signed_rq* consists of a formula standing for a set of type *module*, a formula of type *component_dsc* and a formula of type *prp_req*. The tactical *fp_as_cdsc_combine* reasons about different cases of the second input formula of type *component_dsc*. The termination of the tactical *fp_as_cdsc_combine* can be proved by induction on the structure of the second input formula.

As already shown in the proof of the tactical *fp_cdsc_valid*, there exist two formula constructors for type *component_dsc*: *mk_cdsc* and *cdsc_combine*.

The base case occurs when the second input formula of the formula constructor *signed_rq* is built using the constructor *mk_cdsc*. In this case, only the tactic *sig_rq_sng_pf* is applicable. Applying the tactic *sig_rq_sng_pf* is always reduced to calling the tactical *fp_signed_by_ma* with the list Γ of assumptions and a formula of the formula constructor *signed_by_ma*. As already shown, calling the tactical *fp_signed_by_ma* with a list of assumptions and a formula of the formula constructor *signed_by_ma* always terminates. Therefore calling the tactical *fp_as_cdsc_combine* with a formula of *signed_rq* terminates in the base case.

For the inductive step, suppose that calling the tactical *fp_as_cdsc_combine* with a list Γ of assumptions and a formula signed_rq(m, d1, rq) terminates, and calling the tactical *fp_as_cdsc_combine* with a list Γ of assumptions and a formula signed_rq(m, d2, rq) terminates.

There are two applicable tactics for a formula signed_rq(m, cdsc_combine(d1, d2), rq): *sig_rq_cdsc_combine_i1_pf* and *sig_rq_cdsc_combine_i2_pf*.

B.19.1 Tactic sig_rq_cdsc_combine_i1_pf

Applying the tactic *sig_rq_cdsc_combine_i1_pf* is always reduced to calling the tactical *fp_cdsc_valid* with formulas component_dsc_valid(d1) and component_dsc_valid(d2), and calling the tactical *fp_as_cdsc_combine* with Γ and a formula signed_rq(m, d1, rq). Calling the tactical *fp_cdsc_valid* with a formula of *component_dsc_valid* always terminates as shown before, and calling *fp_as_cdsc_combine* with Γ and a formula signed_rq(m, d1, rq) terminates by the induction hypothesis; thus applying the tactic *sig_rq_cdsc_combine_i1* terminates.

B.19.2 Tactic sig_rq_cdsc_combine_i2_pf

Applying the tactic *sig_rq_cdsc_combine_i2_pf* is always reduced to calling the tactical *fp_cdsc_valid* with formulas component_dsc_valid(d1) and component_dsc_valid(d2), and calling the tactical *fp_as_cdsc_combine* with Γ and a formula signed_rq(m, d2, rq). Calling the tactical *fp_cdsc_valid* with a formula of *component_dsc_valid* always terminates as shown before, and calling *fp_as_cdsc_combine* with Γ and a formula signed_rq(m, d2, rq) terminates by the induction hypothesis; thus applying the tactic *sig_rq_cdsc_combine_i2* terminates.

Therefore calling the tactical *fp_as_cdsc_combine* with a list of assumptions and a formula of *signed_rq* terminates in the inductive step.

Hence by induction calling the tactical *fp_as_cdsc_combine* with a list of assumptions and a formula of the formula constructor *signed_rq* always terminates.

B.20 Tactical fp_as_set_union

Given a list of assumptions Γ and a formula of the formula constructor *all_signed*, the tactical *fp_as_set_union* returns the derivation of the formula.

As mentioned, the input of the formula constructor *all_signed* consists of a formula standing for a set of type *module*, a formula of type *component_dsc* and a formula standing for a set of type *property*. The tactical *fp_as_set_union* reasons about different cases of the third input formula.

The termination of calling the tactical *fp_as_set_union* with a formula all_signed(m, dsc, rqset) can be proved by induction on the structure of rqset of the formula constructor *all_signed*:

B.20.1 Case of empty set

The tactic *all_sig_emptyset_pf* is applied when rqset is empty. Applying the tactic *all_sig_emptyset_pf* is reduced to calling the tactical *fp_emptyset* with a formula isempty(rqset). Since calling the tactical *fp_emptyset* with a formula of *isempty* terminates, applying the tactic *all_sig_emptyset_pf* terminates.

B.20.2 Case of non-empty set

The base case occurs when rqset is built using the formula constructor *singleton*. The tactic *all_sig_pf* is the only tactic applicable in this case. Applying the tactic *all_sig_pf* with a list of assumption Γ and a formula of all_signed(m, dsc, singleton(rq)) is reduced to calling the tactical *fp_as_cdsc_combine* with a formula signed_rq(m, dsc, rq). Since calling the tactical *fp_as_cdsc_combine* with a formula of the formula constructor *signed_rq* always terminates, applying the tactic *all_sig_pf* terminates in the base case.

For the inductive step, suppose that calling the tactical *fp_as_set_union* with a list of assumptions Γ and a formula all_signed(m,dsc, s1) terminates, and and calling the tactical *fp_as_set_union* with Γ and a formula all_signed(m,dsc, s2) terminates.

Then calling the tactical *fp_as_set_union* with Γ and a formula all_signed(m,dsc, set_union(s1,s2)) is reduced to calling the tactical *fp_as_set_union* with Γ and a formula all_signed(m,dsc, s1), and calling the tactical *fp_as_set_union* with Γ and a formula all_signed(m,dsc, s2) by the tactic *all_sig_set_union_pf*.

Since calling the tactical *fp_as_set_union* with Γ and all_signed(m,dsc, s1) and calling the tactical *fp_as_set_union* with Γ and all_signed(m,dsc, s2) terminate by the induction hypothesis, applying the tactic *all_sig_set_union_pf* terminates.

By induction, the tactical *fp_as_set_union* terminates when the third input formula is not empty.

Hence by induction on cases, calling the tactical *fp_as_set_union* with a list of assumptions and a formula of the formula constructor *all_signed* terminates.

B.21 Tactical fp_valid_cdsc

Given a list of assumptions and a formula of the formula constructor *signed_component_dsc*, the tactical *fp_valid_cdsc* returns the derivation of the formula.

Calling the tactical *fp_valid_cdsc* with a formula signed_component_dsc(m,dsc,rqset) is always reduced to calling the tactical *fp_cdsc_valid* with component_dsc_valid(dsc), and calling the tactical *fp_as_set_union* with all_signed(m,dsc,rqset). As proved already, those two calls always terminates. Hence calling the tactical *fp_valid_cdsc* always terminates.

B.22 Tactical fp_valid_lib

Given a list of assumptions and a formula of the formula constructor *valid_library*, the tactical *fp_valid_lib* returns the derivation of the formula.

The input of the formula constructor *valid_library* consists of a list lib whose elements are set of type *module* and a list libdsc whose elements are of type *component_dsc*.

If the lengths of the two input lists are different, the numbers of *list_cons* and *list_nil* of two lists is also different. That means, during proving, *fp_valid_lib* can be called with two input lists when the first input list of *valid_library* is built by *list_cons* although the second input list is built by *list_nil*, vice versa. In either case, there exist no tactics of *fp_valid_lib*. Thus the prover terminates with reporting failure.

If the lengths of the two input lists are the same, the termination of the tactical *fp_valid_lib* with a list of assumptions and a formula of *valid_library* is be proved by induction on the structure of the input lists.

The base case occurs when two input lists are nil. The tactic *valid_lib_nil_pf* is applied in this case, and applying the tactic *valid_lib_nil_pf* is reduced to calling the tactical *fp_list_nil* with a formula list_is_nil(lib) and calling the tactical *fp_list_nil* with a formula list_is_nil(libdsc).

Since the tactical *fp_list_nil* always terminates with a formula of *list_is_nil*, applying the tactic *valid_lib_nil_pf* terminates. Therefore calling the tactical *fp_valid_lib* with a list of assumptions and a formula of *valid_library* terminates in the base case.

For the inductive step, suppose that calling the tactical *fp_valid_lib* with a list of assumptions Γ and a formula valid_library(libtl, libdsctl) terminates where lib = list_cons(m, libtl) and libdsc = list_cons(dsc, libdsctl).

Then calling the tactical *fp_valid_lib* with Γ and a formula valid_library(lib, libdsc) is reduced to calling *fp_valid_lib* with Γ and a formula valid_library(libtl, libdsctl) and calling the tactical *search_assmp* with Γ and a formula library_dsc(m,dsc) by the tactic *valid_lib_cons_pf*.

We already prove that calling the tactical *search_assmp* with a list of assumptions and a formula of *library_dsc* always terminates. Calling the tactical *fp_valid_lib* with Γ and a formula valid_library(libtl, libdsctl) terminates by the induction hypothesis.

Therefore applying the tactic *valid_lib_cons_pf* terminates; in other words, calling the tactical *fp_valid_lib* with a list of assumptions and a formula of *valid_library* terminates in the inductive step.

Hence, calling the tactical *fp_valid_lib* with a list of assumptions and a formula of *valid_library* always terminates by induction.

B.23 Tactical fp_has_prps_set_union′

Given a formula of the formula constructor *has_prp*, the tactical *fp_has_prps_set_union′* returns the derivation of the formula.

The input of the formula constructor *has_prp* consists of a formula of type *prp_req* and a formula standing for a set of type *property*. The tactical *fp_has_prps_set_union′* reasons about different cases of the second input formula.

B.23.1 Case of empty set

If the second input formula of the formula constructor *has_prp* is a empty set, then there exists no tactic applicable. Hence the prover terminates with reporting failure.

B.23.2 Case of non-empty set

As shown in Section B.5, there exist two formula constructors for non-empty sets : *singleton* and *set_union*. The termination of the tactical *fp_has_prps_set_union′* can be proved by induction on the structure of the second input formula of *has_prp*.

The base case occurs when the second input formula of *has_prp* is built using the formula constructor *singleton*. Only the tactic *has_prps_singleton_pf* is applicable in this case. Applying the tactic *has_prps_singleton_pf* is always reduced to calling the tactical *fp_prp_match* with a formula of *prp_match*. As already shown, calling the tactical *fp_prp_match* with a formula of *prp_match* always terminates. Therefore calling the tactical *fp_has_prps_set_union′* with a formula of *has_prp* terminates in the base case.

For the inductive step, suppose that calling the tactical *fp_has_prps_set_union′* with a formula has_prp(rq, p1) terminates, and calling the tactical *fp_as_cdsc_combine* with a formula has_prp(rq, p2) terminates. There are two applicable tactics:

- **Tactic has_prps_union1_pf**
  Applying the tactic *has_prps_union1_pf* is always reduced to calling the tactical *fp_has_prps_set_union′* with a formula has_prp(rq, p1). This call terminates by the induction hypothesis; thus applying the tactic *sig_rq_cdsc_combine_i1* terminates.
- **Tactic has_prps_union2_pf**
  Applying the tactic has_prps_union2_pf is always reduced to calling the tactical *fp_has_prps_set_union′* with a formula has_prp(rq, p2). This call terminates by the induction hypothesis; thus applying the tactic *sig_rq_cdsc_combine_i2* terminates.

