

# Defining and Formally Specifying the Behavior of Cooperative Distributed Systems

Toufik Taibi

Department of Electrical and Computer Engineering  
University of Western Ontario  
London N6A5B9, Ontario, Canada  
e-mail:ttaibi@uwo.ca

## Abstract

*Cooperative Distributed Systems (CDS) represent a class of open distributed systems, where entities are willing to share their capabilities with others. This paper first presents a high-level architecture of CDS and provides an informal description of the behavior (defined by the set of actions that can be performed) of such a system. Secondly, the behavior of the system is formally specified using the Temporal Logic of Actions (TLA). Lastly, the behavior of the system is model-checked using TLC—the TLA model checker in order to validate that the invariants and properties defined were satisfied by the behavior. To our knowledge, there are no recent work on formalizing the generic behavior of a CDS.*

## 1 Introduction

Traditionally, a distributed system was defined as a collection of independent “computers” that appears to its users as a single coherent system [8]. Here the term “computers” abstracts the hardware, operating system, middleware and the applications running on top of them. A distributed system can be closed by design such as the case of virtual organizations [1] or open. Openness makes the system dynamic as entities may join and leave the system anytime.

In Distributed systems, entities rely on other to perform some tasks. We call this the capability-dependency problem. One may think, why don't we build self-contained entities i.e. monolithic entities able to perform all possible tasks within a given application domain? The answer to this question is

that spreading capabilities among coordinating entities make it easy to manage the complexity of the system as well easily support other system quality attributes such as availability, modifiability, performance, etc. instead of merely achieving desired functionality.

Cooperative Distributed Systems (CDS) represent a special type of open distributed systems where entities are willing to share their capabilities with others. Today, businesses rely on thousands of different software applications, each of which developed using different programming languages, run on different operating systems and hardware platforms, and may define and represent concepts differently. As a result, it is very difficult for different applications to communicate with one another and share their capabilities in a coordinated way.

Due to the dynamic nature of the system, an entity or its required capability may not be available when needed. This creates a challenge for requesting entities to keep track of other entities and their capabilities. In order to facilitate the development of efficient and effective CDS, new integration and coordination approaches need to be explored and developed in order to allow entities to seamlessly share their capabilities. The main objective of the "integration" is to hide the distribution nature as well as the heterogeneity and provide a virtual homogeneous environment that can be accessed anywhere and anytime. CDS integration requires the provision of right information and services at the right time. This in turn, requires an explicit knowledge of the dynamically changing available capabilities in the system.

Throughout the years, CDS design and architectures made many assumptions for making their solutions feasible. However, with the explosive growth of Internet and proliferations of applications using the World Wide Web, these assumptions can not longer be made. As such, there is a pressing need to tackle issues that openness, dynamism and heterogeneity are bringing to current and future CDS [9]. However, first of all we need to define an architecture and a sound and generic interaction protocol which can be easily extended to accommodate efficient and effective solutions to the above mentioned issues. We define the behavior of a CDS as a set of actions to be executed in the system. These actions make-up an interaction protocol.

The Temporal Logic of Actions (TLA) [2] was used to formally specify the behavior of a CDS. TLA is a logic that combines features of both linear temporal logic as well as a logic of actions. It is well suited to specify and reason about concurrent and distributed systems. Unlike other logics, TLA possesses a fully-fledged language called TLA+ [3] that allows the specifying the behavior of virtually any system. For years it has been successfully used to specify hardware systems and is gaining momentum when it comes to spec-

ifying software systems. Moreover, TLA+ has a model checker named TLC that allows to check if a given model satisfies a given TLA formula as well as allowing the verification of the satisfiability of invariants and properties of the system.

The rest of the paper is organized as follows. Section 2, provides a description of the high-level architecture of a CDS. Section 3 provides a detailed description of actions performed in the CDS. Section 4 provides an overview of TLA, TLA+ and TLC, while section 5 describes how TLA+ was used to specify the behavior of the entire system and how TLC was used to model-check its behavior and show indeed that invariants and properties are satisfied by the specification. Section 6 describes how specifications can be validly refined, while section 7 concludes the paper and highlight ongoing and future research work.

## 2 A High-Level Architecture for CDS

A CDS is made-up of entities continuously playing either or both of the roles of requester or provider. Figure 1 depicts the high-level architecture of such a system. Whenever an entity wants to join the system it has to do so through a registration service which provides it with a unique identity. Similarly, when an entity wants to leave the system it does so through the registration service which relinquishes its identity and may give it to another entity joining the system. The registration service can be queried by any entity in the system to check whether a given identity is registered or not with the system.

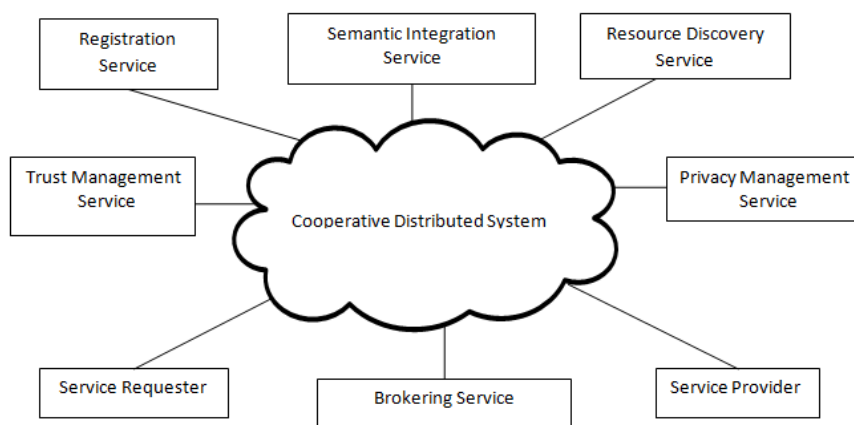


Figure 1: High-Level Architecture of a Cooperative Distributed System

Any entity can advertise its capabilities with the resource discovery. Brokers also need to advertise the fact that they can offer brokering services. Any registered entity in the system (including brokers) can query the resource discovery to get a list of providers having capabilities matching a certain request. As such the resource discovery service plays here the role of a matchmaker. It is then the responsibility of this entity to choose the provider(s) which best fulfills its selection criteria (or the selection criteria of its requester in case of a broker).

Entities requiring a full mediation service, should go through a broker. The basic scenario for using a broker is that an entity acting as a requester requests through the broker for a given service. The broker in turn find the required provider(s) through the resource discovery service. Next, the broker select the best provider(s) available according to criteria set by the requester in its request. The broker then calls the service(s) of the chosen provider(s) and get back with a reply to the requester in a transparent manner. As seen from the description above the broker continuously switch from playing the role of a requester to playing the role of a provider. As such, “requester” and “provider” as just roles played by entities. As shown in Figure 1, besides the services we described, the system has three more services—privacy management, trust managements and semantic integration, which are keys success factors for the system.

Privacy is defined as the degree of information that an entity decides to show or hide to other entities in the system [5]. For example a requester could opt to hide either its identity or its request or both. On the other hand, a provider could opt to hide either its identity or its capabilities or both. The tasks of the broker (and to a certain extend the resource discovery) are more complex and thus more time consuming if it handles entities (requesters, providers) opting to hide some or all of the above attributes.

Trust can be defined as the degree of confidence that an entity is capable of acting reliably and securely in a particular transaction. Trust management thus entails the collection of information necessary for defining trust values of entities and continuously monitoring and adjusting such values [6]. The relationship between trust and privacy is clear. Depending on the level of trust between an entity (requester or provider) and a given broker (and to a certain extend the resource discovery), it can decide to hide/reveal part/all of its attributes and by doing so either increase or decreases the complexity and thus cost of the entire mediation(or matchmaking) process.

The semantic integration service is responsible for the semantic integrations of (incompatible) concepts used by the requester and the provider as to get them to reach a common understanding. This service is becoming a crucial part of current/future cooperative distributed systems which are

heterogeneous in nature [4].

