Formal Verification of Requirements using SPIN: A case Study on Web Services

Marco Roveri

roveri@irst.itc.it

http://sra.itc.it/people/roveri

joint work with R. Kazhamiakin and M. Pistore

ITC-irst – Automated Reasoning System Division

Via Sommarive 18, 38050 Trento

Italy

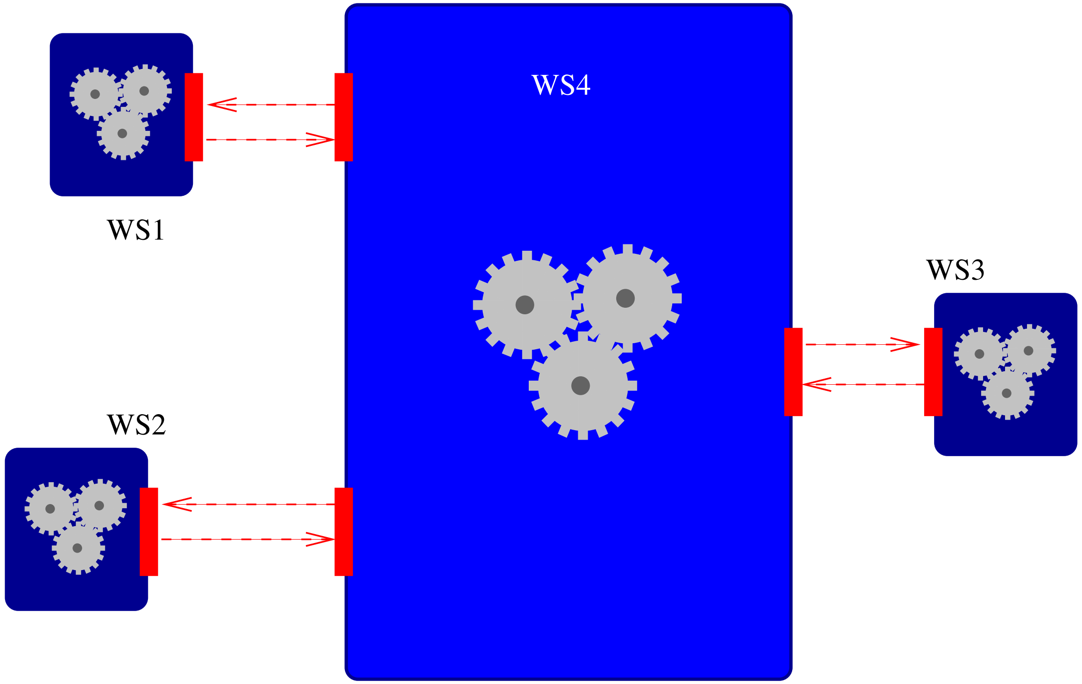
Introduction

- Integration of distributed business process is an emerging problem. . .
  - participants from different organizations
  - heterogeneity among services
  - autonomous evolution of processes
- Web Services (WS) offer the technology for business process integration:
  - languages for WS interoperability (SOAP, WSDL, UDDI,. . .)
    - In particular BPEL4WS (Business Process Execution Language)
  - tools for the design and the execution of WS
- Nevertheless, there is a need for advanced techniques for supporting the most complex aspects of business process integration:
  - simulation and (formal) verification
  - monitoring and diagnosis
  - (automated) support for composition and evolution

Outline

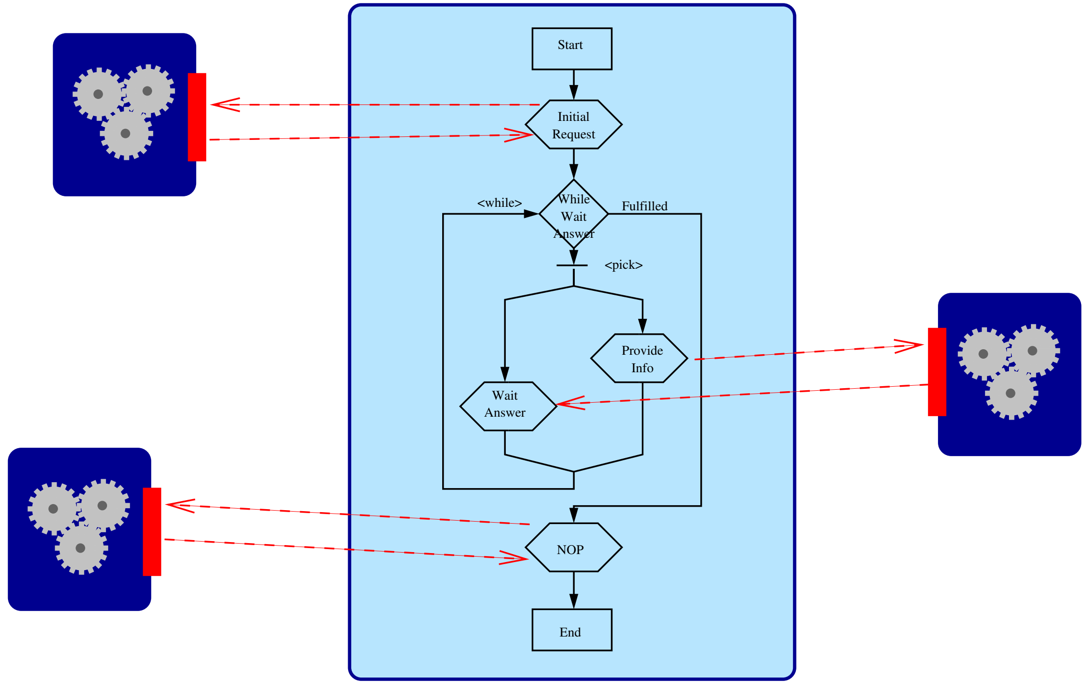
- Introduction to WS and related problems
- The need for business requirements
- A methodology for defining business requirements and for deriving executable code
- Verification of Business Requirements/Processes
- The tool supporting the methodology using SPIN
- Some experimental results
- Conclusions and Future works

Web Services



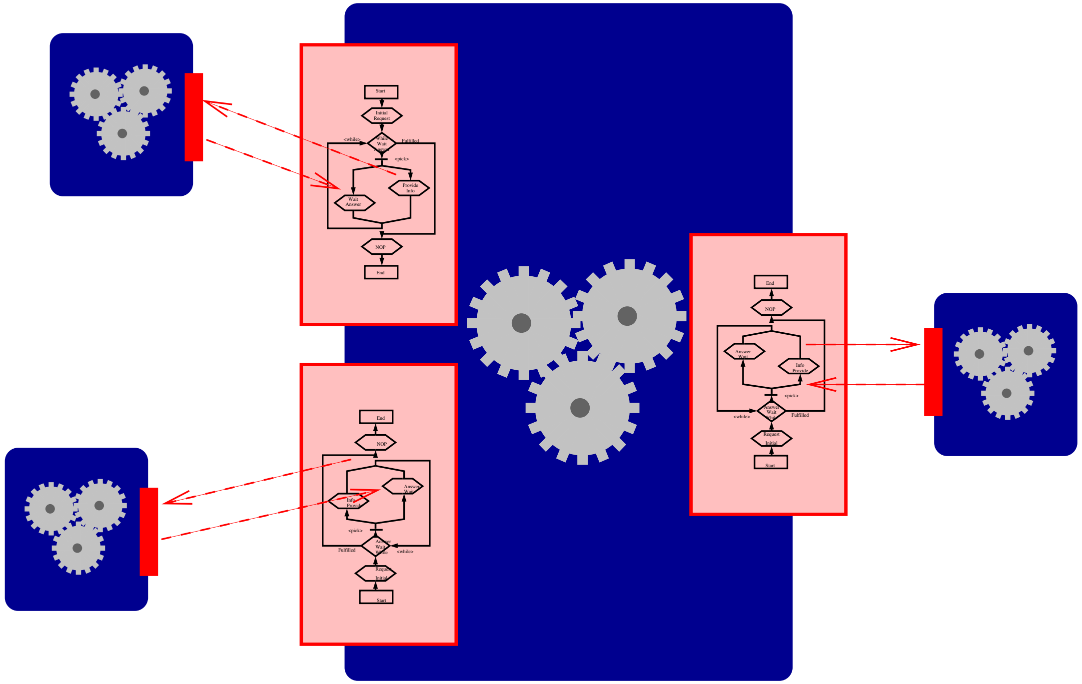
- Several web services participate to a business interaction.

WS: Executable Processes...



- WS languages (BPEL4WS) offers a set of core concepts for process description that can be used for:
  - the definition and the execution of the internal business process of a participant to a business interaction.

WS: ... and Interaction Protocols



- WS languages (e.g. BPEL4WS) offers a set of core concepts for process description that can be used for:
  - the definition and the execution of the internal business process of a participant to a business interaction.
  - the description and publication of the external business protocol that define the interaction behavior of a participant.

Verification for BPEL4WS

BPEL4WS allows for several forms of basic verification tasks:

Verification for BPEL4WS

BPEL4WS allows for several forms of basic verification tasks:

- At **design time**:
  - Is the internal BPEL4WS process specification **consistent** with the published protocol interfaces?
  - Given two or more BPEL4WS interfaces aiming to communicate, do they define a correct (e.g., **deadlock free**) protocol?

Verification for BPEL4WS

BPEL4WS allows for several forms of basic verification tasks:

- At **design time**:
  - Is the internal BPEL4WS process specification **consistent** with the published protocol interfaces?
  - Given two or more BPEL4WS interfaces aiming to communicate, do they define a correct (e.g., **deadlock free**) protocol?
- At **execution time**:
  - Do the other participants respect the protocol interface that they have published?

Verification for BPEL4WS

BPEL4WS allows for several forms of basic verification tasks:

- At **design time**:
  - Is the internal BPEL4WS process specification **consistent** with the published protocol interfaces?
  - Given two or more BPEL4WS interfaces aiming to communicate, do they define a correct (e.g., **deadlock free**) protocol?
- At **execution time**:
  - Do the other participants respect the protocol interface that they have published?

In order to do advanced verification based on specific properties on the behavior, a requirements language is needed.