Therefore calling the tactical *fp_has_prps_set_union′* with a formula of *has_prp* terminates in the inductive step.

Hence, calling the tactical *fp_has_prps_set_union′* with a formula of the formula constructor *has_prp* always terminates by induction.

B.24 Tactical fp_has_prps_set_union

Given a formula of the formula constructor *has_property*, the tactical *fp_has_prps_set_union* returns the derivation of the formula.

The input of the formula constructor *has_property* consists of a formula standing for a set of type *prp_req* and a formula standing for a set of type *property*. The tactical *fp_has_prps_set_union* reasons about different cases of the first input formula.

The termination of calling the tactical *fp_has_prps_set_union* with a formula has_property(rqset, prpset) can be proved by induction on the structure of the first input formula rqset of *has_property*.

B.24.1 Case of empty set

The tactic *has_prps_emptyset_pf* is applied when rqset is empty. Applying *has_prps_emptyset_pf* is reduced to calling the tactical *fp_emptyset* with a formula isempty(rqset). Since calling *fp_emptyset* with a formula of *isempty* terminates, applying *has_prps_emptyset_pf* terminates.

B.24.2 Case of non-empty set

The base case occurs when rqset is built using the formula constructor *singleton*. The tactic *has_prps_pf* is the only tactic applicable in this case. Applying *has_prps_pf* with a formula has_property(singleton(rq), prpset) is reduced to calling the tactical *fp_has_prps_set_union′* with a formula has_prp(rq, prpset). Since calling *fp_has_prps_set_union′* with a formula of *has_prp* always terminates, applying *has_prps_pf* terminates.

For the inductive step, suppose that calling *fp_has_prps_set_union* with a formula has_property(r1, prpset) terminates, and calling *fp_has_prps_set_union* with a formula has_property(r2, prpset) terminates.

Then calling the tactical *fp_has_prps_set_union* with a formula has_property(set_union(r1,r2), prpset) is reduced to calling the tactical *fp_has_prps_set_union* with a formula has_property(r1, prpset), and calling the tactical *fp_has_prps_set_union* with a formula has_property(r2, prpset) by the tactic *has_prps_set_union_pf*.

Since calling the tactical *fp_has_prps_set_union* with the formulas has_property(r1, prpset) and has_property(r2, prpset) terminate by the induction hypothesis, applying the tactic *has_prps_set_union_pf* terminates.

By induction, the tactical *fp_has_prps_set_union* terminates when the first input formula of *has_property* is not empty.

Hence by induction on cases, calling the tactical *fp_has_prps_set_union* with a formula of the formula constructor *has_property* terminates.

B.25 Tactical fp_exp_prps′

Given a formula of the formula constructor *exp_rprp*, the tactical *fp_exp_prps′* returns the derivation of the formula.

The input of the formula constructor *exp_rprp* consists of a formula of type *prp_req* and a formula of type *component_dsc*. The tactical *fp_exp_prps′* reasons about different cases of the second input formula of type *component_dsc*. The termination of the tactical *fp_exp_prps′* can be proved by induction on the structure of the second input formula.

As shown in Section B.12, there exist two formula constructors for type *component_dsc*: *mk_cdsc* and *cdsc_combine*.

The base case occurs when the second input formula of the formula constructor *exp_rprp* is built using the constructor *mk_cdsc*. Only the tactic *exp_rprp_cdsc_pf* is applicable in this case. Applying the tactic *exp_rprp_cdsc_pf* with a formula exp_rprp(rq, mk_cdsc(exp, imp)) is always reduced to calling the tactical *fp_has_prps_set_union′* with a formula has_prp(rq, exp). As shown already, calling the tactical *fp_has_prps_set_union′* with a formula of *has_prp* always terminates. Therefore calling the tactical *fp_exp_prps′* with a formula of *exp_rprp* terminates in the base case.

For the inductive step, suppose that calling the tactical *fp_exp_prps′* with a formula exp_rprp(rq, c1) terminates and calling the tactical *fp_exp_prps′* with a formula exp_rprp(rq, c2) terminates. There are two applicable tactics for a formula exp_rprp(m, cdsc_combine(c1, c2), rq):

B.25.1 Tactic exp_rprp_cdsc_i1_pf

Applying the tactic *exp_rprp_cdsc_i1_pf* with a formula exp_rprp(m, cdsc_combine(c1, c2), rq) is always reduced to calling the tactical *fp_exp_prps′* with exp_rprp(rq, c1). This call terminates by the induction hypothesis; thus applying the tactic *exp_rprp_cdsc_i1_pf* terminates.

B.25.2 Tactic exp_rprp_cdsc_i2_pf

Applying the tactic *exp_rprp_cdsc_i2_pf* with a formula exp_rprp(m, cdsc_combine(c1, c2), rq) is always reduced to calling the tactical *fp_exp_prps′* with a formula exp_rprp(rq, c2). This call terminates by the induction hypothesis; thus applying the tactic *exp_rprp_cdsc_i2_pf* terminates.

Therefore calling the tactical *fp_exp_prps′* with a formula of *exp_rprp* terminates in the inductive step.

Hence calling the tactical *fp_exp_prps′* with a formula of the formula constructor *exp_rprp* always terminates by induction.

B.26 Tactical fp_exp_prps

Given a formula of the formula constructor *export_required_prps*, the tactical *fp_exp_prps* returns the derivation of the formula.

The input of the formula constructor *export_required_prps* consists of a formula standing for a set of type *prp_req* and a formula of type *component_dsc*. The tactical *fp_exp_prps* reasons about different cases of the first input formula.

The termination of calling the tactical *fp_exp_prps* with a formula export_required_prps(rqset, dsc) can be proved by induction on the structure of the first input formula rqset of the formula constructor *export_required_prps*.

B.26.1 Case of empty set

The tactic *export_rprps_empty_pf* is applied when rqset is empty. Applying *export_rprps_empty_pf* is reduced to calling the tactical *fp_emptyset* with a formula isempty(rqset). Since calling *fp_emptyset* with a formula of *isempty* terminates, applying *export_rprps_empty_pf* terminates.

B.26.2 Case of non-empty set

The base case occurs when rqset is built using the formula constructor *singleton*. The tactic *export_rprps_sng_pf* is the only tactic applicable in this case. Calling *export_rprps_sng_pf* with a formula export_required_prps(singleton(rq), dsc) is reduced to calling the tactical *fp_exp_prps′* with a formula with a formula exp_rprp(rq, dsc). Since calling *fp_exp_prps′* with a formula of *exp_rprp* always terminates, applying *export_rprps_sng_pf* terminates.

For the inductive step, suppose that calling *fp_exp_prps* with a formula export_required_prps(r1, dsc) terminates, and calling *fp_exp_prps* with a formula export_required_prps(r2, dsc) terminates.

Then calling the tactical *fp_exp_prps* with a formula export_required_prps(set_union(r1,r2), dsc) is reduced to calling the tactical *fp_exp_prps* with a formula export_required_prps(r1, dsc), and calling the tactical *fp_exp_prps* with a formula export_required_prps(r2, dsc) by the tactic *export_rprps_set_union_pf*.

Since calling *fp_exp_prps* with export_required_prps(r1, dsc) and calling *fp_exp_prps* with export_required_prps(r2, dsc) terminate by the induction hypothesis, applying the tactic *export_rprps_set_union_pf* terminates.

By induction, the tactical *fp_exp_prps* terminates when the first input formula is not empty.

Hence, calling the tactical *fp_exp_prps* with a formula of the formula constructor *export_required_prps* terminates by induction on cases.

B.27 Tactical fp_st_imprt_cdsc

The tactical *fp_st_imprt_cdsc* returns a derivation of a formula of the formula constructor *satisfy_imprt_cdsc*.

The input of the formula constructor *satisfy_imprt_cdsc* consists of a formula standing for a set of *prp_req*, and a formula standing for a list of sets of *property*. The tactical *fp_st_imprt_cdsc* reasons about different cases of the second input formula.

B.27.1 Tactic satisfy_imprt_hd_pf

Calling *satisfy_imprt_hd_pf* with a formula satisfy_imprt_cdsc(imp, list_cons(set_equiv, exp, explist)) is reduced to calling the tactical *fp_has_prps_set_union* with a formula has_property(imp, exp).

Since *fp_has_prps_set_union* always terminates with a formula formed by *has_property*, the tactic *satisfy_imprt_hd_pf* always terminate.

B.27.2 Tactic satisfy_imprt_tl_pf

The termination of calling the tactical *fp_st_imprt_cdsc* with a formula satisfy_imprt_cdsc(imp, l) can be proved by induction on the length of list l. As shown in Section B.9, there exist two formula constructors for lists : *list_nil* and *list_cons*.

The base case occurs when the length of a list is zero. Since there is no matching tactic, the prover simply terminates after reporting failure. Therefore calling *fp_st_imprt_cdsc* with a formula of *satisfy_imprt_cdsc* terminates in the base case.

For the inductive step, suppose that calling *fp_st_imprt_cdsc* with a formula satisfy_imprt_cdsc(imp, explist) terminates where the length of explist is k. Then calling *fp_st_imprt_cdsc* with satisfy_imprt_cdsc(imp, list_cons(set_equiv, exp, explist) is reduced to calling the tactical *fp_list_valid* with list_valid(set_equiv, explist), and calling *fp_st_imprt_cdsc* with satisfy_imprt_cdsc(imp, explist). Calling *fp_list_valid* with a formula of *list_valid* always terminates as shown before, and calling *fp_st_imprt_cdsc* with satisfy_imprt_cdsc(imp, explist) terminates by the inductive hypothesis; therefore calling *fp_st_imprt_cdsc* with a formula of *satisfy_imprt_cdsc* terminates when the length of the second input list is $k + 1$.

By induction, the tactical *fp_st_imprt_cdsc* with a formula of *satisfy_imprt_cdsc* always terminates.

B.28 Tactical fp_st_imprt

Given a list of assumptions and a formula of the formula constructor *satisfy_import_req*, the tactical *fp_st_imprt* returns the derivation of the formula.

The input of the formula constructor *satisfy_import_req* consists of a formula standing for a list of sets of *prp_req*, and a formula standing for a list of sets of *property*.

The termination of the tactical *fp_st_imprt* with a formula satisfy_import_req(implist, libdsc) can be proved by induction on the structure of the first input list implist.

The base case occurs when the first input list is nil. The tactic *satisfy_imprt_nil_pf* is applied in this case, and applying *satisfy_imprt_nil_pf* to satisfy_import_req(list_nil, libdsc) is reduced to calling the tactical *fp_list_nil* with a formula list_is_nil(list_nil).