It is to be noted that the above services (privacy management, trust management and semantic integration) are not by any means centralized services (as it may look in Figure 1 and should be implemented in a completely decentralized fashion).

This paper is aimed at providing a formal treatment (using TLA) to the behavior associated with the interaction between service providers, service requesters, brokers, registration service and resource discovery service. The other services will be the focus of future work. Indeed, the formal specification of core interaction protocol of a CDS can be validly refined by augmenting it with the specification of interaction protocols involving privacy, trust and semantic integration (see section 6).

### 3 Interaction Protocol of the Proposed System

In this section, we will provide an overview of the actions involved in CDS. These actions make-up the interaction protocol of the system. The formal specification of these actions will be provided in section 5. Figure 2 depicts actions involved in the registration process where each entity (requester, provider, broker) can either join or leave the system through the registration service. As noted earlier, entities can play different roles. Each time an entity registers, it is given a unique identity (not being used). When an entity wants to leave the system, it should not have any pending request/reply to be processed. These will be formally specified in section 5.

Figure 3 depicts the actions involved in the CDS. A provider uses action *Provider\_Advertise* to advertise its capabilities with the resource discovery. A capability is defined as a tuple made-up of elements of the following types:  $\langle provID, capDescr, servName, servParam, QoS \rangle$ . *provID* is a unique *ID* given to the provider when it first joined the system. *capDescr* describes the capability of the provider which is advertised by this action. This can be done using text or a (formal) capability description language. *servName* represents the name of the service to be called by requesters (or brokers), *servParam* represents the service parameters, while *QoS* represents the Quality of Service (QoS) offered. A provider can make as many advertisement as the services it has to offer. Similarly, a broker uses action *Broker\_Advertise* to advertise with the resource discovery for the fact that it can do brokering services. Its advertisement is defined as a tuple made-up of two elements of the following types:  $\langle brID, brDescr \rangle$ . *brID* is a unique *ID*

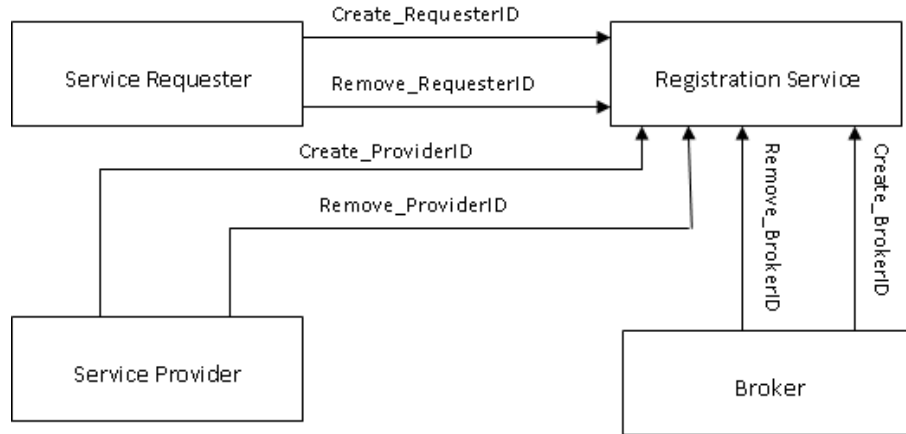


Figure 2: Actions Involved in the Registration Process

given to a broker when it first joined the system. *brDescr* is a singleton set that contains the keyword “Broker”. Any time a provider or broker wants to unadvertise its capabilities it simply calls actions *Provider\_Unadvertise* or *Broker\_Unadvertise* respectively.

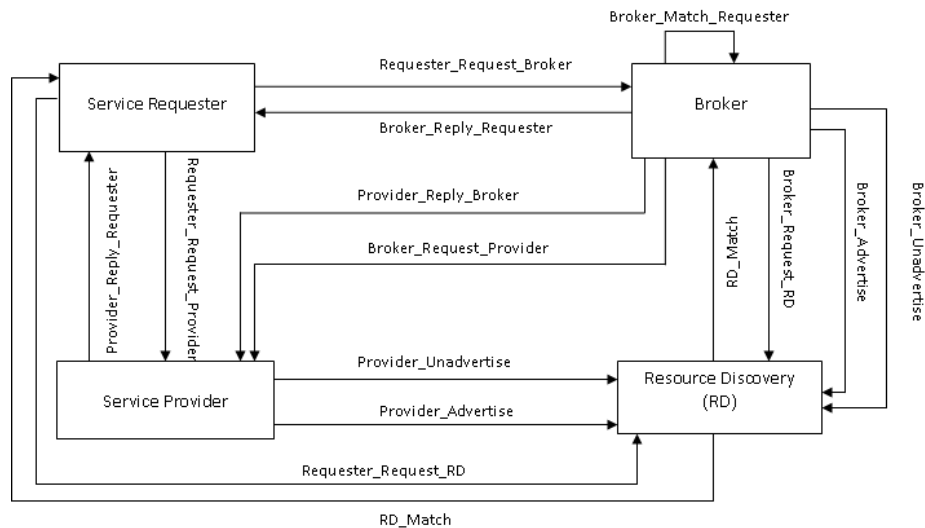


Figure 3: Actions Involved in the Cooperative Distributed System

Requesters have two options to find service providers. They can either submit their request to the resource discovery if they just want a match-making service or to a broker if they want a full-mediation service. In the

case where they just want a matchmaking service they need to use actions *Requester\_Request\_RD* and *RD\_Match* to make a request and get a list of matching providers respectively. A request is defined as a tuple made-up of elements of the following types:  $\langle reqID, rqtrID, reqDescr, pref \rangle$ . *reqID* is a unique *ID* given to a requester when it first joined the system. *rqtrID* is a unique transaction ID that is generated for each transaction made by a requester. *reqDescr* is a request description. Similarly to what has been said about *capDescr*, *reqDescr* can be described using either plain text or a (formal) request description language. *pref* is a list of preferences that a requester wants. This could include and are not limited to: Quality of Service (QoS) requirements, required level of trust from the provider, cost, etc. This is mainly used when the request is made through a broker. In its simplest form, the resource discovery matches between *reqDescr* and *capDescr*. Of course a “valid” refinement (see section 6) can make this matching more sophisticated. Consequently, the requester can select the provider(s) that can best fulfill its criteria (for example that its *pref* matches *QoS* of the provider) and call them using actions *Requester\_Request\_Provider* (using *servName* and *servParam* defined earlier) and get a reply through action *Provider\_Reply\_Requester*.

If a requester wishes to use full-mediation it must use a broker. Since brokers advertise their services with the resource discovery similar to providers, a requester can get broker IDs from querying the resource discovery using actions *Request\_Request\_RD* and *RD\_Match*. The brokering process is initiated by a requester making a request to the broker, represented here my action *Requester\_Request\_Broker*. A request has the same structure as defined in action *Requester\_Request\_RD*.