Tropos: A Language for Business Requirements

Tropos: A Language for Business Requirements

- Tropos is **requirements-driven**:
  - focus on early phases of requirements analysis, aiming to the understanding of the operational environment of the software system

Tropos: A Language for Business Requirements

- Tropos is **requirements-driven**:
  - focus on early phases of requirements analysis, aiming to the understanding of the operational environment of the software system
- Tropos is **agent-oriented**:
  - agents and related notions, such as goals and plans, are used in all phases of software development

Tropos: A Language for Business Requirements

- Tropos is **requirements-driven**:
  - focus on early phases of requirements analysis, aiming to the understanding of the operational environment of the software system
- Tropos is **agent-oriented**:
  - agents and related notions, such as goals and plans, are used in all phases of software development
- Tropos has been applied in several case studies on **information systems** and **agent-based software systems**

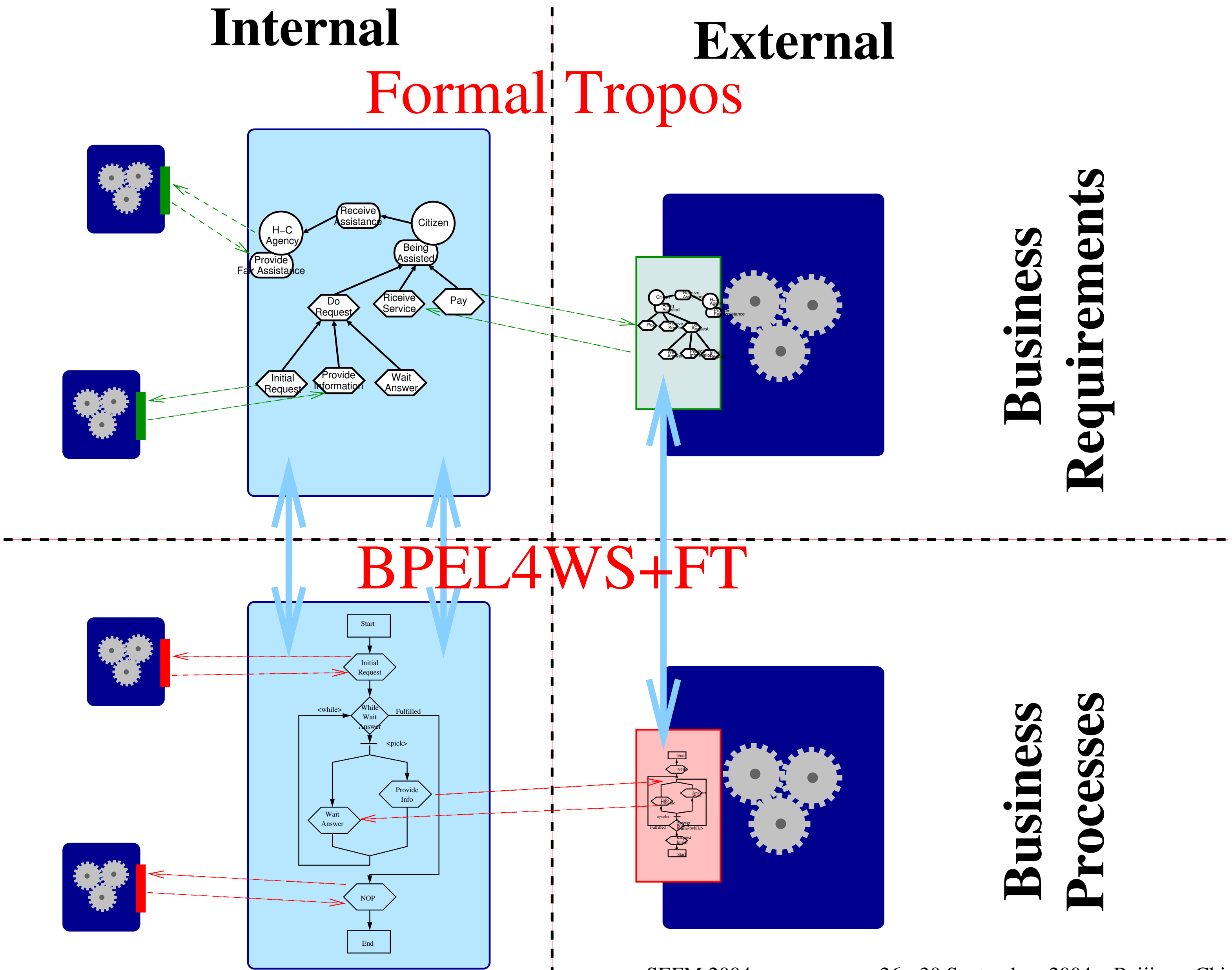
Tropos: A Language for Business Requirements

- Tropos is **requirements-driven**:
  - focus on early phases of requirements analysis, aiming to the understanding of the operational environment of the software system
- Tropos is **agent-oriented**:
  - agents and related notions, such as goals and plans, are used in all phases of software development
- Tropos has been applied in several case studies on **information systems** and **agent-based software systems**
- Tropos offers a set of **graphical notations** and of analysis techniques to support the designer in the development of the software system

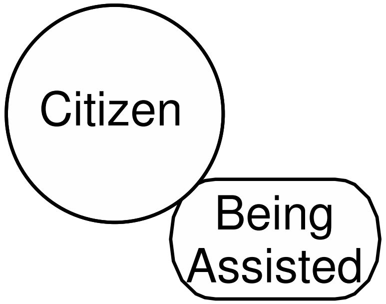
Tropos: A Language for Business Requirements

- Tropos is **requirements-driven**:
  - focus on early phases of requirements analysis, aiming to the understanding of the operational environment of the software system
- Tropos is **agent-oriented**:
  - agents and related notions, such as goals and plans, are used in all phases of software development
- Tropos has been applied in several case studies on **information systems** and **agent-based software systems**
- Tropos offers a set of **graphical notations** and of analysis techniques to support the designer in the development of the software system
- Formal Tropos extends Tropos with a **formal specification** language and with **verification** based on Model Checking

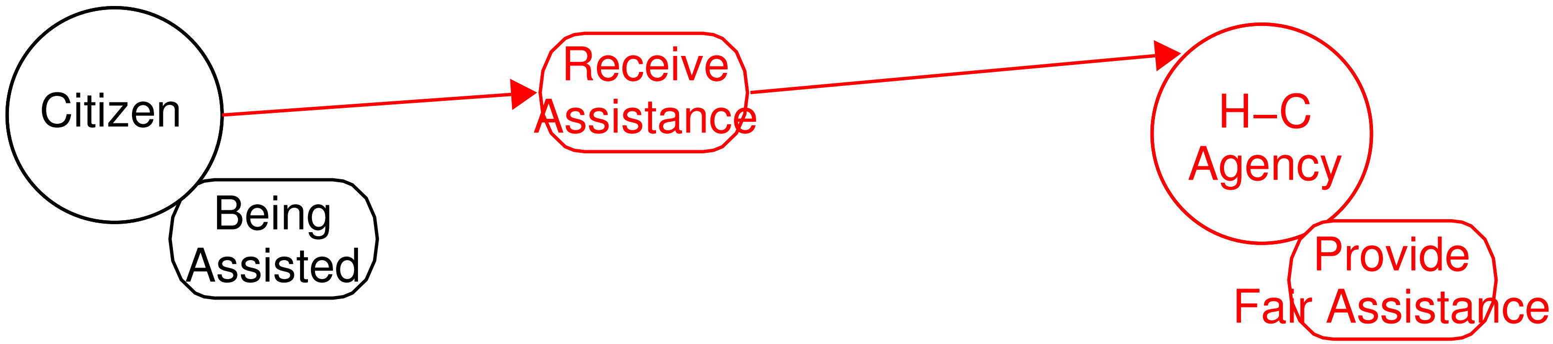
Proposed methodology: Tropos4WS



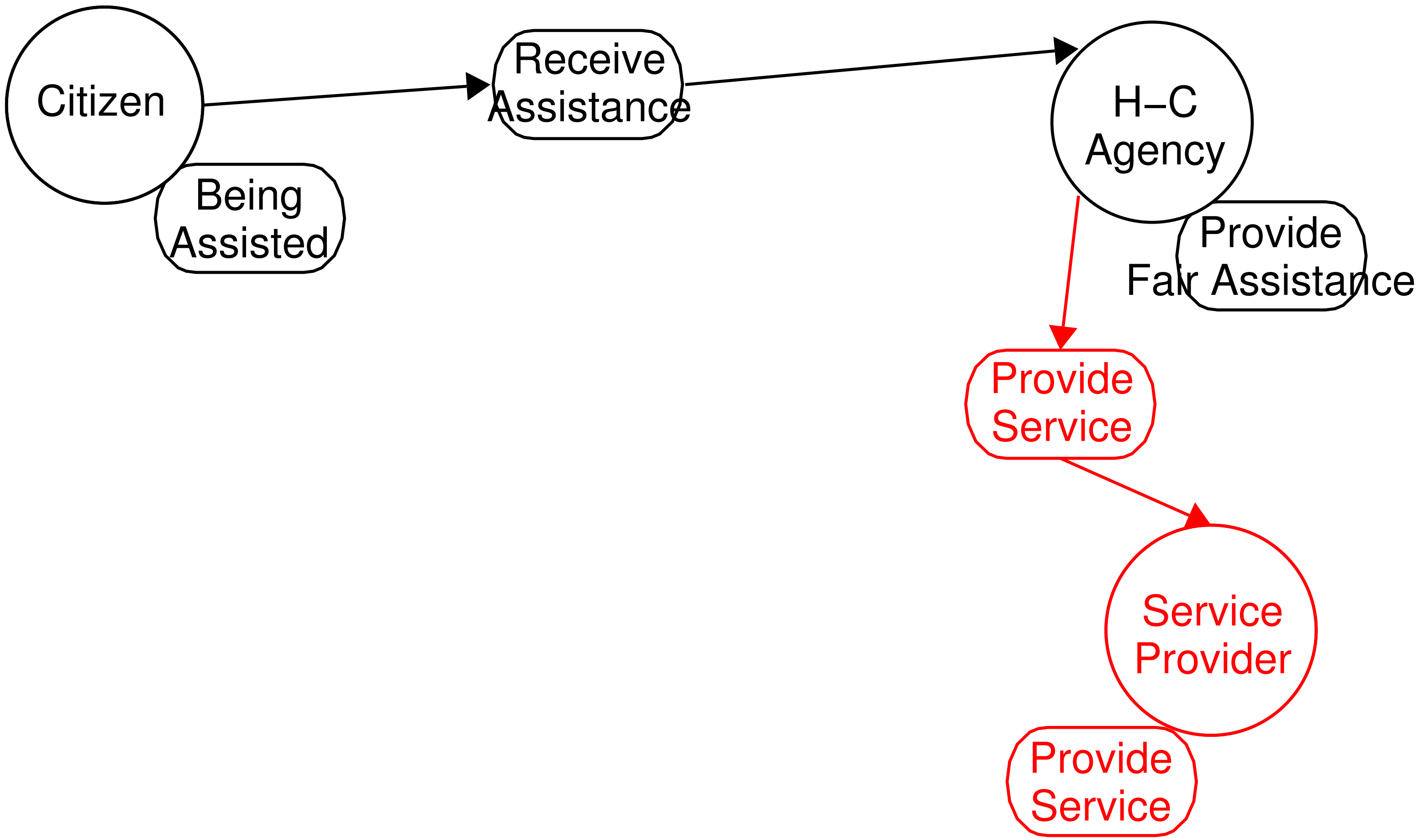
Specifying Business Requirements: Case Study