Since the tactical *fp_list_nil* always terminates with a formula of *list_is_nil*, applying *satisfy_imprt_nil_pf* terminates.

For the inductive step, suppose that calling the tactical *fp_st_imprt* with a formula satisfy_import_req(imptl, libdsc) terminates where implist = list_cons(set_equiv, imp, imptl). Then calling the tactical *fp_st_imprt* with a formula satisfy_import_req(implist, libdsc) is reduced to calling the tactical *fp_st_imprt_cdsc* with a formula satisfy_imprt_cdsc(imp, libdsc), and calling the tactical *fp_st_imprt* with a formula satisfy_import_req(imptl, libdsc) by the tactic *satisfy_imprt_cons_pf*.

We already prove that calling the tactical *fp_st_imprt_cdsc* with a formula of *satisfy_imprt_cdsc* always terminates. Calling the tactical *fp_st_imprt* with satisfy_import_req(imptl, libdsc) terminates by the induction hypothesis.

Therefore applying *fp_st_imprt* to a formula of *satisfy_import_req* terminates in the inductive step.

Hence, calling the tactical *fp_st_imprt* with a formula of *satisfy_import_req* always terminates by induction.

B.29 Tactical fp_prv_prps

Given a list of assumptions and a formula of the formula constructor *provide_enough_prps*, the tactical *fp_prv_prps* returns the derivation of the formula.

The input of the formula constructor *provide_enough_prps* consists of a formula of type *component_dsc*, a formula standing for a list of sets of *module*, and a formula standing for a list of sets of *property*. The tactical *fp_prv_prps* reasons about different cases of the first input formula. The termination of calling the tactical *fp_prv_prps* with a list of assumptions and a formula provide_enough_prps(dsc, lib, libdsc) can be proved by induction on the structure of the first input formula dsc of the formula constructor *provide_enough_prps*.

As shown in Section B.12, there exist two formula constructors for type *component_dsc*: *mk_cdsc* and *cdsc_combine*.

The base case occurs when dsc is built using the formula constructor *mk_cdsc*. The tactic *prv_prps_pf* is the only tactic applicable in this case. Calling *prv_prps_pf* with a list Γ of assumptions and a formula provide_enough_prps(mk_dsc(exp, imp), lib, libdsc) is reduced to calling the tactical *fp_list_valid* with a formula list_valid(set_equiv, imp), calling the tactical *fp_valid_lib* with Γ and a formula valid_library(lib, libdsc), and calling the tactical *fp_st_imprt* with a formula satisfy_import_req(imp, libdsc). Since calling *fp_list_valid* with a formula of *list_valid*, calling *fp_valid_lib* with a list of assumptions and a formula of *valid_library*, and calling *fp_st_imprt* with a formula of *satisfy_import_req* always terminate as proved before, applying the tactic *prv_prps_pf* terminates.

For the inductive step, suppose that calling *fp_prv_prps* with a list Γ of assumptions and a formula provide_enough_prps(c1, lib, libdsc) terminates, and calling *fp_prv_prps* with Γ and a formula provide_enough_prps(c2, lib, libdsc) terminates.

Then calling the tactical *fp_prv_prps* with Γ and a formula provide_enough_prps(cdsc_combine(c1, c2), lib, libdsc) is reduced to calling *fp_cdsc_valid* with component_dsc_valid(c1), calling *fp_cdsc_valid* with component_dsc_valid(c2), calling the tactical *fp_prv_prps* with Γ and a formula provide_enough_prps(c1, lib, libdsc), and calling *fp_prv_prps* with Γ and a formula provide_enough_prps(c2, lib, libdsc) by the tactic *prv_prps_cdsc_combine_pf*.

Since calling *fp_cdsc_valid* with a formula of *component_dsc_valid* always terminates as proved earlier, and calling *fp_prv_prps* with provide_enough_prps(c1, lib, libdsc) and provide_enough_prps(c2, lib, libdsc) terminates by the induction hypothesis, applying the tactic *prv_prps_cdsc_combine_pf* terminates.

Hence, calling the tactical *fp_prv_prps* with a list of assumptions and a formula of *provide_enough_prps* terminates by induction.

B.30 Tactical findproof

Given a list of assumptions and a formula of the formula constructor *ok_to_link*, the tactical *findproof* returns the derivation of the formula.

Calling the tactical *findproof* with a list Γ of assumptions and a formula ok_to_link(m, dsc, lib, libdsc, rqset) is always reduced to calling the tactical *fp_valid_cdsc* with Γ and a formula signed_component_dsc_valid(m, dsc, rqset), calling the tactical *fp_prv_prps* with Γ and a formula provide_enough_prps(dsc, lib, libdsc), and calling the tactical *fp_exp_prps* with a formula export_required_prps(rqset, dsc).

As proved earlier, calling to calling *fp_valid_cdsc* with a list of assumptions and a formula of *signed_component_dsc_valid*, calling *fp_prv_prps* with a list of assumptions and a formula of *provide_enough_prps*, and calling *fp_exp_prps* with a formula of *export_required_prps* always terminate.

Hence calling the tactical *findproof* with a list of assumptions and a formula of *ok_to_link* always terminates. □

C. COMPLETENESS OF THE PROVER

We will prove Theorem 2 by proving all the tacticals on which the tactical *ok_to_link_pf* depends are complete.

C.1 Tactical search_assmp

The tactical *search_assmp* is related to the formula constructors *singed*, *key_bind*, *key_authority*, *prp_server*, *module_authority* and *library_dsc*.

The tactical *search_assmp* is used for finding a proof from a list of assumptions, which are axioms believed true by the prover. Therefore, for the formula constructors related to the tactical *search_assmp*, whether or not any formula from these constructors is true depends on whether or not a proof of the formula is in the list of assumptions.

We can prove that the tactical *search_assmp* is complete, i.e., it always finds the proof (or a derivation) of a true formula of related formula constructors by showing that the tactical *search_assmp* always find a proof if it is in the assumption list.

There are two tactics calling the tactical *search_assmp*: *initpf* and *init2pf*. The tactic *initpf* is always called before the tactic *init2pf* is called.

The tactic *initpf* determines the first element of the list is the proof which the prover is currently looking for. If so, it returns the proof; otherwise it reports failure. The tactic *init2pf* is applied only after applying the tactic *initpf* fails. That means the first element of the assumption list is not the proof the prover is looking for. So the tactic *init2pf* searches the tail of the list of assumptions for the proof of the formula.

We can prove the completeness of the tactical *search_assmp* by induction on the length of the assumption list.

C.1.1 Case of nil assumption list

Every formula of the related formula constructors is considered true if and only if the proof of it is in the assumption list. Therefore a nil assumption list means that there exists no true formula for the given constructors. This makes the proposition vacuously true.

C.1.2 Case of non-nil assumption list

The base case is when the length of the assumption list is one, in other words, it is the case when calling *search_assmp* with an assumption list, (A by P, nil) and a true formula A. Since formulas of the related constructors are considered true only when their proofs are in the assumption list, there exists one true formula A in this case.

The tactical *search_assmp* finds the proof of the true formula A by the tactic *initpf*; thus *search_assmp* is complete in the base case.

For the inductive step, suppose that the tactical *search_assmp* is complete on any assumption list of length k.

For any assumption list of length $k + 1$, with a true formula A, first, the prover calls the tactic *initpf* to see if the first element of the assumption list is the proof of A. If so, it returns the proof from the list. Otherwise, the prover calls the tactic *init2pf*. The tactic *init2pf* calls the tactical *search_assmp* with the tail of the assumption list of length k. By the induction hypothesis, the tactical *search_assmp* is complete on an assumption list of length k, i. e., it always finds the proof of any true formula from an assumption list of length k. Therefore the tactical *search_assmp* is complete for an assumption list of length $k + 1$.

By induction, the tactical *search_assmp* always finds a proof from a list of assumptions. Hence, the tactical *search_assmp* is complete.

C.2 Tactical fp_string_valid

The tactical *fp_string_valid* is related to the formula constructor *string_valid*. The input of the formula constructor *string_valid* is a formula of type *string* and there exists two formula constructors for type *string*:

- mk_str1 : char $\rightarrow$ string.
- mk_str : char $\rightarrow$ string $\rightarrow$ string.

The constructor *mk_str1* is used to construct a formula of type *string* from a formula of type *char*. The formula built by *mk_str1* can be considered as a string of length 1. The only way to construct a bigger formula of type *string* is using the constructor *mk_str*. The constructor *mk_str* returns a new formula of type *string* after adding the input formula of type *char* to the head of another input formula of type *string*.

The completeness of the tactical *fp_string_valid* can be proved by induction on the structure of the input formula of *string_valid*.

The base case occurs when an input formula is formed by the formula constructor *mk_str1*. If the input formula is true, then the prover always returns the proof *string_valid_mk_str1* using the tactic *string_valid_mk_str1_pf*. Therefore, in the base case, the tactical *fp_string_valid* always find the proof of a true formula of the formula constructor *string_valid*.

For the inductive step, suppose that the tactical *fp_string_valid* returns the proof of a true formula string_valid(str). We can make a new and bigger formula of type *string* by using the formula constructor *mk_str*. Then calling the tactical *fp_string_valid* with a true formula string_valid(mk_str(ch, str)) is always reduced to calling the tactical *fp_string_valid* with string_valid(str) by the tactic *string_valid_mk_str_pf*. By the induction hypothesis, the tactical *fp_string_valid* can find the proof of string_valid(str). Therefore the tactical *fp_string_valid* can find the proof of string_valid(mk_str(ch, str)).

By induction, the tactical *fp_string_valid* always find a proof of a true formula formed by *string_valid*; thus it is complete.

C.3 Tactical fp_ide_valid

The tactical *fp_ide_valid* is related to the formula constructor *ide_valid*.

The formula constructor *ide_valid* takes a formula of type *ide*, and there exists only one formula constructor in the linking logic which can build a formula of type *ide*. The formula constructor *mk_ide* constructs a formula of type *ide* from a formula of *string*: mk_ide : string → ide.

For a true formula ide_valid(mk_ide(str)), the tactic *ide_valid_mk_ide_pf* is the only applicable tactic and applying the tactic *ide_valid_mk_ide_pf* is always reduced to calling the tactical *fp_string_valid* with a formula string_valid(str).

Since the tactical *fp_string_valid* is complete on the formula constructor *string_valid*, calling the tactical *fp_string_valid* with a true formula string_valid(str) always finds the proof.

Hence calling the tactical *fp_ide_valid* with a true formula of *ide_valid* always finds the proof; thus, the tactical *fp_ide_valid* is complete on *ide_valid*.

C.4 Tactical fp_emptyset

The tactical *fp_emptyset* is related to the formula constructor *isempty*. In the linking logic, there is only one true formula isempty(emptyset) for the tactical *fp_emptyset* and the tactical *fp_emptyset* always returns the proof *set_empty_pf* by the tactic *set_empty_pf*.