When a broker receives a request from a requester it in turn creates from it, its own request to be sent to the resource discovery service using action *Broker\_Request\_RD*. In this request, the broker uses its own request ID (taken from the type *brID*) and its own transaction ID (taken from the type *brtrID*). The resource discovery uses a matchmaking mechanism (action *RD\_Match*) to come-up with a list of providers having *capDescr* that best match *reqDescr* and sends it to the broker. The broker makes the final decision on choosing the provider(s) that can best fulfill the request by matching *pref* with *QoS* of the “shortlisted” providers (this is done using action *Broker\_Match\_Requester*). Again this process is at its simplest form and can be made sophisticated by “valid” refinements (see section 6). Using the information of the chosen provider(s) the broker calls the service of the relevant provider(s) using (using action *Broker\_Request\_Provider*) using the service names and parameters provided by the resource discovery service. Next, the result of calling the services (that comes through action *Provider\_Reply\_Broker*) is send by the broker to the requester using action

*Broker\_Reply\_Requester.*

## 4 Overview of TLA, TLA+ and TLC

TLA was developed by Lamport for describing and reasoning about concurrent and distributed systems. The semantics of TLA is defined in terms of states, where a state is a function from the set of variables to the set of values. A state function is a non-Boolean expression built from variables and constant symbols. As such a state function is a mapping from the collection of states to the collection of values. A state predicate (or predicate) is a Boolean expression built from variable and constant symbols. As such, a state predicate is a mapping from states to Booleans. An action is a Boolean-valued expression formed from variables, primed variables and constant symbols. An action represents a relation between old states and new states, where the unprimed variables refer to the old state and the prime variables refer to the new state. As such, an action is a function that assigns a Boolean to a pair of states. A pair of successive states is called a step. For any state function or predicate  $F$ , we define  $F'$  to be the expression obtained by replacing each variable  $v$  in  $F$  by the primed variable  $v'$ . For any action  $A$ , *Enabled*  $A$  is a predicate that is true for a state iff it is possible to take an  $A$  step starting in that state.

Actions can contain parameter symbols which do not represent known values like 1 or "abc". However, unlike the variables we have considered so far, the value of a parameter does not change. It must be the same in the old and new state. The parameter denotes some fixed but unknown value. It is thus called a *rigid variable*. The variables introduced are called flexible variables, or simply variables. A temporal formula is built from elementary formulas using Boolean operators (basically  $\wedge$  and  $\neg$  as the others can be derived from these two) and the unary operator  $\square$  (always). The operator  $\diamond$  (eventually) can be derived from  $\square$  by  $\diamond F \triangleq \neg \square \neg F$ . The symbol " $\triangleq$ " means "by definition".

A behavior is an infinite sequence of states. The meaning of a formula  $F$ , is a Boolean-valued function on behaviors. Another component of TLA syntax is the stuttering operator on actions. A stuttering on action  $A$  under the vector of variables  $f$  occurs when either the action  $A$  occurs or the variables in  $f$  remain unchanged (while either some other independent action occurs or the system remains idle). The stuttering operator and its dual the angle operator are defined as follows.  $[A]_f \triangleq A \vee (f' = f)$  and  $\langle A \rangle_f \triangleq A \wedge (f' \neq f)$ .

Specification are usually written to handle two types of properties for a system—*safety* and *liveness*. *Safety* properties say what a system must not



do, while *liveness* properties say that something does happen. *Safety* is handled by the way specifications are written, which implicitly define behaviors that could satisfy them. *Liveness* is handled through “explicit” *fairness* requirement. TLA defines two types of *fairness* properties: *weak fairness* and *strong fairness* as follows:

- $WF_f(A) \triangleq (\Box \Diamond \langle A \rangle_f) \vee ((\Box \Diamond \neg Enabled \langle A \rangle_f)$ , which means that either infinitely many A steps occur or A is infinitely often disabled.
- $SF_f(A) \triangleq (\Box \Diamond \langle A \rangle_f) \vee ((\Diamond \Box \neg Enabled \langle A \rangle_f)$ , which means that either infinitely many A steps occur or A is eventually disabled forever.

Alternatively, *weak fairness* and *strong fairness* can be defined as follows [3]:

- *Weak fairness* of A asserts that an A step must eventually occur if A is *continuously* enabled.
- *Strong fairness* of A asserts that an A step must eventually occur if A is *continually* enabled.

*Continuously* means without interruption. *Continually* means repeatedly, possibly with interruptions. As such *strong fairness* implies *weak fairness*.

In TLA, systems are usually represented as a conjunction of an initial condition, an action that is continually repeated under stuttering, and a set of fairness conditions. As such, TLA formulas can be written as  $\Phi \triangleq Init_\Phi \wedge \Box [N]_f \wedge F$ , where:

- $Init_\Phi$  is a predicate specifying the initial values of variables.
- $N$  is the system’s next-state relation (disjunction of actions).
- $f$  is an n-tuple of variables.
- $F$  is the conjunction of formulas of the form  $SF_f(A)$  and/or  $WF_f(A)$ , where A represents an actions or a disjunction of actions.

TLA+ is a high-level language based on set theory, predicate logic, and TLA. It is a complete specification language for TLA. To get the reader a feel of a TLA specification, let’s model a light switch and provide concrete examples of all concepts defined above.

$$\text{Invariant} \triangleq S \in \{0, 1\}$$

$$\text{Init} \triangleq S = 0$$

$On \triangleq S = 0 \wedge S' = 1$   
 $Off \triangleq S = 1 \wedge S' = 0$   
 $Next \triangleq On \vee Off$   
 $Spec \triangleq Init \wedge \square[Next]_S \wedge WF_S(Off)$   
 Theorem  $Spec \Rightarrow \square Invariant$

The above specification has only two possible states. The state where the light switch is off ( $S = 0$  which is also the initial state) and the state where the light switch is on ( $S = 1$ ).  $S' = 1$  is an example of a state function that assigns the value 1 to  $S$  after action  $On$  is executed. The following are examples of behaviors that satisfies formula  $Spec$ :

- $[S = 0] \rightarrow [S = 1] \rightarrow [S = 0] \rightarrow [S = 1] \rightarrow [S = 1] \rightarrow [S = 1] \rightarrow [S = 0] \rightarrow [S = 1] \rightarrow \dots$
- $[S = 0] \rightarrow [S = 0] \rightarrow [S = 1] \rightarrow [S = 1] \rightarrow [S = 0] \rightarrow [S = 0] \rightarrow [S = 0] \rightarrow \dots$

The *weak fairness* condition stipulates that action  $Off$  should be executed infinitely often (provided it is enabled) as we wish the light to be off to save energy especially if there are many stuttering steps in which  $S = 1$  and the light is not being used.

Model checking is the process of checking whether a given model satisfies a given logical formula (generally a temporal logic formula). Model checkers can explore traces allowed by the model, possibly detecting deadlock or violation of invariants. Moreover, they can assist in the formal verification of properties. TLC [3] is a model checker for specifications written in TLA+. TLC is an explicit-state, on-the-fly model checker. TLA+ and TLC have been successfully used by hardware engineers to check the correctness of hardware protocols. It is gaining popularity among software engineers to specify and check concurrent algorithms and protocols for software systems.

TLC can analyze the state space of finite instances of TLA+ models. In addition to the TLA+ model, TLC requires a configuration file that defines the finite-state instance to analyze and declares the specifications and the properties to verify. TLC needs to know explicitly (through the configuration file) which of the formulas represent the system specification to analyze, constants, invariants and properties. Figure 4 provides a sample configuration file.

The above configuration file defines concrete instances of a TLA+ module by defining the sets  $S_1$  and  $S_2$ . The keyword **SPECIFICATION** indicates the formula representing the main system specification. The keyword **CONSTANTS** provides values to the constants defined in the specification and it is through

```
SPECIFICATION Spec
```

```
CONSTANTS
```

```
    S1 = {a1, a2}
```

```
    S2 = {b1, b2, b3, b4}
```

```
INVARIANTS Inv
```

```
PROPERTY Prop
```

Figure 4: Sample TLC Configuration File

these values that a model is created. Properties to be checked are specified with the `PROPERTY Prop` statement. This means that TLC checks if  $Spec \Rightarrow Prop$  is valid for the entire state space. Invariants to be checked are specified with the statement `INVARIANT Inv`, which requires checking that  $Spec \Rightarrow \Box Inv$  for every step of a behavior.

TLC firsts checks the syntactic and semantic correctness and well-formedness of a TLA+ specification. It then computes the graph of reachable states for the instance of the model defined by the configuration file, while verifying the invariants. Finally, the temporal properties are verified over the state space. TLC also reports the number of states it generated during its analysis, the number of distinct states, and the depth of the state graph (the length of the longest path). For small models TLC run completes after few seconds. Trying to analyze somewhat larger models, leads to the well-known problem of state-space explosion.