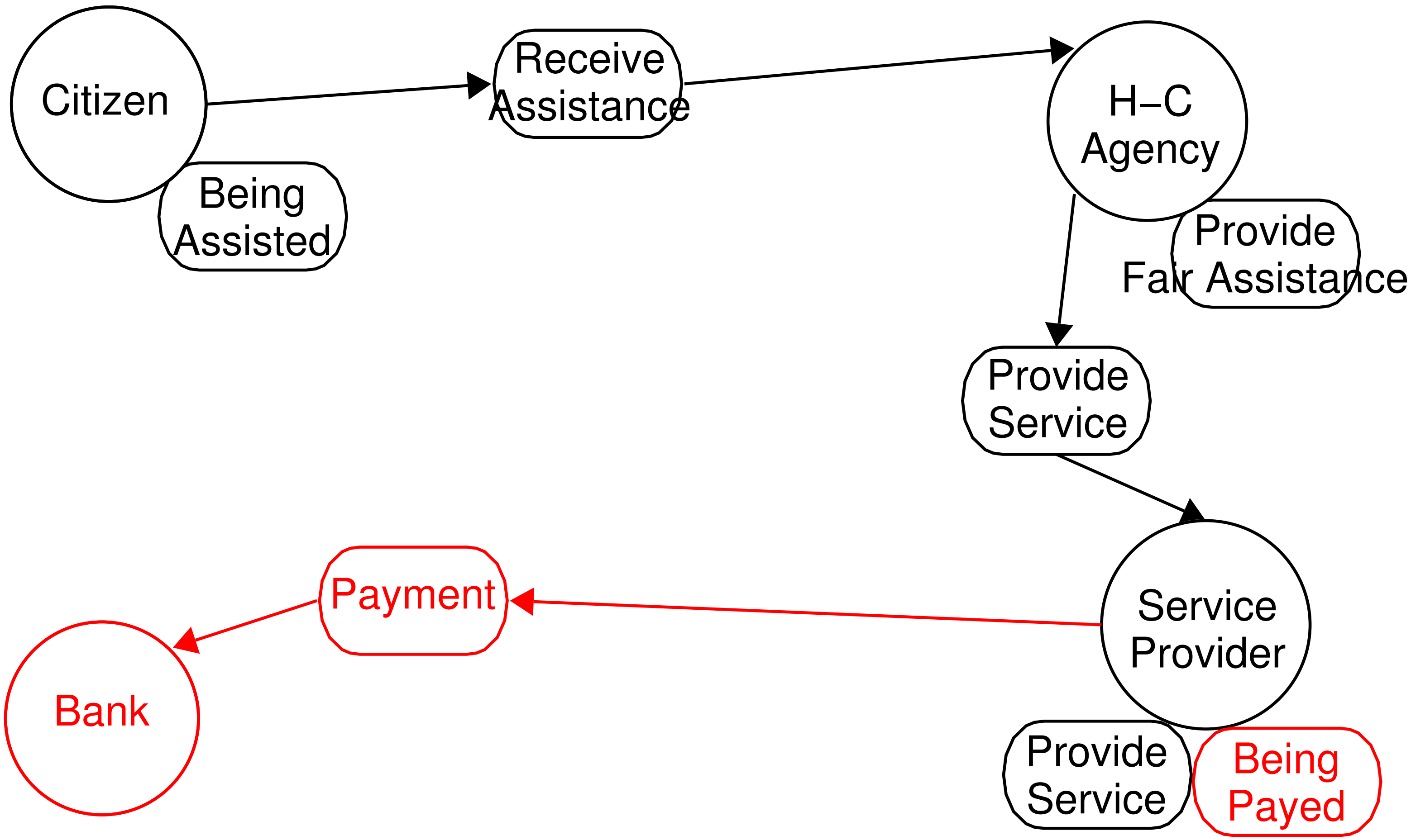
Specifying Business Requirements: Case Study



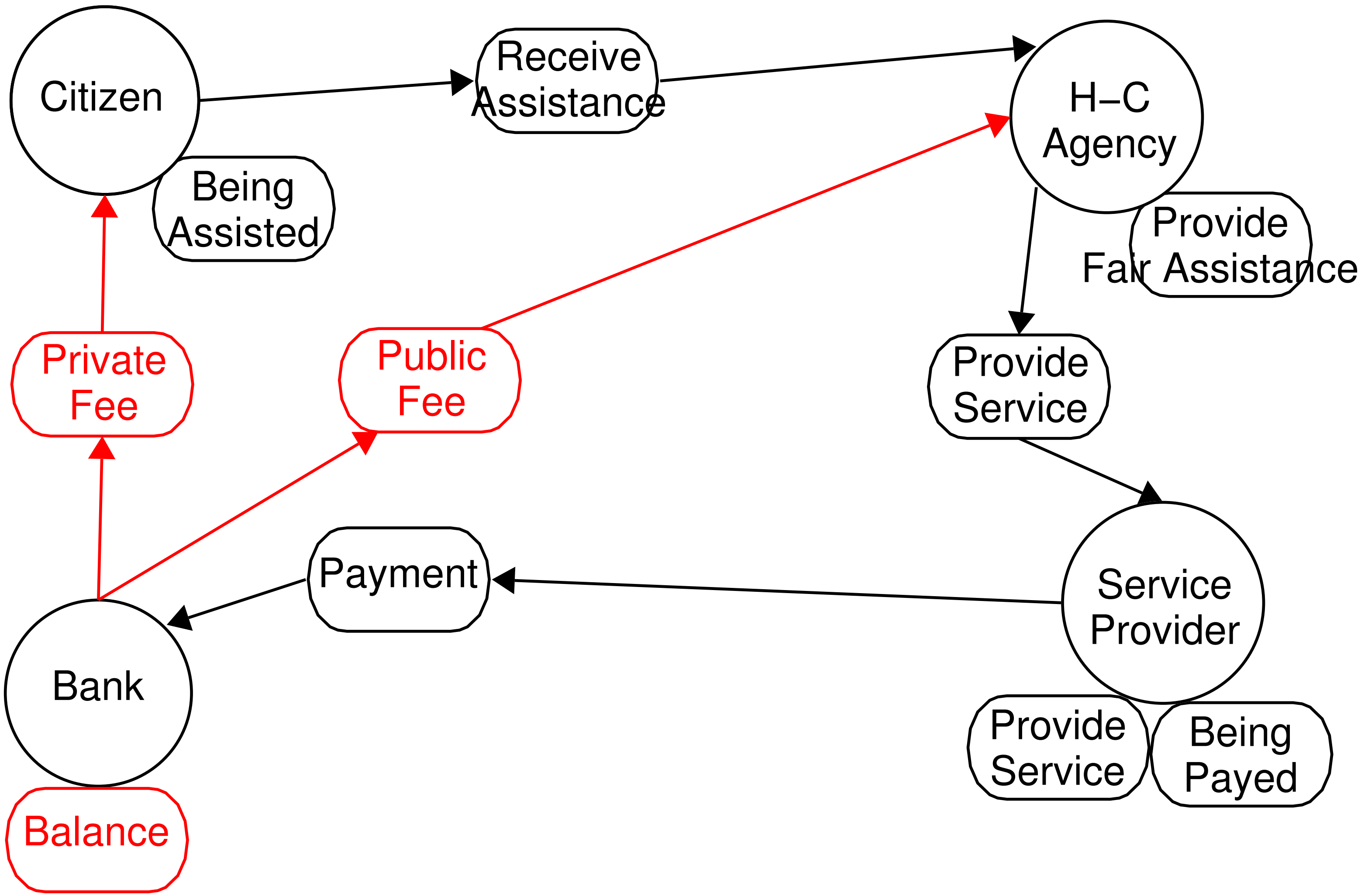
Specifying Business Requirements: Case Study



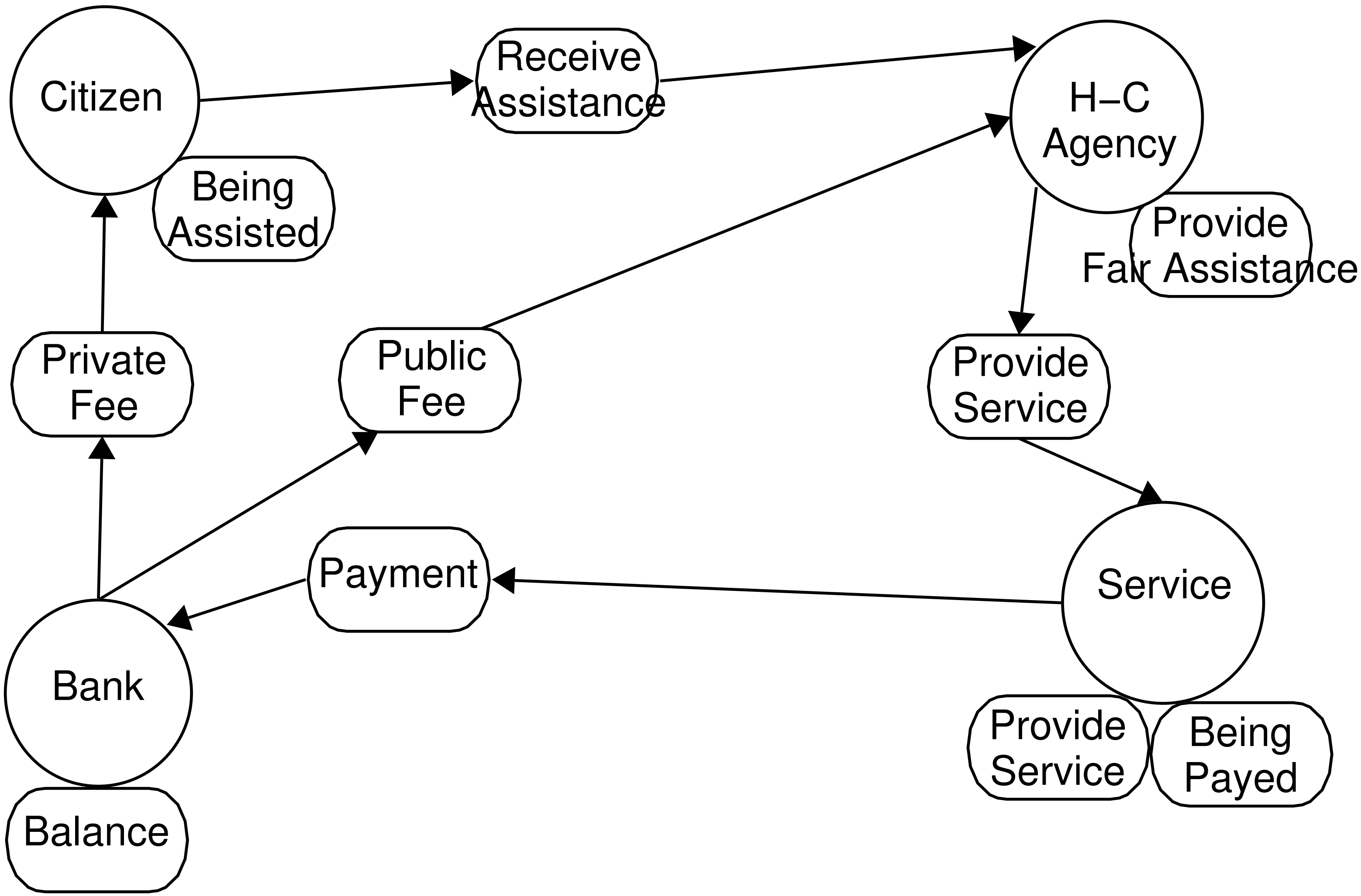
Specifying Business Requirements: Case Study



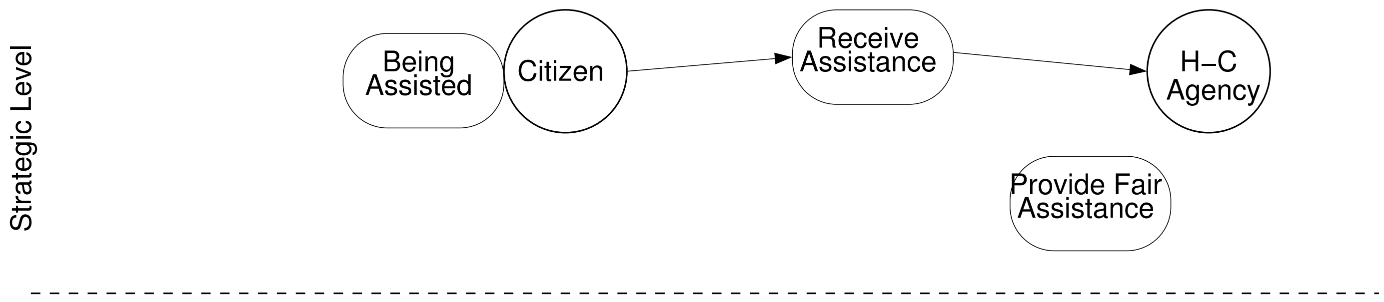
Specifying Business Requirements: Case Study