Therefore the tactical *fp_emptyset* is complete.

C.5 Tactical fp_set

The tactical *fp_set* is related to the formula constructors, *singleton* and *set_union*:

- singleton : T → (set T).
- set_union : (set T) → (set T) → (set T).

For a given type T, the constructor *singleton* returns a set of single element, and the constructor *set_union* returns a union set of two sets. In the linking logic, the constructor *set_union* is the only operator on sets for making a bigger set.

The completeness of the tactical *fp_set* can be proved by induction on the structure of the formulas of the formula constructors *singleton* and *set_union*.

The base case occurs when a set consists of only one element. Since the formula constructor *singleton* is used to make a singleton set, the only applicable tactic is the tactic *singleton_refl_pf*. The tactical *fp_set* always find the proof of a true formula of *singleton* by applying the tactic *singleton_refl_pf*; thus, the tactical *fp_set* is complete in the base case.

For the inductive step, suppose that the tactical *fp_set* finds the proof of a formula $\mathsf{X} \in \mathsf{S1}$ if the formula is true, and the tactical *fp_set* also finds the proof of a formula $\mathsf{X} \in \mathsf{S2}$ if the formula is true. Calling the tactical *fp_set* with a true formula $\mathsf{X} \in \mathsf{set\_union}(\mathsf{S1}, \mathsf{S2})$ falls into one of the following two cases:

C.5.1 Tactic set_union1_pf

If the formula $\mathsf{X} \in \mathsf{S1}$ is true, calling the tactical *fp_set* with $\mathsf{X} \in \mathsf{set\_union}(\mathsf{S1}, \mathsf{S2})$ is reduced to calling the tactical *fp_set* with $\mathsf{X} \in \mathsf{S1}$ by the tactic *set_union1_pf*. By inductive hypothesis, the tactical *fp_set* finds the proof of $\mathsf{X} \in \mathsf{S1}$.

C.5.2 Tactic set_union2_pf

If the formula $\mathsf{X} \in \mathsf{S1}$ is not true, then the formula $\mathsf{X} \in S2$ must be true since the formula $\mathsf{X} \in \mathsf{set\_union}(\mathsf{S1}, \mathsf{S2})$ is true. Therefore calling the tactical *fp_set* with $\mathsf{X} \in \mathsf{set\_union}(\mathsf{S1}, \mathsf{S2})$ is reduced to calling the tactical *fp_set* with $\mathsf{X} \in \mathsf{S2}$ by the tactic *set_union2_pf*. By inductive hypothesis, the tactical *fp_set* finds the proof of $\mathsf{X} \in \mathsf{S2}$.

By induction, the tactical *fp_set* always finds the proof of a formula of *set_union*; thus, the tactical *fp_set* is complete in the inductive step.

Hence the tactical *fp_set* is complete on *singleton* and *set_union* by induction.

C.6 Tactical fp_validper

The tactical *fp_validper* is related to the formula constructor *validper*. There are two true formulas for the tactical *fp_validper* in the linking logic: validper(module_eq) and validper(set_equiv). The tactical *fp_validper* returns the proof *module_eq_validper* by using the tactic *module_eq_validper_pf* if an input formula is validper(module_eq), and it returns the proof *set_equiv_validper* by using the tactic *set_equiv_validper_pf* if an input formula is validper(set_equiv).

Therefore the tactical *fp_validper* is complete.

C.7 Tactical fp_validper_refl

The tactical *fp_validper_refl* is related to the formula constructors *module_eq* and *set_equiv*. For the formula constructor *module_eq*, there is only one true formula module_eq(m,m) in the linking logic. The tactical *fp_validper_refl* returns the proof *module_eq_refl* by using the tactic *module_eq_refl_pf*. For the formula constructor *set_equiv*, there is only one true formula set_equiv(s,s) in the linking logic, and the tactical *fp_validper_refl* returns the proof *set_equiv_refl* by using the tactic *set_equiv_refl_pf*.

Therefore the tactical *fp_validper_refl* is complete.

C.8 Tactical fp_list_nil

The tactical *fp_list_nil* is related to the formula constructor *list_is_nil*. In the linking logic, there is only one true formula list_is_nil(list_nil) for the tactical *fp_list_nil*. The tactical *fp_list_nil* always returns the proof *list_is_nil_nil* by the tactic *list_is_nil_nil_pf*.

Therefore the tactical *fp_list_nil* is complete.

C.9 Tactical fp_list_valid

The tactical *fp_list_valid* is related to the formula constructor *list_valid*. The input of the formula constructor *list_valid* is a formula standing for a list of a given type T. There exist 2 formula constructors for type *list T* :

- list_nil : list T.
- list_cons : T → (list T) → (list T).

The constructor *list_nil* returns a nil list. The constructor *list_cons* returns a new list after adding a given head element to a tail list, and using the constructor *list_cons* is the only way of making a new and longer list out of an existing list in the linking logic.

The completeness of the tactical *fp_list_valid* can be proved by induction on the structure of input formulas of the formula constructor *list_valid*.

The base case occurs when an input list is nil. Calling the tactical *fp_list_valid* with a true formula list_valid(eq, list_nil) is reduced to calling the tactical *fp_validper* with a formula validper(eq), and calling the tactical *fp_list_nil* with a formula list_is_nil(list_nil) by the tactic *list_valid_nil_pf*. Since the tactical *fp_validper* and the tactical *fp_list_nil* are complete as proved earlier, the tactical *fp_list_valid* always finds the proof of list_valid(eq, list_nil); thus, it is complete in the base case.

For the inductive step, suppose that calling the tactical *fp_list_valid* with a true formula list_valid(eq, tl) finds the proof when l = list_cons(eq, hd, tl). Calling *fp_list_valid* with a true formula list_valid(eq, l) is always reduced to calling the tactical *fp_validper_refl* with a formula eq(hd, hd), and calling the tactical *fp_list_valid* with list_valid(eq,tl) by the tactic *list_valid_cons_pf*. Calling the tactical *fp_validper_refl* always finds a proof of formulas built by the formula constructors *module_eq* and *set_equiv* as proved earlier, and calling the tactical *fp_list_valid* finds the proof of list_valid(eq,tl) by the inductive hypothesis. So calling the tactical *fp_list_valid* finds the proof of list_valid(eq, l).

By induction, the tactical *fp_list_valid* always find a proof of a true formula formed by *list_nil* and *list_cons*; thus the tactical *fp_list_valid* is complete.

C.10 Tactical fp_prp_valid

The tactical *fp_prp_valid* is related to the formula constructor *prp_valid*. The formula constructor *prp_valid* takes a formula of type *property*, and it checks the validity of the input which is built by the formula constructor *mk_prp*.

For a true formula prp_valid(mk_prp(pname)), the tactic *mk_prp_valid_pf* is the only applicable tactic, and applying the tactic *mk_prp_valid_pf* is always reduced to calling the tactical *fp_ide_valid* with a formula ide_valid(pname).

Since the tactical *fp_ide_valid* is complete on the formula constructor *ide_valid*, calling the tactical *fp_ide_valid* with a true formula ide_valid(pname) always finds the proof.

Hence calling the tactical *fp_prp_valid* with a true formula of *prp_valid* and *mk_prp* always finds the proof; thus, the tactical *fp_prp_valid* is complete on *prp_valid* and *mk_prp*.

C.11 Tactical fp_prp_match

The tactical *fp_prp_match* is related to five different formula constructors *mk_prp_req*, *rq_component_name_exists*, *mk_rq_component_name*, *rq_hashcode_checked* and *rq_export_id_exists*. There exist five tactics corresponding to each formula constructor in the prover, and the tactical *fp_prp_match* is complete on those five formula constructors by induction on cases.

C.11.1 Tactic mk_req_refl_pf

The formula constructor *mk_prp_req* takes two formulas of type *property* built by the constructor *mk_prp* and checks if two formulas exactly match each other.

Calling the tactical *fp_prp_match* with a true formula mk_prp_req(mk_prp(pname), mk_prp(pname)) is always reduced to calling the tactical *fp_prp_valid* with a formula prp_valid(mk_prp(pname)) by the tactic *mk_req_refl_pf*. Since the tactical *fp_prp_valid* is complete on the formula constructors *prp_valid* and *mk_prp*, calling the tactical *fp_prp_valid* with prp_valid(mk_prp(pname)) always finds the proof.

Hence calling the tactical *fp_prp_match* with a true formula of *mk_prp_req* and *mk_prp* always finds the proof; thus, the tactical *fp_prp_match* is complete on *mk_prp_req* and *mk_prp*.

C.11.2 Tactic req_mk_cname_pf

The formula constructor *rq_component_name_exists* takes a formula of type *property* built by the constructor *mk_cname* and checks if the *prp_kind* of the input formula is *prp_component_name*. In the linking logic, using the constructor *mk_cname* is the only way of making a formula of type *property* whose *prp_kind* is *prp_component_name*. This means a true formula of *rq_component_name_exists* always has an input formula built by the constructor *mk_cname*.

Calling the tactical *fp_prp_match* with a true formula rq_component_name_exists(mk_cname(n)) always finds the proof *req_mk_cname* by the tactic *req_mk_cname_pf*. Therefore the tactical *fp_prp_match* is complete on the formula constructor *rq_component_name_exists*.

C.11.3 Tactic req_mk_cname′_pf

The formula constructor *mk_rq_component_name* takes two formulas of type *property*, each of which is built by the constructor *mk_cname*, and checks if two formulas exactly match each other as well as if their *prp_kind* is *prp_component_name*. In the linking logic, using the constructor *mk_cname* is the only way of making a true formula of type *property* whose *prp_kind* is *prp_component_name*. This means a true formula of the formula constructor *mk_rq_component_name* always has two input formulas built by the constructor *mk_cname*.

Calling the tactical *fp_prp_match* with a true formula mk_rq_component_name(mk_cname(n), mk_cname(n)) is always reduced to calling tactical *fp_ide_valid* with a formula ide_valid(n) by the tactic *req_mk_cname′_pf*. Since the tactical *fp_ide_valid* is complete on the formula constructor *ide_valid*, calling the tactical *fp_ide_valid* with ide_valid(n) always finds the proof.

Hence calling the tactical *fp_prp_match* with a true formula of *mk_rq_component_name* always finds the proof; thus, the tactical *fp_prp_match* is complete on *mk_rq_component_name*.

C.11.4 Tactic req_mk_hash_prp_pf

The formula constructor *rq_hashcode_checked* takes a formula of type *property* built by the constructor *mk_hash_prp*, and checks if the *prp_kind* of the input formula is *prp_hash_code*. In the linking logic, using the constructor *mk_hash_prp* is the only way of making a true formula of type *property* whose *prp_kind* is *prp_hash_code*. That means a true formula of the constructor *prp_req_hashcode_checked* always has an input formula built by the constructor *mk_hash_prp*.