## 5 Formal Specification of the System Behavior

This section describes the formal specification of the system behavior described (informally) in the previous section. For the sake of clarity the specification has been split into 6 parts defined as TLA+ modules (Figure 9-Figure 14) and put into an Appendix. Furthermore, the specification uses the generic module named *Identities* (Figure 8) which in fact models the behavior of the registration service for creating and removing identities for requesters, brokers and providers.

The modules used have been extensively commented as to be as self-explanatory as possible. The specification used 12 constants (which need to be populated in a configuration file for the model-checking to work), 18 variables (which values represent the states of the system) and 20 actions that are executed concurrently. Figure 5 depicts the configuration file we used to model-check the system behavior. It represents the system model to

test the interaction protocol.

```
SPECIFICATION Spec
CONSTANTS
  Requesters_IDs={r1,r2}
  rqtrID={t1,t2}
  reqDescr={searching,Broker}
  pref={good}
  Brokers_IDs={b1,b2}
  brDescr={Broker}
  brtrID={t3,t4}
  Providers_IDs={p1,p2}
  capDescr={searching}
  servName={search}
  servParam={o1}
  QoS={good}
INVARIANTS Invariant
PROPERTY Property
```

Figure 5: TLC Configuration File

The properties checked where as follows:

- All requests made by a requester or a broker to the resource discovery need to be processed.
- All requests made by a requester to a broker need to be processed.
- All requests made by a requester or a broker to a provider need to be processed.

On the other side, invariants were mainly used to ensure that the types of all variables are preserved throughout the system execution.

The weak fairness requirement is meant to give priority to the matching actions (*RD\_Match* and *Broker\_Match\_Requester*). Figure 15 shows the output of running TLC on module *CompleteSystemPart6*. The run was successful with no violation of invariants or properties.

## 6 Stepwise Refinement Validation of CDS Specifications

In this sections we introduce a framework which allows the specification of CDS at different levels of abstraction and the validation of the refinement

relationships which exist between the different specifications.

## 6.1 CDS Specification

The structure of a TLA+ specification of a CDS is shown in Figure 6. All TLA+ specifications shown in this paper have been well commented (in shaded gray) in order to make them as self-explanatory as possible. Moreover, TLA+ constructs used will not be detailed here. The reader is advised to see [3] for further details.

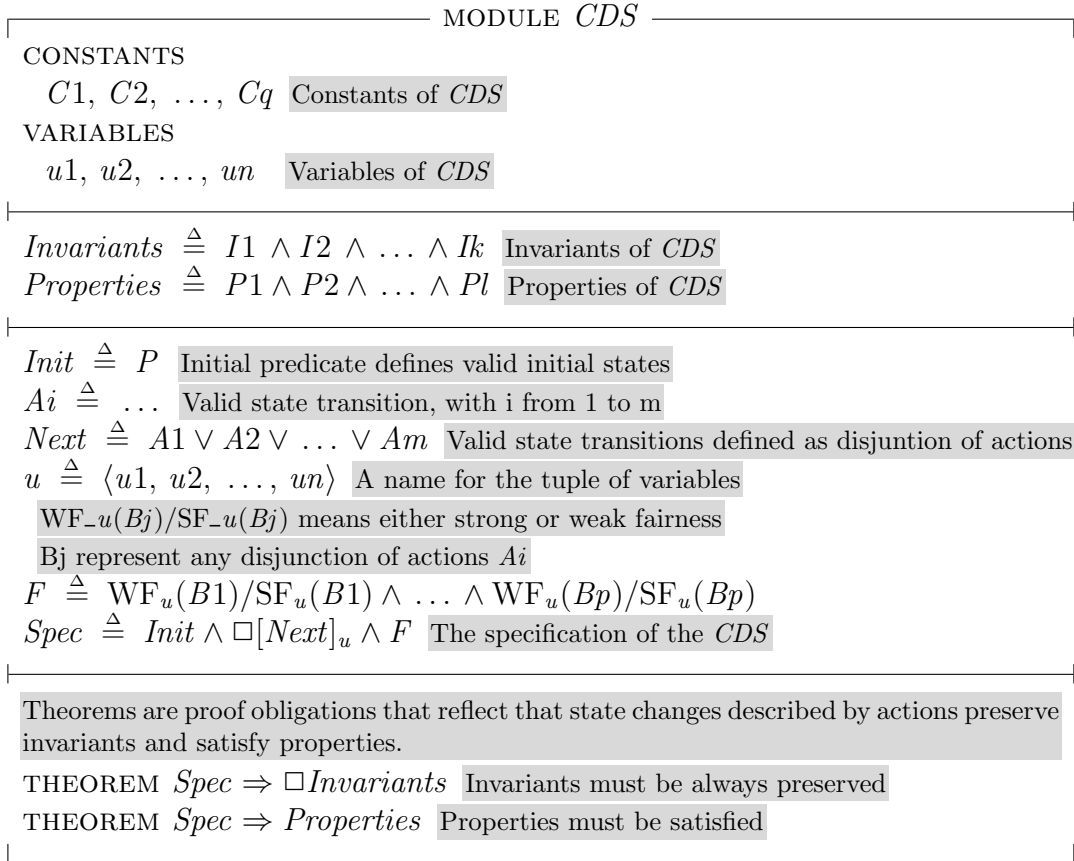


Figure 6: Structure of a TLA+ specification of a CDS

## 6.2 The Refinement Process

The main advantage of our approach is that the focus is first given to specifying the most abstract version of a given CDS such that “low-level” details

are avoided. In later versions of the specification, details can be “gradually” introduced.

A CDS  $Q$  is a refinement (or a lower-level version) of a CDS  $P$  if every allowed behavior in  $Q$  is allowed in  $P$  [2]. If  $Q$  is specified using a TLA formula  $\Psi$  and  $P$  is specified using TLA formula  $\Phi$ ,  $Q$  is a refinement of  $P$  if  $\Psi$  is a refinement of  $\Phi$ .

In order to formally define refinement, we need first to formally define the concept of “refinement mapping” [2]. If  $\Delta$  is a TLA specification, let  $C_\Delta$  be the set of constants of  $\Delta$  and  $V_\Delta$  is the set of variables of  $\Delta$ .

**Definition 6.1** (Refinement mapping). Let  $\Psi$  and  $\Phi$  be two specifications and let  $\rho : C_\Phi \cup V_\Phi \rightarrow C_\Psi \cup V_\Psi$ .  $\rho$  is a **refinement mapping** from  $\Psi$  to  $\Phi$  iff  $\rho$  is a total function and  $\Psi \Rightarrow \rho(\Phi)$ .  $\rho(\Phi)$  represents the substitution of constants and variables of  $\Phi$  by those of  $\Psi$ .

**Definition 6.2** (Refinement). Let  $\Psi$  and  $\Phi$  be two specifications.  $\Psi$  is a **refinement** of  $\Phi$  if there exists a refinement mapping from  $\Phi$  to  $\Psi$ .

As shown in Figure 7, we must explicitly relate states in the concrete specification with states in abstract specifications and this can be done in the form of substitutions (or *refinement mappings*) of flexible variables of the abstract specification with expressions of the concrete specification. Using TLC we can validate that a low-level specification is indeed a refinement of a more abstract one. The above refinement process has been successfully applied to the design patterns field [7]. This work has focused on proving a formal specification of the core interaction protocol of a CDS. This can be validly refined by augmenting it by handling privacy, trust and semantic integration as highlighted in the introduction. However showing the specifications of the possible refinements of the core interaction protocol of a CDS is beyond the scope of this paper.