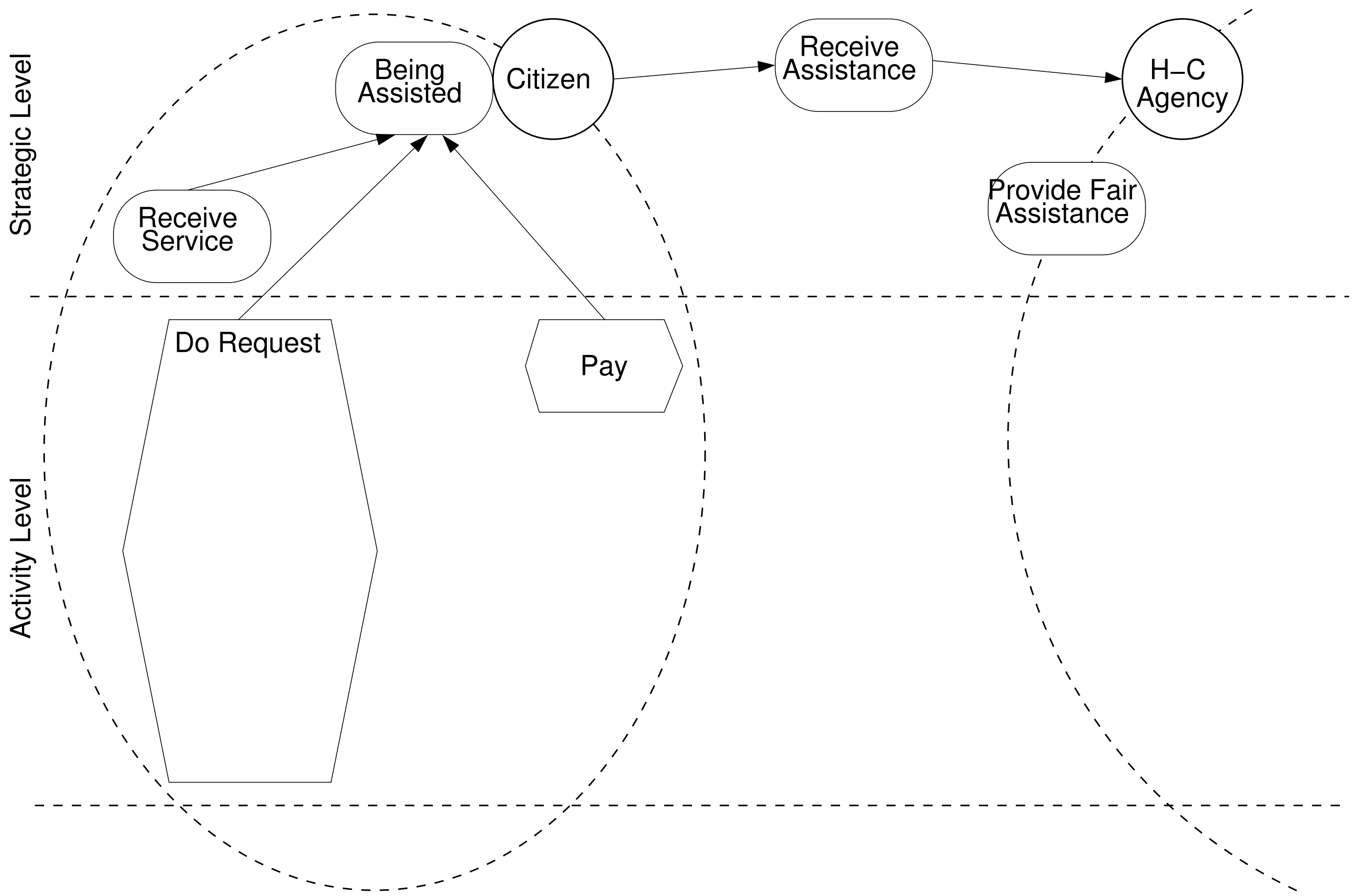
Specifying Business Requirements: Case Study



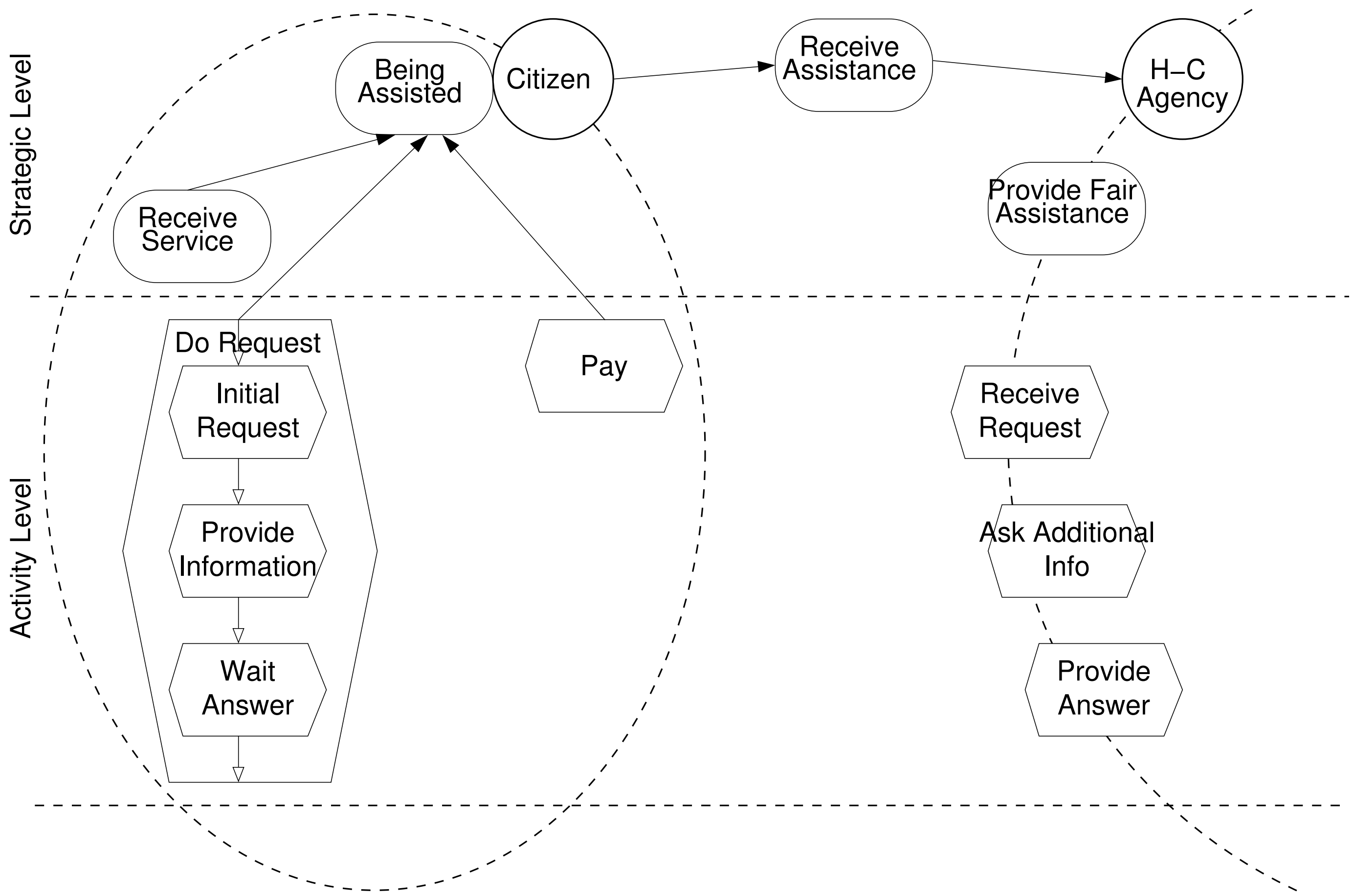
Specifying Business Requirements: Refinement



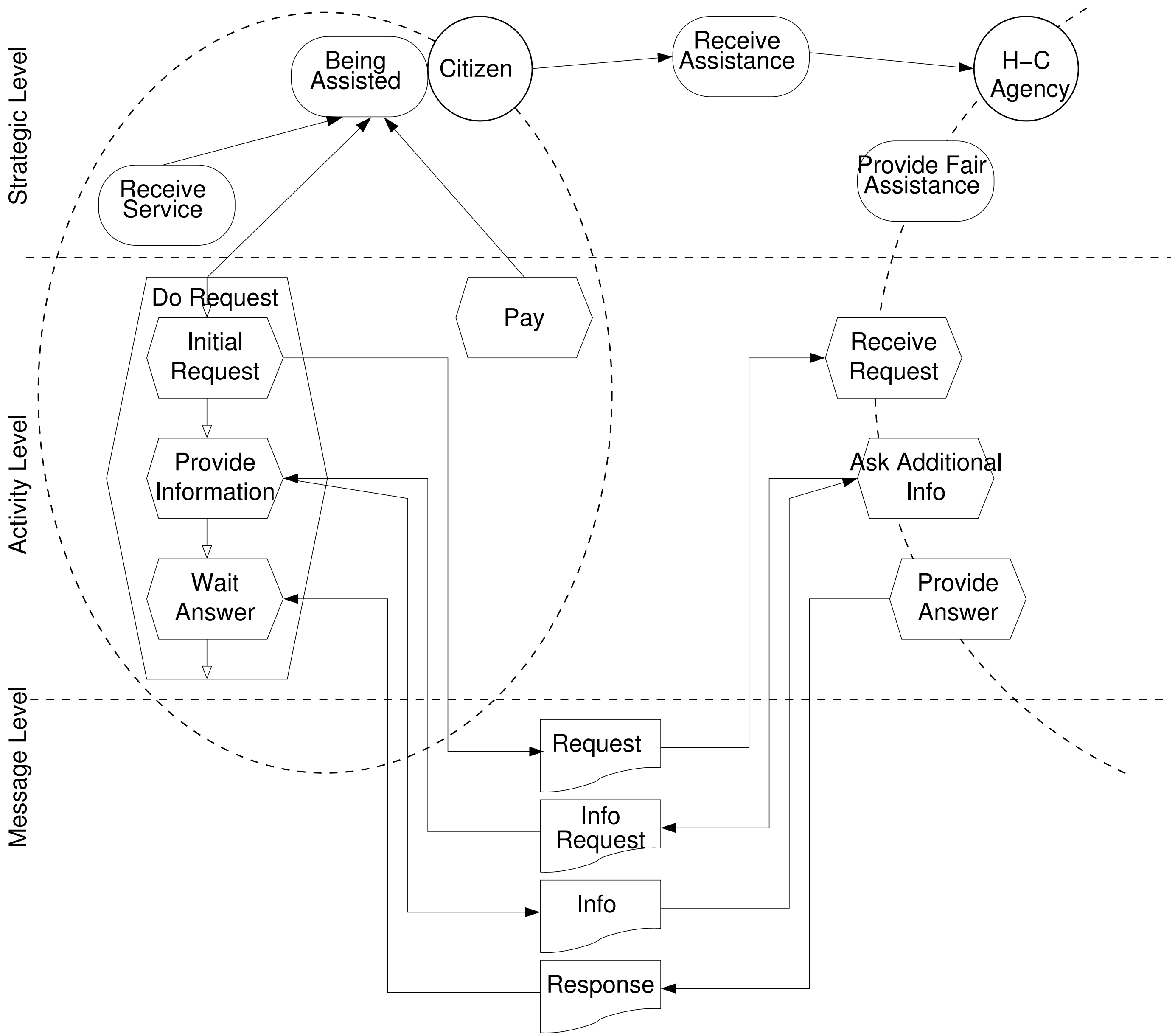
Specifying Business Requirements: Refinement