Calling the tactical *fp_prp_match* with a true formula rq_hashcode_checked(mk_hash_prp(h)) always finds the proof *req_mk_hash_prp* by the tactic *req_mk_hash_prp_pf*. Therefore the tactical *fp_prp_match* is complete on the formula constructor *rq_hashcode_checked*.

C.11.5 Tactic req_mk_ide_prp_pf

The formula constructor *rq_export_id_exists* takes a formula of type *property* built by the constructor *mk_ide_prp*, and checks if the *prp_kind* of the input formula is *prp_export_ids*. In the linking logic, using the constructor *mk_ide_prp* is the only way of making a true formula of type *property* whose *prp_kind* is *prp_export_ids*. That means, a true formula of the constructor *rq_export_id_exists* always has an input formula built by the constructor *mk_ide_prp*.

Calling the tactical *fp_prp_match* with a true formula rq_export_id_exists(mk_ide_prp(i)) always finds the proof *req_mk_ide_prp* by the tactic *req_mk_ide_prp_pf*; therefore the tactical *fp_prp_match* is complete on the formula constructor *rq_export_id_exists*.

C.12 Tactical fp_cdsc_valid

The tactical *fp_cdsc_valid* is related to the formula constructor *component_dsc_valid*. The input of *component_dsc_valid* is a formula of type *component_dsc*. There exist two formula constructors for type *component_dsc*:

- mk_cdsc : (set property) $\rightarrow$ (list (set property)) $\rightarrow$ component_dsc.
- cdsc_combine : component_dsc $\rightarrow$ component_dsc $\rightarrow$ component_dsc.

The constructor *mk_cdsc* builds a formula of type *component_dsc* out of a set of exported properties and a list of sets of imported properties. The constructor *cdsc_combine* returns a formula of type *component_dsc* out of two formulas of type *component_dsc*.

The completeness of the tactical *fp_cdsc_valid* can be proved by induction on the structure of the input formula of *component_dsc_valid*.

The base case occurs when the input formula of the formula constructor *component_dsc_valid* is built by the constructor *mk_cdsc*. Calling the tactical *fp_cdsc_valid* with a true formula component_dsc_valid(mk_cdsc(exp, imp)) is always reduced to calling the tactical *fp_list_valid* with a formula list_valid(set_equiv, imp) by the tactic *mk_cdsc_valid_pf*. Since the tactical
fp_list_valid is complete on *list_valid*, the tactical *fp_list_valid* always finds the proof of list_valid(set_equiv, imp).

Hence the tactical *fp_cdsc_valid* with the true formula component_dsc_valid(mk_cdsc(exp, imp)) always finds the proof; thus the tactical *fp_cdsc_valid* is complete on the formula constructor *component_dsc_valid* in the base case.

For the inductive step, suppose that calling the tactical *fp_cdsc_valid* with a true formula component_dsc_valid(d1) always finds the proof, and calling *fp_cdsc_valid* with a true formula component_dsc_valid(d2) always finds the proof. Calling the tactical *fp_cdsc_valid* with a true formula component_dsc_valid(cdsc_combine(d1, d2)) is always reduced to by the tactic *cdsc_combine_valid_pf* calling the tactical *fp_cdsc_valid* with a formula component_dsc_valid(d1), and calling the tactical *fp_cdsc_valid* with a formula component_dsc_valid(d2).

Then the tactical *fp_cdsc_valid* always finds the proof of the true formula component_dsc_valid(cdsc_combine(d1, d2)) since the tactical *fp_cdsc_valid* finds the proofs of component_dsc_valid(d1) and component_dsc_valid(d2) by the induction hypothesis; thus the tactical *fp_cdsc_valid* is complete on *component_dsc_valid* in the inductive step.

By induction, the tactical *fp_cdsc_valid* is complete on the formula constructor *component_dsc_valid*.

C.13 Tactical fp_signed_keybind

The tactical *fp_signed_keybind* is related to the formula constructors *says* and *keybind*. In the linking logic, there are three cases in which the formula built by *says* is true, and each case depends on input formulas. Any formula built by *says* is considered true if and only if the second argument of the formula comes from the formula constructors *keybind*, *prp_auth* and *module_dsc*.

But the tactical *fp_signed_keybind* is concerned with a formula whose second argument is formed by *keybind*. This means semantically that *fp_signed_keybind* finds a proof of legitimate bindings between a principal and its public key.

Calling the tactical *fp_signed_keybind* with a list Γ of assumptions and a true formula says(ca, keybind(prn, pkey)) is always reduced to calling the tactical *search_assmp* with Γ and a formula keybind(ca, caKey), and calling the tactical *search_assmp* with Γ and a formula signed (caKey, keycert(prn, pkey, pinfo)) by the tactic *keycert_pf*. Since the tactical *search_assmp* is complete on the formula constructors *signed* and *keybind*, calling the tactical *search_assmp* with Γ and a formula keybind(ca, caKey) and calling the tactical *search_assmp* with Γ and a formula signed (caKey, keycert(prn, pkey, pinfo)) always find the proof.

Hence calling the tactical *fp_signed_keybind* with a list of assumptions and a true formula of the formula constructors *says* and *keybind* always finds the proof; thus, the tactical *fp_signed_assmp* is complete on *says* and *keybind*.

C.14 Tactical fp_signed_auth

The tactical *fp_signed_auth* is related to the formula constructors *says* and *prp_auth*.

Calling the tactical *fp_signed_auth* with a list Γ of assumptions and a true formula says(auth, prp_auth(prn, rset)) is always reduced to calling the tactical *fp_signed_keybind* with Γ and a formula says(ca, keybind(auth, authKey)), calling the tactical *search_assmp* with Γ and a formula key_authority(ca), and calling the tactical *search_assmp* with Γ and a formula signed (authKey, prp_auth(prn, rset)) by the tactic *prp_auth_pf*.

Since the tactical *fp_signed_keybind* is complete on *says* and *keybind*, and the tactical *search_assmp* is complete on the formula constructors *key_authority* and *signed*, calling the tactical *fp_signed_keybind* with Γ and says(ca, keybind(auth, authKey)), and calling the tactical *search_assmp* with Γ and formulas key_authority(ca) and signed (authKey, prp_auth(prn, rset)) always find the proofs.

Hence calling the tactical *fp_signed_auth* with a list of assumptions and a true formula of the formula constructors *says* and *prp_auth* always finds the proof; thus, the tactical *fp_signed_auth* is complete on *says* and *prp_auth*.

C.15 Tactical fp_signed_dsc

The tactical *fp_signed_dsc* is related to the formula constructors *says* and *module_dsc*.

Calling the tactical *fp_signed_dsc* with a list Γ of assumptions and a true formula says(auth, module_dsc(m, dsc)) is always reduced to calling the tactical *fp_signed_keybind* with Γ and a formula says(ca, keybind(auth, authKey)), calling the tactical *search_assmp* with Γ and a formula key_authority(ca), and calling the tactical *search_assmp* with Γ and a formula signed (caKey, module_dsc(m, dsc)) by the tactic *component_dsc_pf*.

Since the tactical *fp_signed_keybind* is complete on *says* and *keybind*, and the tactical *search_assmp* is complete on *key_authority* and *signed*, calling *fp_signed_keybind* with Γ and says(ca, keybind(auth, authKey)), and calling *search_assmp* with Γ and formulas key_authority(ca) and signed (auth, module_dsc(m, dsc)) always find the proofs.

Hence calling the tactical *fp_signed_dsc* with a list of assumptions and a true formula of the formula constructors *says* and *module_dsc* always finds the proof; thus, the tactical *fp_signed_dsc* is complete on *says* and *module_dsc*.

C.16 Tactical fp_valid_sig_auth

Calling the tactical *fp_valid_sig_auth* with a list Γ of assumptions and a true formula valid_sig_prp_auth(ma, prp) is reduced to five calls to other tacticals:

- calling the tactical *search_assmp* with Γ and a formula key_authority(ca),
- calling the tactical *search_assmp* with Γ and a formula keybind(ca, caKey),
- calling the tactical *search_assmp* with Γ and a formula prp_server(pa),
- calling the tactical *fp_signed_keybind* with Γ and a formula says(ca, keybind(pa, paKey)), and
- calling the tactical *fp_signed_auth* with Γ and a formula says(pa, prp_auth(ma, prp))

As proved earlier, the tactical *search_assmp* is complete on the formula constructors *key_authority*, *keybind* and *prp_server*, the tactical *fp_signed_keybind* is complete on the formula constructors *says* and *keybind*, and the tactical *fp_signed_auth* is complete on the formula constructors *says* and *prp_auth*. Hence the tactic *valid_sig_auth_pf* always finds the proof of valid_sig_prp_auth(ma, prp) with a list of assumptions; thus, the tactical *valid_sig_auth_pf* is complete on the formula constructor *valid_sig_prp_auth*.

C.17 Tactical fp_valid_sig_cdsc

Calling the tactical *fp_valid_sig_cdsc* with a list Γ of assumptions and a true formula valid_sig_component_dsc(ma, m, dsc) is reduced to five calls to other tacticals:

- calling the tactical *search_assmp* with Γ and a formula key_authority(ca),
- calling the tactical *search_assmp* with Γ and a formula keybind(ca, caKey),
- calling the tactical *search_assmp* with Γ and a formula module_authority(ma),
- calling the tactical *fp_signed_keybind* with Γ and a formula says(ca), keybind(ma, maKey)), and
- calling the tactical *fp_signed_dsc* with Γ and a formula says(pa, module_dsc(m, dsc))

As proved earlier, the tactical *search_assmp* is complete on the formula constructors *key_authority*, *keybind* and *module_authority*, the tactical *fp_signed_keybind* is complete on the formula constructors *says* and *keybind*, and the tactical *fp_signed_dsc* is complete on the formula constructors *says* and *module_dsc*. Hence the tactic *valid_sig_cdsc_pf* always finds the proof of valid_sig_component_dsc(ma, m, dsc) with a list of assumptions; thus, the tactical *valid_sig_cdsc_pf* is complete on the formula constructor *valid_sig_component_dsc*.

C.18 Tactical fp_signed_by_ma

Calling the tactical *fp_signed_by_ma* with a list Γ of assumptions and a true formula signed_by_ma(m, dsc, rq) is reduced to three calls to other tacticals:

- calling the tactical *fp_valid_sig_auth* with Γ and a formula valid_sig_prp_auth(ma, rqset)
- calling the tactical *fp_valid_sig_cdsc* with Γ and a formula valid_sig_component_dsc(ma, m, dsc)
- calling the tactical *fp_set* with a formula rq ∈ rqset

As proved earlier, the tactical *fp_valid_sig_auth* is complete on the formula constructor *valid_sig_prp_auth*, the tactical *fp_valid_sig_cdsc* is complete on the formula constructor *valid_sig_component_dsc*, and the tactical *fp_set* is complete on the formula constructors *singleton* and *set_union*.