## 7 Conclusion

This paper presented a new architecture for CDS. The main features of such an architecture is that it accommodates the pressing needs of openness, dynamism and heterogeneity through the incorporation of privacy, trust and semantic integration as integral parts (services) of the system. The behavior of the proposed system was described both informally and formally using TLA+. This has allowed it to be easily model-checked using TLC—the TLA+ model checker. To our knowledge, there is no recent work on formalizing the behavior of a CDS. Our specification tried to stay as much as

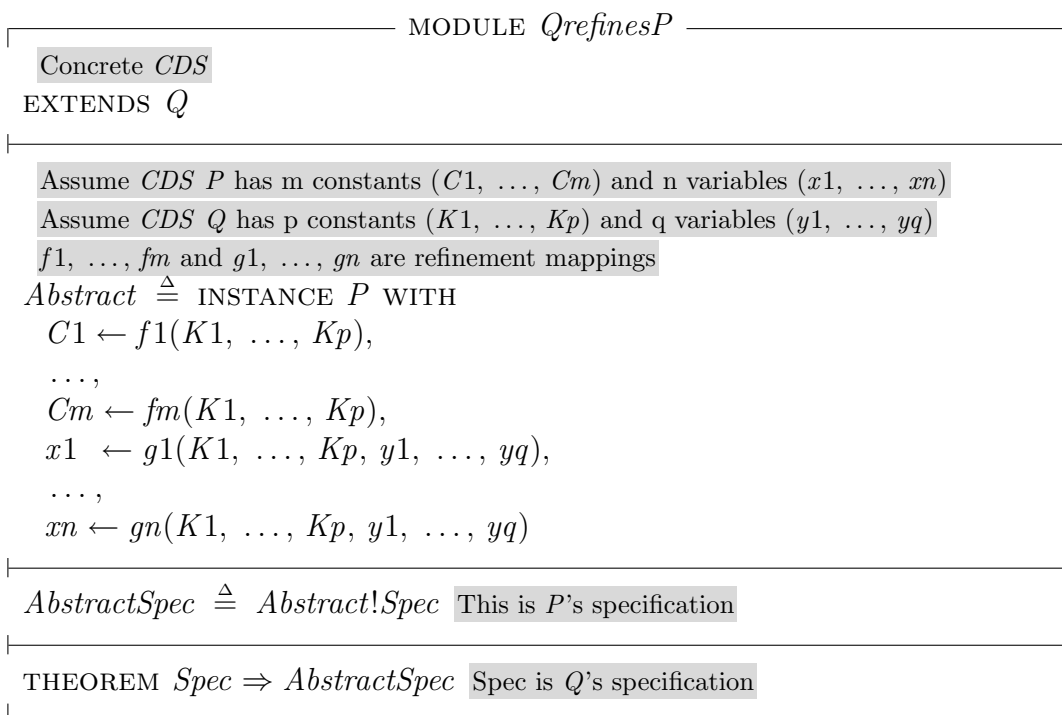


Figure 7: Structure of a TLA+ refinement of CDS

possible at the highest level of abstraction such that valid refinements can be easily made when either low-level details are added or the specifications of “vertical” protocols for trust, privacy and semantic integration are added to the current protocol.

Currently, we are working on devising protocols for handling privacy and trust in a completely decentralized manner. This will definitely add another dimension to the current and help tackle the issues and challenges faced by current and future CDS.

## References

- [1] J.E. Klobas and P.D. Jackson, editors. *Becoming Virtual: Knowledge Management and Transformation of the Distributed Organization*. Physica-Verlag, 2007.
- [2] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

- [3] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, MA, USA, 2002.
- [4] Natalya F. Noy. Semantic integration: a survey of ontology-based approaches. *SIGMOD Rec.*, 33(4):65–70, 2004.
- [5] Demazeau Y. Piolle, G. and J. Caelen. Privacy management in user-centred multi-agent systems. In *In Proceedings of the 7th Annual International Workshop "Engineering Societies in the Agents World" (ESAW 2006)*, pages 354–367, 2006.
- [6] S. D. Ramchurn, D. Hunyh, and N. R. Jennings. Trust in multi-agent systems. *Knowledge Engineering Review*, 19(1):1–25, 2004.
- [7] T. Taibi, Angel Herranz, and Juan Jose Moreno-Navarro. Stepwise refinement validation of design patterns formalized in tla+ using the tlc model checker. *Journal of Object Technology*, To Appear, 2009.
- [8] A.S. Tanenbaum and M. Van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, 2006.
- [9] Andrew Warfield, Yvonne Coady, and Norm Hutchinson. Identifying open problems in distributed systems. In *European Research Seminar on Advances in Distributed Systems (ERSADS)*, 2001.



Toufik Taibi received his PhD from Multimedia University, Malaysia in 2003. He is currently a post-doctoral fellow at the department of Electrical and Computer Engineering at the University of Western Ontario, Canada. Prior to that he was Assistant Professor at United Arab Emirates University, UAE. Dr. Taibi has more than 14 years of combined teaching, research and industry experience. His research interests include formal methods as it applied to software engineering, multi-agent systems and cooperative distributed systems.



## Appendix: TLA+ Specifications

MODULE <i>Identities</i>
Allocation and deallocation of identities for entities by the registration service.
EXTENDS <i>Naturals</i>
LOCAL INSTANCE <i>TLC</i>
Finite set of possible identities.
CONSTANT <i>IDs</i>
The set of identities that are in use.
VARIABLE <i>ids_in_use</i>
<hr/>
<i>ids_in_use</i> is always a subset of <i>IDs</i> .
<i>Invariant</i> $\triangleq$ $ids\_in\_use \subseteq IDs$
<hr/>
Initially <i>ids_in_use</i> is empty.
<i>Init</i> $\triangleq$ $ids\_in\_use = \{\}$
<hr/>
<i>New</i> $\triangleq$ $\exists x \in IDs :$
The <i>ID</i> to be created should not be in use.
$\wedge x \in (IDs \setminus ids\_in\_use)$
The set <i>ids_in_use</i> is augmented with the newly created <i>ID</i> .
$\wedge ids\_in\_use' = ids\_in\_use \cup \{x\}$
<i>Delete</i> ( <i>x</i> ) $\triangleq$
The <i>ID</i> to be deleted should be in the set <i>ids_in_use</i>
$\wedge x \in ids\_in\_use$
The deleted <i>ID</i> is removed from the set <i>ids_in_use</i> .
$\wedge ids\_in\_use' = ids\_in\_use \setminus \{x\}$
<hr/>
<i>Next</i> $\triangleq$
$\vee New$
$\vee \exists x \in IDs : Delete(x)$
<i>u</i> $\triangleq$ $\langle ids\_in\_use \rangle$
<hr/>
<i>Spec</i> $\triangleq$ $Init \wedge \square[Next]_u$