Specifying Business Requirements: Refinement



Specifying Business Requirements: Refinement

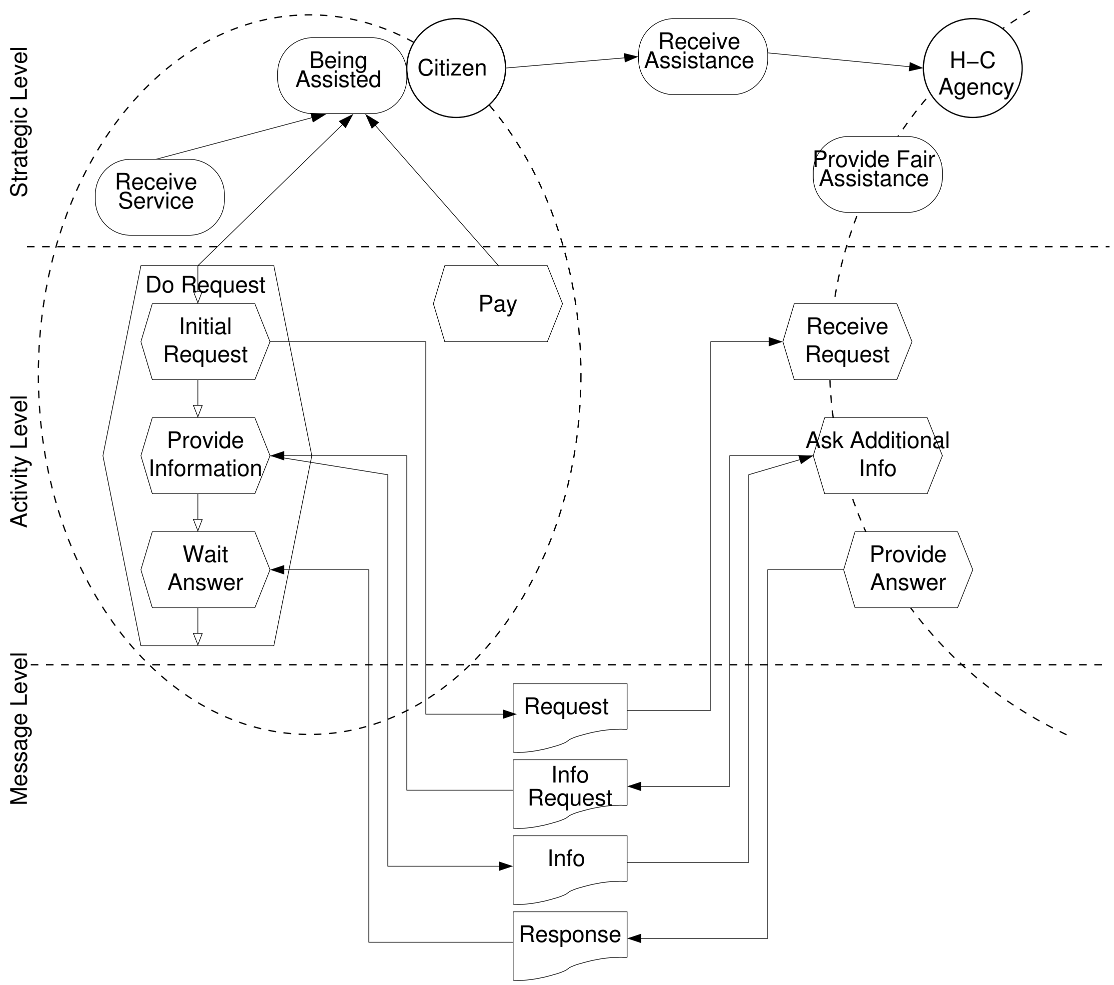


Formal Tropos

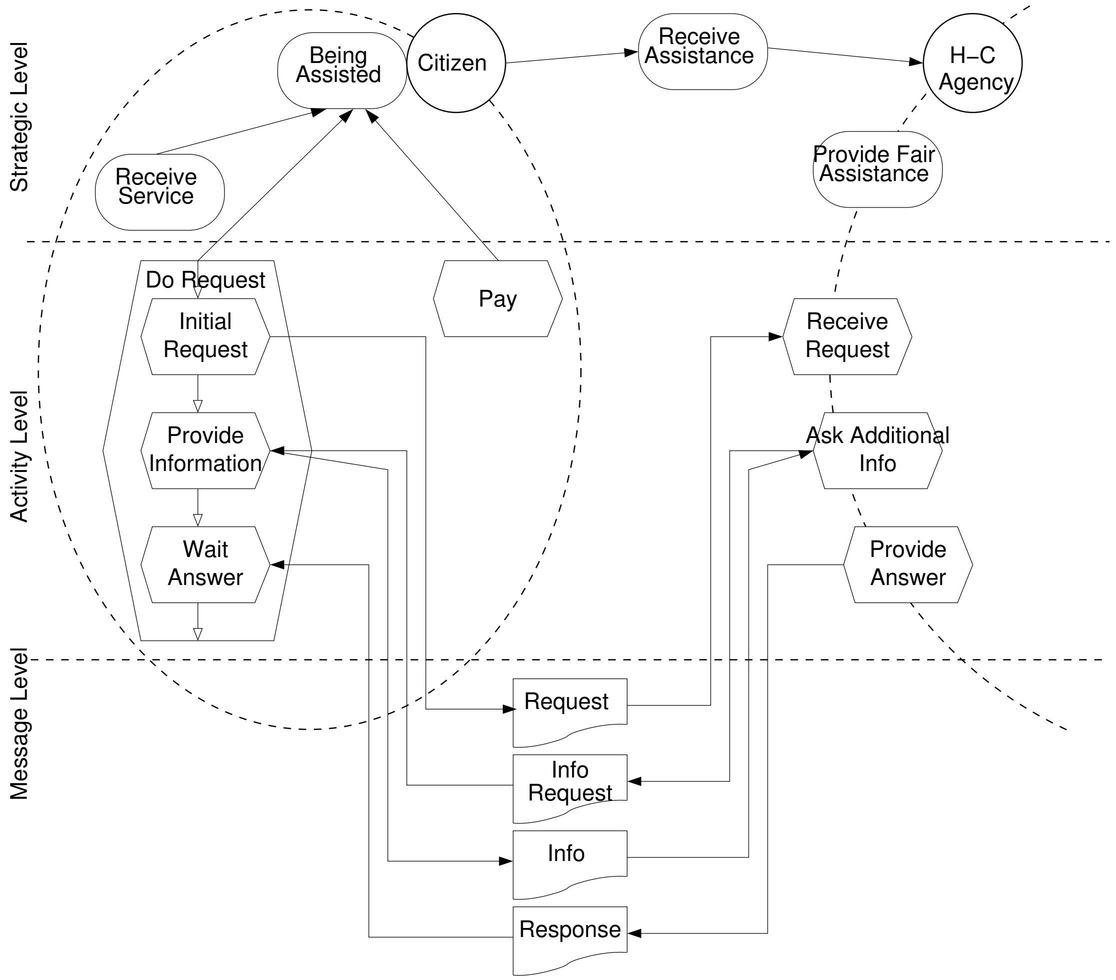
Formal Tropos:

- first-order linear-time temporal constraints on the evolutions of the model:
  - (past and future) temporal operators: $\mathbf{G}\phi$, $\mathbf{F}\phi$, $\mathbf{H}\phi$, $\mathbf{O}\phi$...
  - quantification on class instances: $\forall c : C$..., $\exists c : C$...
- focus on **creation** and **fulfillment** of activities:
  - FT can describe the **state diagram** defining the behavior of services
  - FT can describe the **activity diagram** defining the interaction of services

Specifying Business Requirements: Formal Tropos



Specifying Business Requirements: Formal Tropos



Goal Dependency ReceiveAssistance **Mode maintain**
Depender Citizen **Dependee** HealthcareAgency

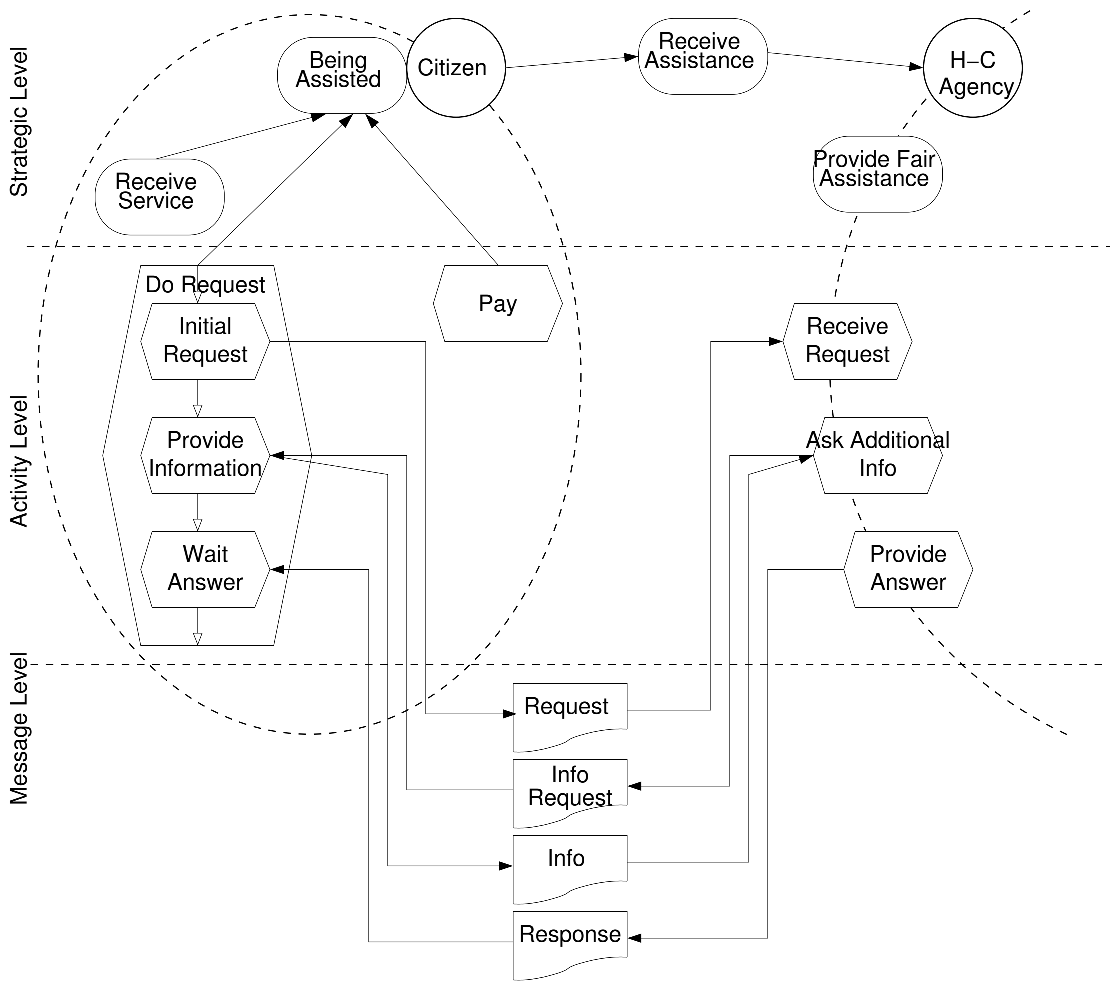
Task DoRequest **Mode achieve**
Super BeingAssisted **Actor** Citizen
Attribute result : **boolean**