The above three calls to these tacticals find the proofs of the true formulas. Hence the tactic *signed_by_ma_pf* always finds the proof of signed_by_ma(m, dsc, rq) with a list of assumptions; thus, the tactical *fp_signed_by_ma* is complete on the formula constructor *signed_by_ma*.

C.19 Tactical fp_as_cdsc_combine

The tactical *fp_as_cdsc_combine* is related to the formula constructor *signed_rq*.

The input of the formula constructor *signed_rq* consists of a formula standing for a set of type *module*, a formula of type *component_dsc* and a formula of type *prp_req*. The tactical *fp_as_cdsc_combine* reasons about different cases of the second input formula of type *component_dsc*. The completeness of the tactical *fp_as_cdsc_combine* on *signed_rq* is proved by induction on the structure of the second input formula of *signed_rq*.

As already shown in the proof of the tactical *fp_cdsc_valid*, there exist two formula constructors for type *component_dsc*: *mk_cdsc* and *cdsc_combine*.

The base case occurs when the second input formula of the formula constructor *signed_rq* is built using the constructor *mk_cdsc*. Calling the tactical *fp_as_cdsc_combine* with a list Γ of assumptions and a true formula signed_rq(m, mk_cdsc(exp, imp), rq) is always reduced to calling the tactical *fp_signed_by_ma* with Γ and a formula signed_by_ma(m, mk_cdsc(exp, imp)) by the tactic *sig_rq_sng_pf*.

As already shown, the tactical *fp_signed_by_ma* always finds the proof of a true formula of *signed_by_ma* with a list of assumptions. Therefore the tactical *fp_as_cdsc_combine* finds the proof of signed_rq(m, mk_cdsc(exp, imp), rq) with Γ; thus the tactical *fp_as_cdsc_combine* is complete on *signed_rq* in the base case.

For the inductive step, suppose that the tactical *fp_as_cdsc_combine* finds the proof of a formula signed_rq(m, d1, rq) with a list Γ of assumptions if the formula is true, and the tactical *fp_as_cdsc_combine* finds the proof of a formula signed_rq(m, d2, rq) with Γ if the formula is true.

The only way of building a new formula of type *component_dsc* is using the formula constructor *cdsc_combine*. Therefore calling the tactical *fp_as_cdsc_combine* with Γ and a true formula signed_rq(m, cdsc_combine(d1, d2), rq) falls into one of the following two cases:

C.19.1 Tactic sig_rq_cdsc_combine_i1_pf

If the formula signed_rq(m, d1, rq) is true, calling the tactical *fp_as_cdsc_combine* with Γ and signed_rq(m, cdsc_combine(d1, d2), rq) is reduced to calling the tactical *fp_as_cdsc_combine* with Γ and a formula signed_rq(m, d1, rq), calling the tactical *fp_cdsc_valid* with a formula component_dsc_valid(d1), and calling the tactical *fp_cdsc_valid* with a formula component_dsc_valid(d2) by the tactic *sig_rq_cdsc_combine_i1_pf*.

By the induction hypothesis, the tactical *fp_as_cdsc_combine* finds the proof of signed_rq(m, d1, rq) with Γ. Since the tactical *fp_cdsc_valid* is complete on the formula constructor *component_dsc_valid* as shown before, the tactical *fp_cdsc_valid* always finds the proofs of the formulas component_dsc_valid(d1) and component_dsc_valid(d2).

C.19.2 Tactic sig_rq_cdsc_combine_i2_pf

If the formula signed_rq(m, d1, rq) is not true, then the formula signed_rq(m, d2, rq) must be true, since the formula signed_rq(m, cdsc_combine(d1, d2), rq) is true by the assumption.

Calling the tactical *fp_as_cdsc_combine* with Γ and signed_rq(m, cdsc_combine(d1, d2), rq) is reduced to calling the tactical *fp_as_cdsc_combine* with Γ and a formula signed_rq(m, d2, rq), calling the tactical *fp_cdsc_valid* with a formula component_dsc_valid(d1), and calling the tactical *fp_cdsc_valid* with a formula component_dsc_valid(d2) by the tactic *sig_rq_cdsc_combine_i2_pf*.

By the induction hypothesis, the tactical *fp_as_cdsc_combine* finds the proof of signed_rq(m, d2, rq) with Γ. Since the tactical *fp_cdsc_valid* is complete on the formula constructor *component_dsc_valid* as shown before, the tactical *fp_cdsc_valid* always finds the proofs of the formulas component_dsc_valid(d1) and component_dsc_valid(d2).

Therefore calling the tactical *fp_as_cdsc_combine* finds the proof of a true formula of the formula constructor *signed_rq*; thus the tactical *fp_as_cdsc_combine* is complete on *signed_rq* in the inductive step.

Hence by induction the tactical *fp_as_cdsc_combine* is complete on the formula constructor *signed_rq*.

C.20 Tactical fp_as_set_union

The tactical *fp_as_set_union* is related to the formula constructor *all_signed*.

The input of the formula constructor *all_signed* consists of a formula standing for a set of type *module*, a formula of type *component_dsc* and a formula standing for a set of type *prp_req*. The tactical *fp_as_set_union* reasons about different cases of the third input formula. The completeness of the tactical *fp_as_set_union* on *all_signed* is proved by induction on the structure of the third input formula of *all_signed*.

As already shown in Section C.5, there exist two formula constructors for sets, *singleton* and *set_union*.

Calling the tactical *fp_as_set_union* with a list of assumptions and a true formula of *all_signed* falls into the following two cases.

C.20.1 Case of empty set

If the third input is an empty set, then calling *fp_as_set_union* with a list Γ of assumptions and a formula all_signed(m, dsc, emptyset) is reduced to calling the tactical *fp_emptyset* with a formula isempty(emptyset) by the tactic *all_sig_emptyset_pf*. Since *fp_emptyset* is complete on the formula constructor *isempty*, the call returns the proof of all_signed(m, dsc, emptyset).

C.20.2 Case of non-empty set

The base case occurs when the third input is built using the constructor *singleton*. calling the tactical *fp_as_set_union* with a list Γ of assumptions and a true formula all_signed(m, dsc, singleton(rq)) is always reduced to calling the tactical *fp_as_cdsc_combine* with Γ and a formula signed_rq(m, dsc, rq) by the tactic *all_sig_pf.*

As shown before, the tactical *fp_as_cdsc_combine* always finds a proof of a true formula of *signed_rq* with a list of assumptions. Therefore the tactical *fp_as_set_union* finds the proof of all_signed(m, dsc, singleton(rq)) with Γ; thus the tactical *fp_as_set_union* is complete on *all_signed* in the base case.

For the inductive step, suppose that the tactical *fp_as_set_union* finds the proof of a formula all_signed(m, dsc, rset1) with a list Γ of assumptions if the formula is true, and the tactical *fp_as_set_union* finds the proof of a formula all_signed(m, dsc, rset2) with Γ if the formula is true.

The only way of building a new set out of two sets is using the formula constructor *set_union.* Therefore calling the tactical *fp_as_set_union* with Γ and a true formula all_signed(m, dsc, set_union(rset1, rset2)) is always reduced to calling *fp_as_set_union* with Γ and a formula all_signed(m, dsc, rset1) and calling *fp_as_set_union* with Γ and a formula all_signed(m, dsc, rset2) by the tactic *all_sig_set_union_pf.*

By the induction hypothesis, *fp_as_set_union* finds the proofs of all_signed(m, dsc, rset1) and all_signed(m, dsc, rset2) with Γ. Hence the tactical *fp_as_set_union* is complete on *all_signed* in the inductive step.

By induction, the tactical *fp_as_set_union* is complete on the formula constructor *all_signed.*

C.21 Tactical fp_valid_cdsc

Calling the tactical *fp_valid_cdsc* with a list Γ of assumptions and a true formula signed_component_dsc(m, dsc, rqset) is reduced to two calls to other tacticals: Calling the tactical *fp_cdsc_valid* with a formula component_dsc_valid(dsc), and calling the tactical *fp_as_set_union* with Γ and a formula all_signed(m, dsc, rqset)

As proved earlier, the tactical *fp_cdsc_valid* is complete on the formula constructor *component_dsc_valid*, and the tactical *fp_as_set_union* is complete on the formula constructor *all_signed.*

Since the above two calls to these tacticals find the proofs of the true formulas, the tactic *valid_cdsc_pf* always finds the proof of a true formula signed_component_dsc(m, dsc, rqset) with a list of assumptions; thus, the tactical *fp_valid_cdsc* is complete on the formula constructor *signed_component_dsc.*

C.22 Tactical fp_valid_lib

The tactical *fp_valid_lib* is related to the formula constructor *valid_library.* The input of the formula constructor *valid_library* is a formula standing for a list of sets of type *module*, and a formula standing for a list of sets of type *property.* As shown in Section C.9, there exist two formula constructors for lists, *list_nil* and *list_cons.*

The completeness of the tactical *fp_valid_lib* can be proved by induction on the structure of input formulas of *valid_library.* The semantics of *valid_library* requires that two input formulas should have the same length. That means, for a true formula of *valid_library*, the number of *list_cons* and *list_nil* in the first input list is the same as that in the second input list. Therefore, it never happens that the first input list is built by *list_cons* although the second input list is built by *list_nil*, vice versa.

The base case occurs when the input formulas of the formula constructor *valid_library* is built using the constructor *list_nil.* Calling the tactical *fp_valid_lib* with a list of assumptions and a true formula valid_library(lib, libdsc) is always reduced to calling the tactical *fp_list_nil* with a formula list_is_nil(lib) and calling the tactical *fp_list_nil* with a formula list_is_nil(libdsc) by the tactic *valid_lib_nil_pf.* Since the tactical *fp_list_nil* is complete on *list_is_nil*, *fp_list_nil* finds the proofs of list_is_nil(lib) and list_is_nil(libdsc).

Therefore the tactical *fp_valid_lib* finds the proof of the true formula valid_library(lib, libdsc); thus *fp_valid_lib* is complete on *valid_library* in the base case.

For the inductive step, suppose that the tactical *fp_valid_lib* finds the proof of a true formula valid_library(libtl, libdsctl) with a list of assumptions Γ, where lib = list_cons(set_equiv, m, libtl) and libdsc = list_cons(set_equiv, dsc, libdsctl).