Figure 8: TLA+ Module *Identities*



Figure 9: TLA+ Module *CompleteSystemPart1*

MODULE <i>CompleteSystemPart2</i>	
EXTENDS <i>CompleteSystemPart1</i>	
These are sets with possible values of brokers <i>IDs</i> , requesters <i>IDs</i> and providers <i>IDs</i> .	
$Broker \triangleq$	$Brokers\_IDs \times brtrID \times (reqDescr \setminus brDescr) \times pref$
$Requester \triangleq$	$Requesters\_IDs \times rqtrID \times reqDescr \times pref$
$Provider \triangleq$	$Providers\_IDs \times capDescr \times servName \times servParam \times QoS$
These are sets with current values of brokers <i>IDs</i> , requesters <i>IDs</i> and Providers <i>IDs</i> . The set difference is used to remove “broker”.	
$cBroker \triangleq$	$brID \times brtrID \times (reqDescr \setminus brDescr) \times pref$
$cRequester \triangleq$	$rqID \times rqtrID \times reqDescr \times pref$
$cProvider \triangleq$	$provID \times capDescr \times servName \times servParam \times QoS$
These are definitions used in <i>Invariant</i> , <i>Property</i> and action <i>Init</i> .	
$Broker\_OR\_Requester \triangleq$	$(Brokers\_IDs \times brtrID) \cup (Requesters\_IDs \times rqtrID)$
$Call\_Format \triangleq$	$(Requesters\_IDs \times rqtrID \times servName \times servParam) \cup (Brokers\_IDs \times brtrID \times servName \times servParam)$
The property define 3 conditions that need to hold. 1-All requests put by a requester or broker needs to be handled by the resource discovery. 3-All requests made through a broker needs to be handled. 3-All requests (from a broker or requester) made to a provider need to be handled.	
$Property \triangleq$	$\wedge \forall \langle x, y, z, t \rangle \in Broker \cup Requester : in\_seq(\langle x, y, z, t \rangle, hrdQ) \Rightarrow done[\langle x, y \rangle] \neq ""$ $\wedge \forall \langle x, y, z, t \rangle \in Requester : in\_func\_seq(\langle x, y, z, t \rangle, hbQ) \Rightarrow done[\langle x, y \rangle] \neq ""$ $\wedge \forall \langle x, y, z, t \rangle \in Call\_Format : in\_func\_seq(\langle x, y, z, t \rangle, hpQ) \Rightarrow result[\langle x, y \rangle] \neq ""$
The invariant defines basically the type definition of each variable in the specification	
$Invariant \triangleq$	$\wedge BrokersIDs!Invariant \wedge RequestersIDs!Invariant \wedge ProvidersIDs!Invariant$ $\wedge capDB \subseteq Provider \wedge brDB \subseteq (Brokers\_IDs \times brDescr)$ $\wedge rdQ \in Seq(Broker \cup Requester) \wedge hrdQ \in Seq(Broker \cup Requester)$ $\wedge bQ \in [Brokers\_IDs \rightarrow Seq(Requester)] \wedge tempbQ \in [Brokers\_IDs \rightarrow Seq(Requester)]$ $\wedge hbQ \in [Brokers\_IDs \rightarrow Seq(Requester)]$ $\wedge matchDB \in [Broker\_OR\_Requester \rightarrow Seq(Provider \cup (Brokers\_IDs \times brDescr))]$ $\wedge done \in [Broker\_OR\_Requester \rightarrow STRING]$ $\wedge mapID \in [Brokers\_IDs \times brtrID \rightarrow Seq(Requesters\_IDs \times rqtrID)]$ $\wedge pQ \in [Providers\_IDs \rightarrow Seq(Call\_Format)] \wedge hpQ \in [Providers\_IDs \rightarrow Seq(Call\_Format)]$ $\wedge result \in [Broker\_OR\_Requester \rightarrow STRING]$ $\wedge brL \in [Requesters\_IDs \rightarrow SUBSET Brokers\_IDs]$ $\wedge prL \in [Brokers\_IDs \cup Requesters\_IDs \rightarrow SUBSET Provider]$
Here are variables are initialized at the initial state. The first 3 initializations are calls to the predicate <i>Init</i> of the module identities	
$Init \triangleq$	$\wedge BrokersIDs!Init \wedge RequestersIDs!Init \wedge ProvidersIDs!Init$ $\wedge capDB = \{\} \wedge brDB = \{\}$ $\wedge rdQ = \langle \rangle \wedge hrdQ = \langle \rangle$ $\wedge bQ = [x \in Brokers\_IDs \mapsto \langle \rangle] \wedge tempbQ = [x \in Brokers\_IDs \mapsto \langle \rangle] \wedge hbQ = [x \in Brokers\_IDs \mapsto \langle \rangle]$ $\wedge matchDB = [X \in Broker\_OR\_Requester \mapsto \langle \rangle]$ $\wedge done = [X \in Broker\_OR\_Requester \mapsto ""]$ $\wedge mapID = [X \in Brokers\_IDs \times brtrID \mapsto \langle \rangle]$ $\wedge pQ = [x \in Providers\_IDs \mapsto \langle \rangle] \wedge hpQ = [x \in Providers\_IDs \mapsto \langle \rangle]$ $\wedge result = [X \in Broker\_OR\_Requester \mapsto ""]$ $\wedge brL = [x \in Requesters\_IDs \mapsto \{\}] \wedge prL = [X \in Brokers\_IDs \cup Requesters\_IDs \mapsto \{\}]$
Action <i>New</i> of module identities is called to create a broker <i>ID</i> and a requester <i>ID</i> .	
$Create\_BrokerID \triangleq$	$\wedge BrokersIDs!New$ $\wedge UNCHANGED \langle rqID, provID, capDB, brDB, rdQ, hrdQ, bQ, tempbQ, hbQ, matchDB, done, mapID, pQ, hpQ, result, brL, prL \rangle$
$Create\_RequesterID \triangleq$	$\wedge RequestersIDs!New$ $\wedge UNCHANGED \langle brID, provID, capDB, brDB, rdQ, hrdQ, bQ, tempbQ, hbQ, matchDB, done, mapID, pQ, hpQ, result, brL, prL \rangle$