Task InitialRequest **Mode achieve**
Super DoRequest **Actor** Citizen

Task ProvideInformation **Mode achieve**
Super DoRequest **Actor** Citizen

Task WaitAnswer **Mode achieve**
Super DoRequest **Actor** Citizen
Attribute result : **boolean**

Specifying Business Requirements: Formal Tropos



Goal Dependency ReceiveAssistance **Mode maintain**
Depender Citizen **Dependee** HealthcareAgency

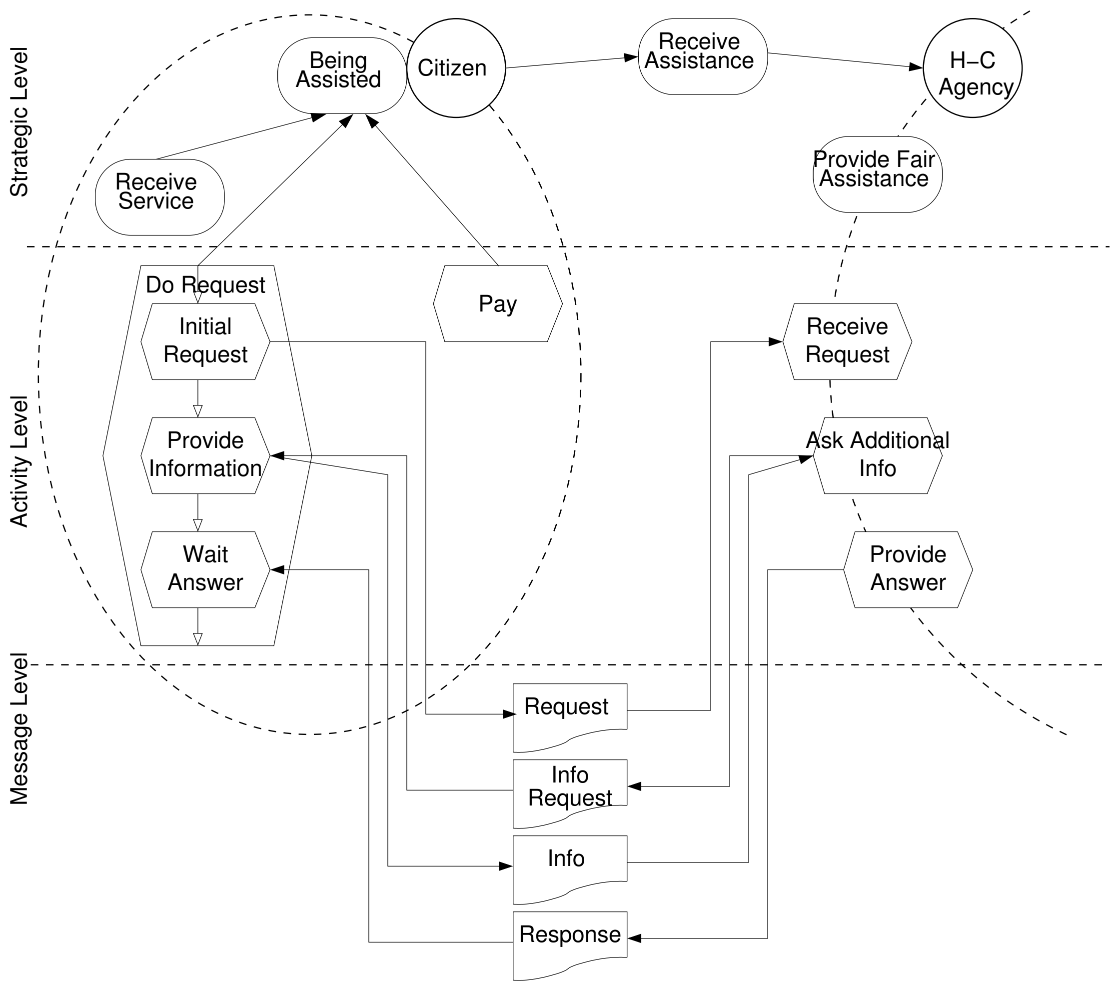
Task DoRequest **Mode achieve**
Super BeingAssisted **Actor** Citizen
Attribute result : **boolean**

Task InitialRequest **Mode achieve**
Super DoRequest **Actor** Citizen
Creation Trigger ∃ dr: DoRequest(**super** = dr)

Task ProvideInformation **Mode achieve**
Super DoRequest **Actor** Citizen
Creation Trigger
∃ dr: DoRequest(**super** = dr ∧
∃ ir: InitialRequest(ir.**super** = dr ∧ **Fulfilled** (ir)))

Task WaitAnswer **Mode achieve**
Super DoRequest **Actor** Citizen
Attribute result : **boolean**
Creation Trigger
∃ dr: DoRequest(**super** = dr ∧
∃ pi: ProvideInformation(pr.**super** = dr ∧ **Fulfilled** (pr)))

Specifying Business Requirements: Formal Tropos



Goal Dependency ReceiveAssistance **Mode maintain**
Depender Citizen **Dependee** HealthcareAgency
Fulfillment condition ∀ dr: DoRequest (
(dr.**actor** = **depender** ∧ **Fulfilled** (dr) ∧ dr.result) →
F ∃ rs: ReceiveService (rs.**actor** = **depender** ∧ **Fulfilled** (rs)))

Task DoRequest **Mode achieve**
Super BeingAssisted **Actor** Citizen
Attribute result : **boolean**
Fulfillment definition
∃ wa:WaitAnswer(wa.**super** = **self** ∧ **Fulfilled** (wa) ∧ (result ↔ wa.result))

Task InitialRequest **Mode achieve**
Super DoRequest **Actor** Citizen
Creation Trigger ∃ dr: DoRequest(**super** = dr)

Task ProvideInformation **Mode achieve**
Super DoRequest **Actor** Citizen
Creation Trigger
∃ dr: DoRequest(**super** = dr ∧
∃ ir: InitialRequest(ir.**super** = dr ∧ **Fulfilled** (ir)))
Fulfillment definition
G (∀ ir: InfoRequest(**Received** (ir) → ∃ i: Info(**Sent** (i)))

Task WaitAnswer **Mode achieve**
Super DoRequest **Actor** Citizen
Attribute result : **boolean**
Creation Trigger
∃ dr: DoRequest(**super** = dr ∧
∃ pi: ProvideInformation(pr.**super** = dr ∧ **Fulfilled** (pr)))
Fulfillment definition
∃ r:Response(**Received** (r) ∧ (result ↔ r.result))

Formal Analysis of Requirements

Formal Tropos enables for the application of formal analysis.

Formal Analysis of Requirements

Formal Tropos enables for the application of formal analysis.

- consistency check: "the specification admits valid scenarios";

Formal Analysis of Requirements

Formal Tropos enables for the application of formal analysis.

- consistency check: “the specification admits valid scenarios”;
- possibility check: “there is *some* scenario for the model that respects certain **possibility** properties”;

Formal Analysis of Requirements

Formal Tropos enables for the application of formal analysis.

- consistency check: "the specification admits valid scenarios";
- possibility check: "there is *some* scenario for the model that respects certain **possibility** properties";

Possibility P1 */* It is possible to fulfill request */*
$\exists$ dr: DoRequest (**Fulfilled** (dr))

Formal Analysis of Requirements

Formal Tropos enables for the application of formal analysis.

- consistency check: “the specification admits valid scenarios”;
- possibility check: “there is *some* scenario for the model that respects certain **possibility** properties”;

Possibility P1 */* It is possible to fulfill request */*
$\exists$ dr: DoRequest (**Fulfilled** (dr))

- assertion validation: “*all* scenarios for the model respect certain **assertion** properties”;

Formal Analysis of Requirements

Formal Tropos enables for the application of formal analysis.

- consistency check: "the specification admits valid scenarios";
- possibility check: "there is *some* scenario for the model that respects certain **possibility** properties";

Possibility P1 */* It is possible to fulfill request */*
$\exists$ dr: DoRequest (**Fulfilled** (dr))

- assertion validation: "*all* scenarios for the model respect certain **assertion** properties";

Assertion A1 */* Service is received only upon a positive response */*
$\forall$ c: Citizen ($\forall$ r: Response (**Received** (r) $\wedge$ r.receiver $=$ c $\rightarrow$ $\neg$ r.result) $\rightarrow$
$\forall$ rs: ReceiveService (rs.**actor** $=$ c $\rightarrow$ $\neg$ **Fulfilled** (rs)))

Assertion A2 */* If some agency provides assistance and the citizen requests a service then the request should be fulfilled */*
$\forall$ dr: DoRequest ($\exists$ ra: ReceiveAssistance (ra.**depender** $=$ dr.**actor** $\wedge$ **Fulfilled** (ra) $\wedge$ $\forall$ r: Request (r.**sender** $=$ dr.**actor** $\rightarrow$ r.**receiver** $=$ ra.**dependee**)) $\rightarrow$ **F Fulfilled** (dr))

Implementing Business Requirements in Promela

Task DoRequest **Mode achieve**
Super BeingAssisted **Actor** Citizen
Attribute result : **boolean**
Fulfillment definition
$\exists$ wa:WaitAnswer(wa.**super** = **self** $\wedge$ **Fulfilled** (wa) $\wedge$
(result $\leftrightarrow$ wa.result))

Task InitialRequest **Mode achieve**
Super DoRequest **Actor** Citizen
Creation Trigger $\exists$ dr: DoRequest(**super** = dr)

Task ProvideInformation **Mode achieve**
Super DoRequest **Actor** Citizen
Creation Trigger
$\exists$ dr: DoRequest(**super** = dr $\wedge$
$\exists$ ir: InitialRequest(ir.**super** = dr $\wedge$ **Fulfilled** (ir)))
Fulfillment definition
G ($\forall$ ir: InfoRequest(**Received** (ir) $\longrightarrow$ $\exists$ i: Info(**Sent** (i)))

Task WaitAnswer **Mode achieve**
Super DoRequest **Actor** Citizen
Attribute result : **boolean**
Creation Trigger
$\exists$ dr: DoRequest(**super** = dr $\wedge$
$\exists$ pi: ProvideInformation(pr.**super** = dr $\wedge$ **Fulfilled** (pr)))
Fulfillment definition
$\exists$ r:Response(**Received** (r) $\wedge$ (result $\leftrightarrow$ r.result))