Calling the tactical *fp_valid_lib* with Γ and a true formula valid_library(lib, libdsc) is always reduced to calling the tactical *search_assmp* with Γ and library_dsc(m, dsc) and calling the tactical *fp_valid_lib* with valid_library(libtl, libdsctl) by the tactic *valid_lib_cons_pf.*

The tactical *search_assmp* is complete on the formula constructor *library_dsc*, so calling *search_assmp* with library_dsc(m, dsc) returns the proof. The tactical *fp_valid_lib* finds the proof of valid_library(libtl, libdsctl) by the induction hypothesis; thus the tactical *fp_valid_lib* is complete on *valid_library* in the inductive step.

By induction, the tactical *fp_valid_lib* is complete on the formula constructor *valid_library.*

C.23 Tactical fp_has_prps_set_union′

The tactical *fp_has_prps_set_union′* is related to the formula constructor *has_prp.*

The input of the formula constructor *has_prp* consists of a formula of type *prp_req* and a formula standing for a set of type *property.* The tactical *fp_has_prps_set_union′* reasons about different cases of the second input formula.

The completeness of the tactical *fp_has_prps_set_union′* on *has_prp* is proved by induction on the structure of the second input formula of *has_prp.* By the semantics of *has_prp*, the second input of it cannot be an empty set. That means the second input formula is always built by the formula constructors *singleton* and *set_union.*

The base case occurs when the second input formula of the formula constructor *has_prp* is built using the constructor *singleton.* Calling the tactical *fp_has_prps_set_union′* with a true formula has_prp(rq, singleton(prp)) is always reduced to calling the tactical *fp_prp_match* with a formula prp_match(rq, prp) by the tactic *has_prp_singleton_pf.*

As already shown, the tactical *fp_prp_match* always finds the proof of a true formula. Therefore the tactical *fp_has_prps_set_union′* finds the proof of has_prp(rq, singleton(prp)); thus the tactical *fp_has_prps_set_union′* is complete on *has_prp* in the base case.

For the inductive step, suppose that the tactical *fp_has_prps_set_union′* finds the proof of a formula has_prp(rq, p1) if the formula is true, and the tactical *fp_has_prps_set_union′* finds the proof of a formula has_prp(rq, p2) if the formula is true.

The only way of building a set out of two sets is using the formula constructor *set_union*. Therefore calling the tactical *fp_has_prps_set_union′* with a true formula has_prp(rq, set_union(p1, p2)) falls into one of the following two cases.

C.23.1 Tactic has_prp_sng_union1_pf

If the formula has_prp(singleton(rq), p1) is true, then calling the tactical *fp_has_prps_set_union′* with the formula has_prp(rq, set_union(p1, p2)) is reduced to calling the tactical *fp_has_prps_set_union′* with has_prp(rq, p1) by the tactic *has_prp_union1_pf*. By the induction hypothesis, the tactical *fp_has_prps_set_union′* finds the proof of has_prp(rq, p1).

C.23.2 Tactic has_prps_sng_union2_pf

If the formula has_prp(rq, p1) is not true, then the formula has_prp(rq, p2) must be true, since the formula has_prp(rq, set_union(p1, p2)) is true by the assumption.

Calling the tactical *fp_has_prps_set_union′* with the formula has_prp(rq, set_union(p1, p2)) is reduced to calling the tactical *fp_has_prps_set_union′* with has_prp(rq, p2) by the tactic *has_prp_union2_pf*. By the induction hypothesis, the tactical *fp_has_prps_set_union′* finds the proof of has_prp(rq, p2).

Calling the tactical *fp_has_prps_set_union′* finds the proof of a true formula of *has_prp*; thus *fp_has_prps_set_union′* is complete on *has_prp* in the inductive step. By induction, the tactical *fp_has_prps_set_union′* is complete on the formula constructor *has_prp*.

C.24 Tactical fp_has_prps_set_union

The tactical *fp_has_prps_set_union* is related to the formula constructor *has_property*.

The input of the formula constructor *has_property* consists of a formula standing for a set of type *prp_req* and a formula standing for a set of type *property*. The tactical *fp_has_prps_set_union* reasons about different cases of the first input formula. The completeness of *fp_has_prps_set_union* on *has_property* is proved by induction on the structure of the first input formula of *has_property*.

As already shown in Section C.5, there exists two formula constructors for sets, *singleton* and *set_union*. Calling the tactical *fp_has_prps_set_union* with a true formula falls into the following 2 cases according to the form of the first input.

C.24.1 Case of empty set

If the first input formula is an empty set, then calling *fp_has_prps_set_union* with a true formula has_property(emptyset, prpset) is reduced to calling the tactical *fp_emptyset* with a formula isempty(emptyset) by the tactic *has_prps_emptyset_pf*. Since the tactical *fp_emptyset* is complete on the formula constructor *isempty*, the call returns the proof of the formula has_property(emptyset, prpset).

C.24.2 Case of non-empty set

The base case occurs when the first input formula is built using the constructor *singleton*. Calling *fp_has_prps_set_union* with a true formula has_property_rq(singleton(rq), prpset) is always reduced to calling the tactical *fp_has_prps_set_union′* with a formula has_prp(rq, prpset) by the tactic *has_prps_pf*.

As already shown, the tactical *fp_has_prps_set_union′* always finds the proof of a true formula of *has_prp*. Therefore the tactical *fp_has_prps_set_union* finds the proof of the formula has_property_rq(singleton(rq), prpset); thus the tactical *fp_has_prps_set_union* is complete on *has_property* in the base case.

For the inductive step, suppose that *fp_has_prps_set_union* finds the proof of a formula has_property(rqset1, prpset) if the formula is true, and the tactical *fp_has_prps_set_union* finds the proof of a formula has_property(rqset2, prpset) if the formula is true.

The only way of building a new set out of 2 sets is using the formula constructor *set_union*. Therefore calling the tactical *fp_has_prps_set_union* with a true formula has_property(set_union(rqset1, rqset2), prpset) is always reduced to calling *fp_has_prps_set_union* with a formula has_property(rqset1, prpset) and calling *fp_has_prps_set_union* with a formula has_property(rqset2, prpset) by the tactic *has_prps_set_union_pf*.

By the induction hypothesis, *fp_has_prps_set_union* finds the proofs of has_property(rqset1, prpset) and has_property(rqset2, prpset). The tactical *fp_has_prps_set_union* is complete on *has_property* in the inductive step.

By induction, the tactical *fp_has_prps_set_union* is complete on *has_property*.

C.25 Tactical fp_exp_prps′

The tactical *fp_exp_prps′* is related to the formula constructor *exp_rprp*.

The input of the formula constructor *exp_rprp* consists of a formula of type *prp_req* and a formula of type *component_dsc*. The tactical *fp_exp_prps′* reasons about different cases of the second input formula. The completeness of the tactical *fp_exp_prps′* on *exp_rprp* is proved by induction on the structure of the second input formula of *exp_rprp*.

As already shown in the proof of the tactical *fp_cdsc_valid*, there exists two formula constructors for type *component_dsc*: *mk_cdsc* and *cdsc_combine*.

The base case occurs when the second input formula of the formula constructor *exp_rprp* is built using the constructor *mk_cdsc*. calling the tactical *fp_exp_prps′* with a true formula exp_rprp(rq, mk_cdsc(exp, imp)) is always reduced to calling the tactical *fp_has_prps_set_union′* with a formula has_prp(rq, exp) by the tactic *exp_rprps_cdsc_pf.*

As already shown, the tactical *fp_has_prps_set_union′* always finds the proof of a true formula of *has_prp.* Therefore the tactical *fp_exp_prps′* finds the proof of the formula exp_rprp(rq, mk_cdsc(exp, imp)); thus the tactical *fp_exp_prps′* is complete on *exp_rprp* in the base case.

For the inductive step, suppose that the tactical *fp_exp_prps′* finds the proof of a formula exp_rprp(rq, d1) if the formula is true, and the tactical *fp_exp_prps′* finds the proof of a formula exp_rprp(rq, d2) if the formula is true.

The only way of building a new *component_dsc* formula is using the formula constructor *cdsc_combine.* Therefore calling the tactical *fp_exp_prps′* with a true formula exp_rprp(rq, cdsc_combine(d1, d2)) falls into one of the following two cases.

C.25.1 Tactic exp_rprp_cdsc_i1_pf

If the formula exp_rprp(rq, d1) is true, calling the tactical *fp_exp_prps′* with the formula exp_rprp(rq, cdsc_combine(d1, d2)) is reduced to calling *fp_exp_prps′* with exp_rprp(rq, d1) by the tactic *exp_rprp_cdsc_i1_pf.* By the induction hypothesis, the tactical *fp_exp_prps′* finds the proof of exp_rprp(rq, d1)

C.25.2 Tactic exp_rprp_cdsc_i2_pf

If the formula exp_rprp(rq, d1) is not true, then the formula exp_rprp(rq, d2) should be true in order to make the formula exp_rprp(rq, cdsc_combine(d1, d2)) true. Calling the tactical *fp_exp_prps′* with exp_rprp(rq, cdsc_combine(d1, d2)) is reduced to calling the tactical *fp_exp_prps′* with exp_rprp(rq, d2) by the tactic *export_rprps_cdsc_i2_pf.* By the induction hypothesis, the tactical *fp_exp_prps′* finds the proof of exp_rprp(rq, d2)

Therefore calling the tactical *fp_exp_prps′* finds the proof of a true formula of *exp_rprp*; thus the tactical *fp_exp_prps′* is complete on *exp_rprp* in the inductive step.

By induction, the tactical *fp_exp_prps′* is complete on the formulas of *exp_rprp.*

C.26 Tactical fp_exp_prps

The tactical *fp_exp_prps* is related to the formula constructor *export_required_prps.*

The input of the formula constructor *export_required_prps* consists of a formula standing for a set of type *prp_req* and a formula of type *component_dsc.* The tactical *fp_exp_prps* reasons about different cases of the first input formula. The completeness of the tactical *fp_exp_prps* on *export_required_prps* is proved by induction on the structure of the first input formula of *export_required_prps.*

As already shown in Section C.5, there exists three formula constructors for sets, *emptyset*, *singleton* and *set_union.*

Calling the tactical *fp_exp_prps* with a true formula falls into the following 2 cases according to the form of the first input formula.

C.26.1 Case of empty set

If the first input formula is an empty set, then calling *fp_exp_prps* with a true formula export_required_prps(emptyset, dsc) is reduced to calling the tactical *fp_emptyset* with a formula isempty(emptyset) by the tactic *export_rprps_empty_pf.* Since the tactical *fp_emptyset* is complete on the formula constructor *isempty*, the call returns the proof of the formula export_required_prps(emptyset, dsc)

C.26.2 Case of non-empty set

The base case occurs when the first input formula is built using the constructor *singleton.* Calling the tactical *fp_exp_prps* with a true formula export_required_prps(singleton(rq), dsc) is always reduced to calling the tactical *fp_exp_prps′* with a formula exp_rprp(rq, dsc) by the tactic *export_rprps_sng_pf.*

As already shown, the tactical *fp_exp_prps′* always finds the proof of a true formula of *exp_rprp.* Therefore the tactical *fp_exp_prps* finds the proof of export_required_prps_rq(singleton(rq), dsc); thus the tactical *fp_exp_prps* is complete on the formula constructor *export_required_prps* in the base case.