Figure 10: TLA+ Module *CompleteSystemPart2*



Figure 11: TLA+ Module *CompleteSystemPart3*

MODULE <i>CompleteSystemPart4</i>
EXTENDS <i>CompleteSystemPart3</i>
<p>                     Action for a requester to make a request to <i>RD</i>  <i>Requester_Request_RD</i> <math>\triangleq \exists \langle r, rt, rd, pr \rangle \in cRequester :</math>  <math>\wedge \neg \text{elt\_in\_seq}(rt, \text{hrd}Q)</math> “<i>rt</i>” was not previously used by a <i>Requester_Request_RD</i> action  <math>\wedge \neg \text{elt\_in\_func\_seq}(rt, \text{hb}Q)</math> “<i>rt</i>” was not previously used by a <i>Requester_Request_Broker</i> action  <math>\wedge \neg \text{elt\_in\_func\_seq}(rt, \text{hp}Q)</math> “<i>rt</i>” was not previously used by a <i>Requester_Request_Provider</i> action  <math>\wedge \text{cap}DB \neq \{\}</math> <i>capDB</i> not empty  <math>\wedge \text{rd}Q' = \text{Append}(\text{rd}Q, \langle r, rt, rd, pr \rangle)</math> Requester tuple added to <i>rdQ</i>  <math>\wedge \text{hrd}Q' = \text{Append}(\text{hrd}Q, \langle r, rt, rd, pr \rangle)</math> Requester tuple added to <i>hrdQ</i>  <math>\wedge \text{UNCHANGED} \langle \text{br}ID, \text{rq}ID, \text{prov}ID, \text{cap}DB, \text{br}DB, bQ, \text{temp}bQ, \text{hb}Q, \text{match}DB, \text{done}, \text{map}ID, pQ, \text{hp}Q, \text{result}, \text{br}L, \text{pr}L \rangle</math> </p> <p>                     Action for a requester to make a request to the broker  <i>Requester_Request_Broker</i> <math>\triangleq \exists rt1 \in \text{rqtr}ID, \langle r, rt2, rd, pr \rangle \in cRequester, b \in \text{br}ID :</math>  <math>\wedge \neg \text{elt\_in\_seq}(rt2, \text{hrd}Q)</math> “<i>rt</i>” was not previously used by a <i>Requester_Request_RD</i> action  <math>\wedge \neg \text{elt\_in\_func\_seq}(rt2, \text{hb}Q)</math> “<i>rt</i>” was not previously used by a <i>Requester_Request_Broker</i> action  <math>\wedge \neg \text{elt\_in\_func\_seq}(rt2, \text{hp}Q)</math> “<i>rt</i>” was not used by a <i>Requester_Request_Provider</i> action  <math>\wedge \text{cap}DB \neq \{\}</math> <i>capDB</i> is not empty  <math>\wedge b \in \text{br}L[r]</math> Broker <i>b</i> is in <i>brL[r]</i>  <math>\wedge \text{rd} \notin \text{br}Descr</math> <i>brDescr</i> = {<i>Broker</i>} can only be used when looking for a broker through <i>Requester_Request_RD</i>  <math>\wedge bQ' = [bQ \text{ EXCEPT } ![b] = \text{Append}(@, \langle r, rt2, rd, pr \rangle)]</math> Requester tuple added to <i>bQ[b]</i>  <math>\wedge \text{temp}bQ' = [\text{temp}bQ \text{ EXCEPT } ![b] = \text{Append}(@, \langle r, rt2, rd, pr \rangle)]</math> Requester tuple added to <i>tempbQ[b]</i>  <math>\wedge \text{hb}Q' = [\text{hb}Q \text{ EXCEPT } ![b] = \text{Append}(@, \langle r, rt2, rd, pr \rangle)]</math> Requester tuple added to <i>hbQ[b]</i>  <math>\wedge \text{UNCHANGED} \langle \text{br}ID, \text{rq}ID, \text{prov}ID, \text{cap}DB, \text{br}DB, \text{rd}Q, \text{hrd}Q, \text{match}DB, \text{done}, \text{map}ID, pQ, \text{hp}Q, \text{result}, \text{br}L, \text{pr}L \rangle</math> </p> <p>                     Action for a broker to make a request to <i>RD</i>  <i>Broker_Request_RD</i> <math>\triangleq \exists \langle b, bt, rd, pr \rangle \in cBroker, \langle r, rt \rangle \in \text{rq}ID \times \text{rqtr}ID :</math>  <math>\wedge \neg \text{elt\_in\_seq}(bt, \text{hrd}Q)</math> “<i>bt</i>” was not previously used by a <i>Broker_Request_Rd</i> action  <math>\wedge \neg \text{elt\_in\_func\_seq}(bt, \text{hp}Q)</math> “<i>bt</i>” was not previously used by a <i>Broker_Request_Provider</i> action  <math>\wedge \text{cap}DB \neq \{\}</math> <i>capDB</i> is not empty  <math>\wedge \text{temp}bQ[b] \neq \langle \rangle</math> queue <i>tempQ[b]</i> is not empty  <math>\wedge \text{Head}(\text{temp}bQ[b]) = \langle r, rt, rd, pr \rangle</math> the request should be the head of <i>tempbQ</i>                      Broker tuple <math>\langle b, bt \rangle</math> is mapped to its original requester tuple <math>\langle r, rt \rangle</math>  <math>\wedge \text{map}ID' = [\text{map}ID \text{ EXCEPT } ![b, bt] = \text{Append}(@, \langle r, rt \rangle)]</math>  <math>\wedge \text{temp}bQ' = [\text{temp}bQ \text{ EXCEPT } ![b] = \text{Tail}(@)]</math> Requester tuple is removed from <i>tempQ[b]</i>  <math>\wedge \text{rd}Q' = \text{Append}(\text{rd}Q, \langle b, bt, rd, pr \rangle)</math> Broker tuple is added to <i>rdQ</i>  <math>\wedge \text{hrd}Q' = \text{Append}(\text{hrd}Q, \langle b, bt, rd, pr \rangle)</math> Broker tuple is added to <i>hrdQ</i>  <math>\wedge \text{UNCHANGED} \langle \text{br}ID, \text{rq}ID, \text{prov}ID, \text{cap}DB, \text{br}DB, bQ, \text{hb}Q, \text{match}DB, \text{done}, pQ, \text{hp}Q, \text{result}, \text{br}L, \text{pr}L \rangle</math> </p> <p>                     This action handles the case where the <i>RD</i> is matching a request (from requester or broker) against the providers in <i>capDB</i>. In its simplest form the match is based on the fact that <i>reqDescr</i> (of a broker or requester) is equal to <i>capDescr</i> of the provider                 </p> <p> <i>RD_Match</i> <math>\triangleq \exists \langle b\_r, t, rd, pr \rangle \in cBroker \cup cRequester :</math>  <math>\wedge \text{rd}Q \neq \langle \rangle</math> <i>rdQ</i> is not empty  <math>\wedge \text{cap}DB \neq \{\}</math> <i>capDB</i> is not empty  <math>\wedge \text{Head}(\text{rd}Q) = \langle b\_r, t, rd, pr \rangle</math> tuple <math>\langle b\_r, t, rd, pr \rangle</math> is the head of <i>rdQ</i>  <math>\wedge \text{rd}Q' = \text{Tail}(\text{rd}Q)</math> <math>\langle b\_r, t, rd, pr \rangle</math> removed from <i>rdQ</i> </p> <p>                     In its simplest form, a match happens when <i>X[2]</i> (<i>capDescr</i> of a provider) matched <i>rd</i> (<i>reqDescr</i> of the request)  <math>\wedge \forall X \in \text{cap}DB : \wedge \text{match}DB' = [\text{match}DB \text{ EXCEPT } ![b\_r, t] = \text{IF } X[2] = \text{rd} \wedge \neg \text{in\_seq}(X, @) \text{ THEN } \text{Append}(@, X) \text{ ELSE } @]</math>  <math>\wedge \text{pr}L' = [\text{pr}L \text{ EXCEPT } ![b\_r] = \text{IF } X[2] = \text{rd} \wedge \text{rd} \notin \text{br}Descr \text{ THEN } @ \cup \{X\} \text{ ELSE } @]</math>  <math>\wedge \text{IF } b\_r \in \text{rq}ID \wedge \text{br}DB \neq \{\} \text{ THEN } \forall X \in \text{br}DB : \text{br}L' = [\text{br}L \text{ EXCEPT } ![b\_r] = \text{IF } X[2] = \text{rd} \text{ THEN } @ \cup \{X[1]\} \text{ ELSE } @]</math>                      ELSE UNCHANGED <i>brL</i>  <math>\wedge \text{done}' = [\text{done} \text{ EXCEPT } ![b\_r, t] = \text{“YES”}]</math> This reflects that the request has been handled  <math>\wedge \text{UNCHANGED} \langle \text{br}ID, \text{rq}ID, \text{prov}ID, \text{cap}DB, \text{br}DB, \text{hrd}Q, bQ, \text{temp}bQ, \text{hb}Q, \text{map}ID, pQ, \text{hp}Q, \text{result} \rangle</math> </p>

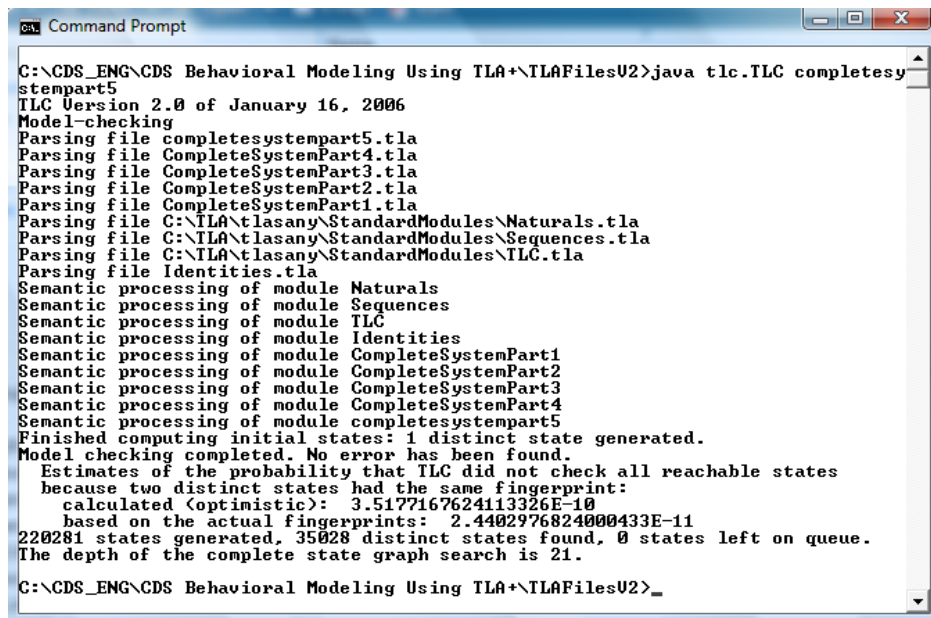
 Figure 12: TLA+ Module *CompleteSystemPart4*