DoRequest process specification in Promela

```
bool waitResponse;
atomic{
  CREATE ri: InitialRequest;
  ri.super = self;
  waitResponse = true};
atomic{
  CREATEMESSAGE vRequest: Request;
  Request_channel ! vRequest};
atomic{
  FULFILL ir: InitialRequest [ir.super == self];
  CREATE pi: ProvideInformation; pi.super = self};
do::atomic{ waitResponse ->
    if::InfoRequest_channel ? vInfoRequest;
        CREATEMESSAGE vInfo : Info;
        vInfo.reference = vInfoRequest;
        Info_channel ! vInfo;
      ::Response_channel ? vResponse;
        FULFILL pi: ProvideInformation [pi.super==self];
        CREATE wa: WaitAnswer; wa.super = self;
        waitResponse = false;
        self.result = vResponse.result;
    fi};
  ::else break;
od;
atomic{
  FULFILL wait: WaitAnswer [wait.super == self];
  FULFILL self};
```

Implementing Business Requirements in BPEL4WS

Task DoRequest **Mode achieve**
Super BeingAssisted **Actor** Citizen
Attribute result : **boolean**
Fulfillment definition
$\exists$ wa:WaitAnswer(wa.**super** = **self** $\wedge$ **Fulfilled** (wa) $\wedge$
(result $\leftrightarrow$ wa.result))

Task InitialRequest **Mode achieve**
Super DoRequest **Actor** Citizen
Creation Trigger $\exists$ dr: DoRequest(**super** = dr)

Task ProvideInformation **Mode achieve**
Super DoRequest **Actor** Citizen
Creation Trigger
$\exists$ dr: DoRequest(**super** = dr $\wedge$
$\exists$ ir: InitialRequest(ir.**super** = dr $\wedge$ **Fulfilled** (ir)))
Fulfillment definition
G ($\forall$ ir: InfoRequest(**Received** (ir) $\longrightarrow$ $\exists$ i: Info(**Sent** (i)))

Task WaitAnswer **Mode achieve**
Super DoRequest **Actor** Citizen
Attribute result : **boolean**
Creation Trigger
$\exists$ dr: DoRequest(**super** = dr $\wedge$
$\exists$ pi: ProvideInformation(pr.**super** = dr $\wedge$ **Fulfilled** (pr)))
Fulfillment definition
$\exists$ r:Response(**Received** (r) $\wedge$ (result $\leftrightarrow$ r.result))

```
<sequence name="DoRequestBody">
  <assign name="Initialization"
    event="Create ir:InitialRequest(ir.super=self)">
    <copy> <from expression="true()"/><to variable="waitResponse"/> </copy>
  </assign>
  <invoke operation="oRequest" inputVariable="vRequest"/>
  <empty name="PhaseSwitch"
    event="Fulfill ir:InitialRequest(ir.super=self) &
           Create pi:ProvideInformation(pi.super=self)"/>
  <while condition="getVariableData('waitResponse')">
    <pick name="WaitMessage">
      <onMessage operation="oInfoRequest" variable="vInfoRequest">
        <reply operation="oInfo" variable="vInfo"/>
      </onMessage>
      <onMessage operation="oResponse" variable="vResponse"
        event="Fulfill pi:ProvideInformation(pi.super=self) &
               Create wa:WaitAnswer(wa.super=self)">
        <assign name="LeaveLoop">
          <copy> <from expression="false()"/><to variable="waitResponse"/> </copy>
          <copy> <from variable="vResponse" part="result"/><to variable="result"/>
        </assign>
      </onMessage>
    </pick>
  </while>
  <empty name="DoRequestFulfilled"
    event="Fulfill wa:WaitAnswer(wa.super=self)"
    constraint="Forall wa:WaitAnswer(wa.super=self →
                G(wa.result↔self.result))"/>
</sequence>
```

Encoding Formal Tropos in Promela

Task DoRequest
Actor Citizen
Super BeingAssisted
Attribute result : **boolean**

Encoding Formal Tropos in Promela

Task DoRequest
  Actor Citizen
  Super BeingAssisted
  Attribute result : **boolean**

typedef DoRequestType{
  byte actor;
  byte super;
  bool result;

}

Encoding Formal Tropos in Promela

Task DoRequest
  Actor Citizen
  Super BeingAssisted
  Attribute result : **boolean**

typedef DoRequestType{
  byte actor;
  byte super;
  bool result;
  bool justcreated, exists;
  bool justfulfilled, fulfilled;
}
DoRequestType DoRequest[2];

Encoding Formal Tropos in Promela

Task DoRequest
  Actor Citizen
  Super BeingAssisted
  Attribute result : **boolean**

```
typedef DoRequestType{
    byte actor;
    byte super;
    bool result;
    bool justcreated, exists;
    bool justfulfilled, fulfilled;
}
DoRequestType DoRequest[2];

proctype DoRequestProc(byte id) {
.../* life cycle of class instance */
.../* encoded as a Promela process */
}
```

Encoding Formal Tropos in Promela

```
proctype ClassProc(byte id) {

}
```

The life-cycle of a Class instance:

Encoding Formal Tropos in Promela

```
proctype ClassProc(byte id) {
  NotExists:
     do
        /* Initial status for class instance */

     od





}
```

The life-cycle of a Class instance:

- **NotExists**: The initial status of class instances (only for actors).
  - It can stay in this state or go to next state.
  - Transition to next state only if conditions for creation hold.

Encoding Formal Tropos in Promela

```
proctype ClassProc(byte id) {
   NotExists:
      do
         /* Initial status for class instance */

      od
   Exists:
      do
         /* start child sub classes */

      od




}
```

The life-cycle of a Class instance:

- **NotExists**: The initial status of class instances (only for actors).
  - It can stay in this state or go to next state.
  - Transition to next state only if conditions for creation hold.
- **Exists**: The class instance exists.
  - It can stay in this state or go to next state.

Encoding Formal Tropos in Promela

```
proctype ClassProc(byte id) {
  NotExists:
    do
      /* Initial status for class instance */

    od
  Exists:
    do
      /* start child sub classes */

    od
  Fulfilled:
    do
      /* stay here forever */

    od
}
```

The life-cycle of a Class instance:

- **NotExists**: The initial status of class instances (only for actors).
  - It can stay in this state or go to next state.
  - Transition to next state only if conditions for creation hold.
- **Exists**: The class instance exists.
  - It can stay in this state or go to next state.
- **Fulfilled**: The class instance is fulfilled (only for tasks, goals, dep.)
  - It stay in this state.

Encoding Formal Tropos in Promela

```
proctype DoRequestProc(byte id) {
Exists:
  do :: atomic /* if the child subtask is not started yet,
                  assign relevant attributes and start it */
       {(!InitialRequest[0].exists)→ system_step();
        InitialRequest[0].super = id;
        InitialRequest[0].actor = DoRequest[id].actor;
        InitialRequest[0].exists = true;
        InitialRequest[0].justcreated = true;
        run InitialRequestProc(0);};
     . . . /* other child subtask may be started here */
     :: atomic /* Modify non-constant attributes */
       {system_step();
        if :: DoRequest[id].result = true;
           :: DoRequest[id].result = false;
        fi; /* The instance is fulfilled nondeterministically */
        if :: DoRequest[id].fulfilled = false;
           :: DoRequest[id].fulfilled = true;
              DoRequest[id].justfulfilled = true; goto Fulfilled;
        fi;}
  od;
```

The DoRequestProc instance: **Exists**

- Transition from **NotExists** to **Exists** only if conditions hold.
  - Class attributes initialized.
  - justcreated and exists set to true.
- Class can nondeterministically create child goals, tasks, dependencies, . . .
  - Child attributes are initialized.
  - Child corresponding processes started.
- In this phase the process nondeterministically modifies values of non-constant attributes.

Encoding Formal Tropos in Promela

```
proctype DoRequestProc(byte id) {
Exists:
      ...
      :: atomic /* Modify non-constant attributes */
         {system_step();
          if :: DoRequest[id].result = true;
             :: DoRequest[id].result = false;
          fi; /* The instance is fulfilled nondeterministically */
          if :: DoRequest[id].fulfilled = false;
             :: DoRequest[id].fulfilled = true;
                DoRequest[id].justfulfilled = true; goto Fulfilled;
          fi;}
 od;
Fulfilled:
 do :: atomic /* Modify non-constant attributes */
         {system_step();
          if :: DoRequest[id].result = true;
             :: DoRequest[id].result = false;
          fi;}
 od;
}
```

The DoRequestProc instance: **Fulfilled**

- Transition from **Exists** to **Fulfilled** nondeterministic.
  - justfulfilled and fulfilled set to true.
- In this phase the process nondeterministically modifies values of non-constant attributes.

Encoding Formal Tropos in Promela: Remarks

```
proctype DoRequestProc(byte id) {
Exists:
  do :: atomic
         /* if the child subtask is not started yet,
          assign relevant attributes and start it */
        {(!InitialRequest[0].exists)→ system_step();
         . . .
     :: atomic /* Modify non-constant attributes */
        {system_step();
         . . .
         }
  od;
Fulfilled:
  do :: atomic /* Modify non-constant attributes */
        {system_step();
         . . .
        }   od;
}
```

- All transitions from life-cycles performed within an **atomic** statement to preserve FT semantics.
- system_step() invoked each time a process performs a step.
  - reset all attributes justcreated and justfulfilled.
  - other activities related to the verification

Encoding Formal Tropos in Promela: Logic Specifications

- FT logic specifications C_i (creation, invariant, fulfillment constraints) exploited to verify assertions and possibilities.

Encoding Formal Tropos in Promela: Logic Specifications

- FT logic specifications C_i (creation, invariant, fulfillment constraints) exploited to verify assertions and possibilities.
- For assertions

$$\bigwedge_{i \in I} C_i \rightarrow A$$

must be valid

Encoding Formal Tropos in Promela: Logic Specifications

- FT logic specifications C_i (creation, invariant, fulfillment constraints) exploited to verify assertions and possibilities.
- For assertions

$$\bigwedge_{i \in I} C_i \rightarrow A$$

  must be valid
- For possibilities

$$\bigwedge_{i \in I} C_i \wedge P$$

  must be satisfiable

Encoding Formal Tropos in Promela: Logic Specifications

- FT logic specifications C_i (creation, invariant, fulfillment constraints) exploited to verify assertions and possibilities.
- For assertions

$$\bigwedge_{i \in I} C_i \rightarrow A$$

  must be valid
- For possibilities

$$\bigwedge_{i \in I} C_i \wedge P$$

  must be satisfiable
- Build a *never claim* for the formula to verify and submit it to SPIN.

Encoding Formal Tropos in Promela: Logic Specifications

- FT logic specifications C_i (creation, invariant, fulfillment constraints) exploited to verify assertions and possibilities.
- For assertions

$$\bigwedge_{i \in I} C_i \rightarrow A$$

  must be valid
- For possibilities

$$\bigwedge_{i \in I} C_i \wedge P$$

  must be satisfiable
- Build a *never claim* for the formula to verify and submit it to SPIN.
  - Problem: on small cases the size of the formula prevents possibility to verify the never claim.
    - A reduced FT specification with 3 simple constraints and 5 classes generated a file whose size was not manageable by the C compiler.