For the inductive step, suppose that the tactical *fp_exp_prps* finds the proof of a formula export_required_prps(rqset1, dsc) if the formula is true, and *fp_exp_prps* finds the proof of a formula export_required_prps(rqset2, dsc) if the formula is true.

The only way of building a new set out of 2 sets is using the formula constructor *set_union.* Therefore calling the tactical *fp_exp_prps* with a true formula export_required_prps(set_union(rqset1, rqset2), dsc) is always reduced to calling *fp_exp_prps* with export_required_prps(rqset1, dsc) and calling *fp_exp_prps* with export_required_prps(rqset2, dsc) by the tactic *export_rprps_set_union_pf.*

By the induction hypothesis, *fp_exp_prps* finds the proofs of export_required_prps(rqset1, dsc) and export_required_prps(rqset2, dsc). Hence the tactical *fp_exp_prps* is complete on the formula constructor *export_required_prps* in the inductive step.

By induction, the tactical *fp_exp_prps* is complete on the formula constructor *export_required_prps.*

C.27 Tactical fp_st_imprt_cdsc

The tactical *fp_st_imprt_cdsc* is related to the formula constructor *satisfy_imprt_cdsc.* The input of *satisfy_imprt_cdsc* is a set of type *prp_req* and a list of sets of type *property.* As mentioned in Section C.9, there exist two constructors for a *list* type, *list_nil* and *list_cons.*

Calling the tactical *fp_st_imprt_cdsc* with a true formula satisfy_imprt_cdsc(imp, explist) semantically means searching a matching list member from the list *explist.* Therefore the completeness of the tactical *fp_st_imprt_cdsc* can be proved by induction on the length of *explist.*

Suppose that the tactical *fp_st_imprt_cdsc* with a true formula satisfy_imprt_cdsc(imp, explist). By the semantics of *satisfy_imprt_cdsc*, explist in a true formula of *satisfy_imprt_cdsc* cannot be a nil list.

The base case occurs when the length of explist is one. That means, it is the case of explist = list_cons(exp, list_nil). Since the formula satisfy_imprt_cdsc(imp, explist) is true, imp of *prp_req* must match the set exp of *property*. Therefore the call is reduced to calling the tactical *fp_has_prps_set_union* with a formula has_property(imp, exp). As proved in Section C.24, *fp_has_prps_set_union* always finds the proof of *has_property*. Hence *fp_st_imprt_cdsc* always finds the proof of a true formula of *satisfy_imprt_cdsc* in the base case.

For the inductive step, suppose that *fp_st_imprt_cdsc* finds the proof of a true formula satisfy_imprt_cdsc(imp, explist) for any imp when the length of explist is k.

Then calling *fp_st_imprt_cdsc* with a true formula satisfy_imprt_cdsc(imp', list_cons(set_equiv, exp, explist)), the length of whose second argument is $k + 1$, falls into the following two cases.

C.27.1 Tactic satisfy_imprt_hd_pf

It is the case when the set imp' of *prp_req* matches the set exp of *property*. Then the call is reduced to calling the tactical *fp_has_prps_set_union* with a formula has_property(imp, exp) by the tactic *satisfy_imprt_hd_pf*.

Since *fp_has_prps_set_union* is complete on the formula constructor *has_property*, *fp_st_imprt_cdsc* finds the proof in this case.

C.27.2 Tactic satisfy_imprt_tl_pf

If the set imp' of *prp_req* doesn't match the set exp of *property*, then the tactic *satisfy_imprt_tl_pf* is applied, and the call is reduced to calling the tactical *fp_list_valid* with a formula list_valid(set_equiv, imp), and calling the tactical *fp_st_imprt_cdsc* with a formula satisfy_import_cdsc(imp, explist).

The tactical *fp_list_valid* always finds the proof of a true formula of *list_valid*, and *fp_st_imprt_cdsc* finds the proof of satisfy_import_cdsc(imp, explist) by the induction hypothesis; thus, the tactical *fp_st_imprt_cdsc* finds the proof in the inductive step.

By induction, the tactical *fp_st_imprt_cdsc* is complete on the formula constructor *satisfy_imprt_cdsc*.

C.28 Tactical fp_st_imprt

The tactical *fp_st_imprt* is related to the formula constructor *satisfy_import_req*.

The input of the formula constructor *satisfy_import_req* is a formula standing for a list of sets of type *prp_req* and a formula standing for a list of sets of type *property*. The tactical *fp_st_imprt* reasons about different cases of the first input formula. The completeness of the tactical *fp_st_imprt* is proved by induction on the structure of the first input formula of *satisfy_import_req*.

As shown in Section C.9, there exist two formula constructors for lists, *list_nil* and *list_cons*.

The base case occurs when the input first formula of *satisfy_import_req* is built using the constructor *list_nil*. Calling the tactical *fp_st_imprt* with a true formula satisfy_import_req(list_nil, libdsc) is always reduced to calling the tactical *fp_list_nil* with a formula list_is_nil(list_nil) by the tactic *satisfy_imprt_nil_pf*. Since the tactical *fp_list_nil* is complete on *list_is_nil*, *fp_list_nil* finds the proof of list_is_nil(list_nil).

Therefore the tactical *fp_st_imprt* finds the proof of the true formula satisfy_import_req(list_nil, libdsc); thus *fp_st_imprt* is complete on *satisfy_import_req* in the base case.

For the inductive step, suppose that the tactical *fp_st_imprt* finds the proof of a true formula satisfy_import_req(imptl, libdsc) where implist = list_cons(set_equiv, imp, imptl).

Calling the tactical *fp_st_imprt* with a true formula satisfy_import_req(implist, libdsc) is reduced to calling the tactical *fp_st_imprt_cdsc* with a formula satisfy_imprt_cdsc(imp, libdsc) and calling the tactical *fp_st_imprt* with a formula satisfy_import_req(imptl, libdsc) by the tactic *satisfy_imprt_cons_pf*.

The tactical *fp_st_imprt_cdsc* is complete on *satisfy_imprt_cdsc*, so calling *fp_st_imprt_cdsc* with a true formula of *satisfy_imprt_cdsc* returns the proof. The tactical *fp_st_imprt* finds the proof of satisfy_import_req(imptl, libdsc) by the induction hypothesis; thus the tactical *fp_st_imprt* is complete on *satisfy_import_req* in the inductive step.

By induction, the tactical *fp_st_imprt* is complete on the formula constructor *satisfy_import_req*.

C.29 Tactical fp_prv_prps

The tactical *fp_prv_prps* is related to the formula constructor *provide_enough_prps*.

The input of the formula constructor *provide_enough_prps* is a formula of type *component_dsc*, a formula standing for a list of type *module* and a formula standing for a list of sets of of type *property*. The tactical *fp_prv_prps* reasons about different cases of the first input formula of type *component_dsc*. The completeness of the tactical *fp_prv_prps* is be proved by induction on the structure of the first input formula.

As shown in Section C.12 There exist two formula constructors for type *component_dsc*, *mk_cdsc* and *cdsc_combine*.

The base case occurs when the first input formula of the formula constructor *provide_enough_prps* is built using the constructor *mk_cdsc*. Calling the tactical *fp_prv_prps* with a list Γ of assumptions and a true formula provide_enough_prps(mk_cdsc(exp, imp), lib, libdsc) is always reduced to calling the tactical *fp_list_valid* with a formula list_valid(set_equiv, imp), calling the tactical *fp_valid_lib* with Γ and a formula valid_library(lib, libdsc), and calling the tactical *fp_st_imprt* with a formula satisfy_import_req(imp, libdsc) by the tactic *prv_prps_pf*. Since *fp_list_valid* is complete on *list_valid*, *fp_list_valid* finds the proof of list_valid(set_equiv, imp). Similarly, *fp_valid_lib* finds the proof of valid_library(lib, libdsc) with Γ, and *fp_st_imprt* finds the proof of satisfy_import_req(imp, libdsc).

Therefore the tactical *fp_prv_prps* finds the proof of the true formula provide_enough_prps(mk_cdsc(exp, imp), lib, libdsc) with Γ; thus *fp_prv_prps* is complete on *provide_enough_prps* in the base case.

For the inductive step, suppose that the tactical *fp_prv_prps* finds proofs of a true formula provides_enough_prps(c1, lib, libdsc) and a true formula provides_enough_prps(c2, lib, libdsc).

Calling the tactical *fp_prv_prps* with a list Γ of assumptions and a true formula provides_enough_prps(cdsc_combine(c1, c2), lib, libdsc) is always reduced to calling the tactical *fp_cdsc_valid* with a formula cdsc_valid(d1), calling the tactical *fp_cdsc_valid* with a formula cdsc_valid(d2), calling the tactical *fp_prv_prps* with Γ and a formula provides_enough_prps(c1, lib, libdsc), and calling the tactical *fp_prv_prps* with Γ and a formula provides_enough_prps(c2, lib, libdsc) by the tactic *prv_prps_cdsc_combine_pf*.

The tactical *fp_cdsc_valid* is complete on the formula constructor *cdsc_valid*, so calling *fp_cdsc_valid* with true formulas cdsc_valid(d1) and cdsc_valid(d2) returns proof. The tactical *fp_prv_prps* finds proofs of provides_enough_prps(c1, lib, libdsc) and provides_enough_prps(c2, lib, libdsc) by the induction hypothesis; thus the tactical *fp_prv_prps* is complete on *provide_enough_prps* in the inductive step.

By induction, the tactical *fp_prv_prps* is complete on the formula constructor *provide_enough_prps*.

C.30 Tactical findproof

Calling the tactical *findproof* with a list Γ of assumptions and a true formula ok_to_link(m, dsc, lib, libdsc, rqset) is reduced to three calls to other tacticals:

- calling the tactical *fp_valid_cdsc* with Γ and a formula valid_component_dsc(m, dsc, rqset)
- calling the tactical *fp_prv_prps* with Γ and a formula provide_enough_prps(dsc, lib, libdsc)
- calling the tactical *fp_exp_prps* with a formula export_required_prps(rqset, dsc)

As proved earlier, the tactical *fp_valid_cdsc* is complete on the formula constructor *valid_component_dsc*, the tactical *fp_prv_prps* is complete on the formula constructor *provide_enough_prps*, and the tactical *fp_exp_prps* is complete on the formula constructor *export_required_prps*.

Hence each tactical finds the proof of a true formula of its relating formula constructor, and the tactic *ok_to_link_pf* always finds the proof of a true formula ok_to_link(m, dsc, lib, libdsc, rqset) with a list of assumptions; thus, the tactical *findproof* is complete on the formula constructor *ok_to_link*.

□