Figure 13: TLA+ Module *CompleteSystemPart5*

MODULE <i>CompleteSystemPart6</i>
EXTENDS <i>CompleteSystemPart5</i>
<p>This action handles the case where a provider replies to a broker's request</p> <p><i>Provider_Reply_Broker</i> <math>\triangleq \exists \langle b, bt \rangle \in brID \times brtrID, \langle p, cd, sn, sp, qs \rangle \in cProvider :</math></p> <p><math>\wedge pQ[p] \neq \langle \rangle</math> <i>pQ[p]</i> is not empty</p> <p><math>\wedge Head(pQ[p]) = \langle b, bt, sn, sp \rangle</math> tuple <math>\langle b, bt, sn, sp \rangle</math> is the head of queue <i>pQ[p]</i></p> <p><math>\wedge pQ' = [pQ \text{ EXCEPT } ![p] = Tail(@)]</math> tuple <math>\langle b, bt, sn, sp \rangle</math> is removed from queue <i>pQ[p]</i></p> <p><math>\wedge result' = [result \text{ EXCEPT } ![(b, bt)] = \text{"YES"}]</math> This reflects the result of a service call</p> <p><math>\wedge \text{UNCHANGED } \langle brID, rqID, provID, capDB, brDB, rdQ, hrdQ, bQ, tempbQ, hbQ, matchDB, done, mapID, hpQ, brL, prL \rangle</math></p> <p>This action handles the case where a broker provides the final reply to a requester's request</p> <p><i>Broker_Reply_Requester</i> <math>\triangleq \exists \langle b, bt \rangle \in brID \times brtrID, \langle r, rt \rangle \in rqID \times rqrtrID :</math></p> <p><math>\wedge result[\langle b, bt \rangle] = \text{"YES"}</math></p> <p><math>\wedge mapID[\langle b, bt \rangle][1] = \langle r, rt \rangle</math></p> <p><math>\wedge result' = [result \text{ EXCEPT } ![(r, rt)] = \text{"YES"}]</math> This reflects the result of a service call</p> <p><math>\wedge \text{UNCHANGED } \langle brID, rqID, provID, capDB, brDB, rdQ, hrdQ, bQ, tempbQ, hbQ, matchDB, done, mapID, pQ, hpQ, brL, prL \rangle</math></p> <p>This action handles the case where a provider replies to a requester's request after an <i>RD_Match</i> action is executed.</p> <p><i>Provider_Reply_Requester</i> <math>\triangleq \exists \langle r, rt \rangle \in rqID \times rqrtrID, \langle p, cd, sn, sp, qs \rangle \in cProvider :</math></p> <p><math>\wedge pQ[p] \neq \langle \rangle</math> <i>pQ[p]</i> is not empty</p> <p><math>\wedge Head(pQ[p]) = \langle r, rt, sn, sp \rangle</math> tuple <math>\langle r, rt, sn, sp \rangle</math> is the head of queue <i>pQ[p]</i></p> <p><math>\wedge pQ' = [pQ \text{ EXCEPT } ![p] = Tail(@)]</math> Tuple <math>\langle r, rt, sn, sp \rangle</math> is removed from the queue</p> <p><math>\wedge result' = [result \text{ EXCEPT } ![(r, rt)] = \text{"YES"}]</math> This reflects the result of a service call</p> <p><math>\wedge \text{UNCHANGED } \langle brID, rqID, provID, capDB, brDB, rdQ, hrdQ, bQ, tempbQ, hbQ, matchDB, done, mapID, hpQ, brL, prL \rangle</math></p> <p>Next action is a disjunction of all actions</p> <p><i>Next</i> <math>\triangleq</math></p> <ul style="list-style-type: none"> <li><math>\vee \text{Create\_BrokerID}</math></li> <li><math>\vee \text{Create\_RequesterID}</math></li> <li><math>\vee \text{Create\_ProviderID}</math></li> <li><math>\vee \exists b \in brID : \text{Remove\_BrokerID}(b)</math></li> <li><math>\vee \exists r \in rqID : \text{Remove\_RequesterID}(r)</math></li> <li><math>\vee \exists p \in provID : \text{Remove\_ProviderID}(p)</math></li> <li><math>\vee \text{Provider\_Advertise}</math></li> <li><math>\vee \text{Provider\_Unadvertise}</math></li> <li><math>\vee \text{Broker\_Advertise}</math></li> <li><math>\vee \text{Broker\_Unadvertise}</math></li> <li><math>\vee \text{Requester\_Request\_RD}</math></li> <li><math>\vee \text{Requester\_Request\_Broker}</math></li> <li><math>\vee \text{Broker\_Request\_RD}</math></li> <li><math>\vee \text{RD\_Match}</math></li> <li><math>\vee \text{Broker\_Match\_Requester}</math></li> <li><math>\vee \text{Requester\_Request\_Provider}</math></li> <li><math>\vee \text{Broker\_Request\_Provider}</math></li> <li><math>\vee \text{Provider\_Reply\_Requester}</math></li> <li><math>\vee \text{Provider\_Reply\_Broker}</math></li> <li><math>\vee \text{Broker\_Reply\_Requester}</math></li> </ul> <p>tuple of all variables, used to allow stuttering</p> <p><math>u \triangleq \langle brID, rqID, provID, capDB, brDB, rdQ, hrdQ, bQ, tempbQ, hbQ, matchDB, done, mapID, pQ, hpQ, result, brL, prL \rangle</math></p> <p><math>WF_u(\text{RD\_Match} \vee \text{Broker\_Match\_Requester})</math> allows these two actions to be executed infinitely often</p> <p><i>Spec</i> <math>\triangleq \text{Init} \wedge \square [Next]_u \wedge WF_u(\text{RD\_Match} \vee \text{Broker\_Match\_Requester})</math></p>
Invariants and Properties are fulfilled
THEOREM <i>Spec</i> $\Rightarrow \square \text{Invariant}$
THEOREM <i>Spec</i> $\Rightarrow \text{Property}$

Figure 14: TLA+ Module *CompleteSystemPart6*



```
C:\CDS_ENG\CDS Behavioral Modeling Using TLA+\TLAFilesU2>java tlc.TLC completesystempart5
TLC Version 2.0 of January 16, 2006
Model-checking
Parsing file completesystempart5.tla
Parsing file CompleteSystemPart4.tla
Parsing file CompleteSystemPart3.tla
Parsing file CompleteSystemPart2.tla
Parsing file CompleteSystemPart1.tla
Parsing file C:\ILA\tlasany\StandardModules\Naturals.tla
Parsing file C:\ILA\tlasany\StandardModules\Sequences.tla
Parsing file C:\ILA\tlasany\StandardModules\TLC.tla
Parsing file Identities.tla
Semantic processing of module Naturals
Semantic processing of module Sequences
Semantic processing of module TLC
Semantic processing of module Identities
Semantic processing of module CompleteSystemPart1
Semantic processing of module CompleteSystemPart2
Semantic processing of module CompleteSystemPart3
Semantic processing of module CompleteSystemPart4
Semantic processing of module completesystempart5
Finished computing initial states: 1 distinct state generated.
Model checking completed. No error has been found.
Estimates of the probability that TLC did not check all reachable states
because two distinct states had the same fingerprint:
  calculated (optimistic): 3.5177167624113326E-10
  based on the actual fingerprints: 2.4402976824000433E-11
220281 states generated, 35028 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 21.
C:\CDS_ENG\CDS Behavioral Modeling Using TLA+\TLAFilesU2>_
```

Figure 15: Output of Running TLC on module *CompleteSystemPart6*