Encoding Formal Tropos in Promela: Logic Specifications

- Encode each FT constraint C_i as a separate automata.
- Generate a new process constraint_verifier() where all automata are executed in parallel.
- Add the constraint_verifier() to the final Promela specification.
- Enforce execution of constraint_verifier() after each system step.
- Restrict the verification to *valid execution paths* i.e. to those execution sequences where all constraints holds.

Encoding Formal Tropos in Promela: Logic Specifications

```
                           if          /* label[n] preserves position reached at previous step */
                           :: label[n]==0 → goto Cn_accept_init
                           :: label[n]==1 → goto Cn_T0_S2
                           fi;
/* G(p → Fq) */            /* G(p → Fq) */
accept_init:               Cn_accept_init:
  if                         if
  :: (¬p)||q →               :: (¬p)||q → label[n] = 0;
    goto accept_init             accepted[n] = true;
  :: (1) →                   :: (1) → label[n] = 1;
    goto T0_S2                   accepted[n] = false; all_accepted = false;
  fi;                        fi; goto Cn_checked;
T0_S2:                     Cn_T0_S2:
  if                         if
  :: q →                     :: q → label[n] = 0;
    goto accept_init             accepted[n] = true;
  :: (1) →                   :: (1) → label[n] = 1;
    goto T0_S2                   accepted[n] = false; all_accepted = false;
  fi;                        fi; goto Cn_checked;
                           Cn_checked:
```

Encoding Formal Tropos in Promela: Logic Specifications

```
proctype constraint_verifier() {
   byte label[n] = 0; bool accepted[n] = false; byte next = 0;
   do :: constraints_done → break;
      :: else atomic
         {all_accepted = true; valid_step = false;
          ...  /* All constraints automata go here */
          valid_step = true; constraints_done = true;
          if :: accepted[next] → /* Look for acceptance again */
                  next_accepted = true; next = (next+1) % n;
             :: else
          fi;}
   od;}
```

```
inline system_step() {
   if :: constraints_done → constraints_done = false;
      :: else valid_step = false;
   fi;
   next_accepted = false;
   ... /* Reset justcreated and justfulfilled flags */
   DoRequest[0].justcreated = false;
   DoRequest[0].justfulfilled = false;
}
```

- constraints_done is set to true each time process constraint_verifier() evolves, to false each time the system_step() evolves.

Encoding Formal Tropos in Promela: Logic Specifications

```
proctype constraint_verifier() {
   byte label[n] = 0; bool accepted[n] = false; byte next = 0;
   do :: constraints_done → break;
      :: else atomic
         {all_accepted = true; valid_step = false;
          ... /* All constraints automata go here */
          valid_step = true; constraints_done = true;
          if :: accepted[next] → /* Look for acceptance again */
                next_accepted = true; next = (next+1) % n;
             :: else
          fi;}
   od;}
```

```
inline system_step() {
   if :: constraints_done → constraints_done = false;
      :: else valid_step = false;
   fi;
   next_accepted = false;
   ... /* Reset justcreated and justfulfilled flags */
   DoRequest[0].justcreated = false;
   DoRequest[0].justfulfilled = false;
}
```

- valid_step is true if each system step is followed by exactly one step of process constraint_verifier() and if the execution is not blocked.

Encoding Formal Tropos in Promela: Logic Specifications

```
proctype constraint_verifier() {
   byte label[n] = 0; bool accepted[n] = false; byte next = 0;
   do :: constraints_done → break;
      :: else atomic
         {all_accepted = true; valid_step = false;
          ... /* All constraints automata go here */
          valid_step = true; constraints_done = true;
          if :: accepted[next] → /* Look for acceptance again */
                next_accepted = true; next = (next+1) % n;
             :: else
          fi;}
   od;}
```

```
inline system_step() {
   if :: constraints_done → constraints_done = false;
      :: else valid_step = false;
   fi;
   next_accepted = false;
   ... /* Reset justcreated and justfulfilled flags */
   DoRequest[0].justcreated = false;
   DoRequest[0].justfulfilled = false;
}
```

- all_accepted store information whether all automata are visiting an acceptance state simultaneously.

Encoding Formal Tropos in Promela: Logic Specifications

```
proctype constraint_verifier() {
  byte label[n] = 0; bool accepted[n] = false; byte next = 0;
  do :: constraints_done → break;
     :: else atomic
       {all_accepted = true; valid_step = false;
        ... /* All constraints automata go here */
        valid_step = true; constraints_done = true;
        if :: accepted[next] → /* Look for acceptance again */
              next_accepted = true; next = (next+1) % n;
           :: else
        fi;}
  od;}
```

```
inline system_step() {
  if :: constraints_done → constraints_done = false;
     :: else valid_step = false;
  fi;
  next_accepted = false;
  ... /* Reset justcreated and justfulfilled flags */
  DoRequest[0].justcreated = false;
  DoRequest[0].justfulfilled = false;
}
```

- next_accepted is set to true if accepted[next] is set to true. It is used to check that all constraint automata visit acceptance states.

Encoding Formal Tropos in Promela: Logic Specifications

```
proctype constraint_verifier() {
  byte label[n] = 0; bool accepted[n] = false; byte next = 0;
  do :: constraints_done → break;
     :: else atomic
       {all_accepted = true; valid_step = false;
        ... /* All constraints automata go here */
        valid_step = true; constraints_done = true;
        if :: accepted[next] → /* Look for acceptance again */
              next_accepted = true; next = (next+1) % n;
           :: else
        fi;}
  od;}
```

```
inline system_step() {
  if :: constraints_done → constraints_done = false;
     :: else valid_step = false;
  fi;
  next_accepted = false;
  ... /* Reset justcreated and justfulfilled flags */
  DoRequest[0].justcreated = false;
  DoRequest[0].justfulfilled = false;
}
```

- next is updated such that all constraints are considered in turn.

Encoding Formal Tropos in Promela: Logic Specifications

- The restriction of the verification to the valid execution paths is captured by the following formula:

$$\mathbf{G}(\texttt{valid\_step} \wedge \mathbf{F}\,\texttt{next\_accepted} \wedge \mathbf{G}(\texttt{next\_accepted} \rightarrow \mathbf{G}\,\texttt{all\_accepted}))$$

- It states that...
  - the constraint automata are not blocked,
  - they visit acceptance states infinitely often,
  - if variable next_accepted stay true forever (execution over finite paths) then variable all_accepted will stay true forever.

Encoding Formal Tropos in Promela: Logic Specifications

The verification of FT thus is performed as:

Encoding Formal Tropos in Promela: Logic Specifications

The verification of FT thus is performed as:

- for an assertions A we verify:

$$\mathbf{G}(\texttt{valid\_step} \wedge \texttt{next\_accepted} \wedge \mathbf{G}(\texttt{next\_accepted} \rightarrow \texttt{G}\ \texttt{all\_accepted})) \rightarrow A$$

  - It checks whether all the valid execution paths satisfy the assertion A.

Encoding Formal Tropos in Promela: Logic Specifications

The verification of FT thus is performed as:

- for an assertions A we verify:

$$\mathbf{G}(\texttt{valid\_step} \wedge \texttt{next\_accepted} \wedge \\ \mathbf{G}(\texttt{next\_accepted} \rightarrow \texttt{G}\ \texttt{all\_accepted})) \\ \rightarrow A$$

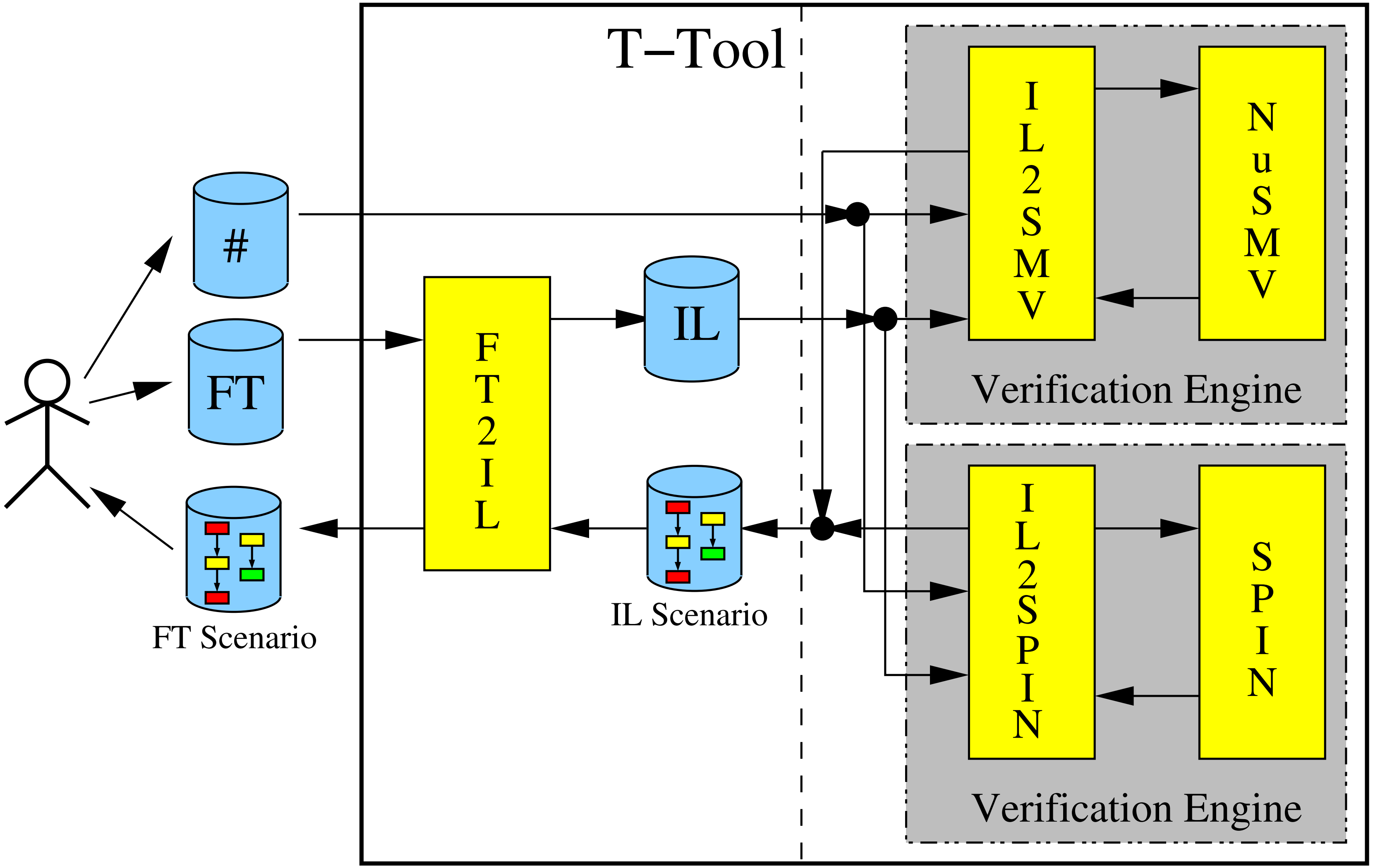
  - It checks whether all the valid execution paths satisfy the assertion A.

- for a possibility P we verify:

$$\mathbf{G}(\texttt{valid\_step} \wedge \texttt{next\_accepted} \wedge \\ \mathbf{G}(\texttt{next\_accepted} \rightarrow \mathbf{G}\ \texttt{all\_accepted})) \\ \rightarrow \neg P$$

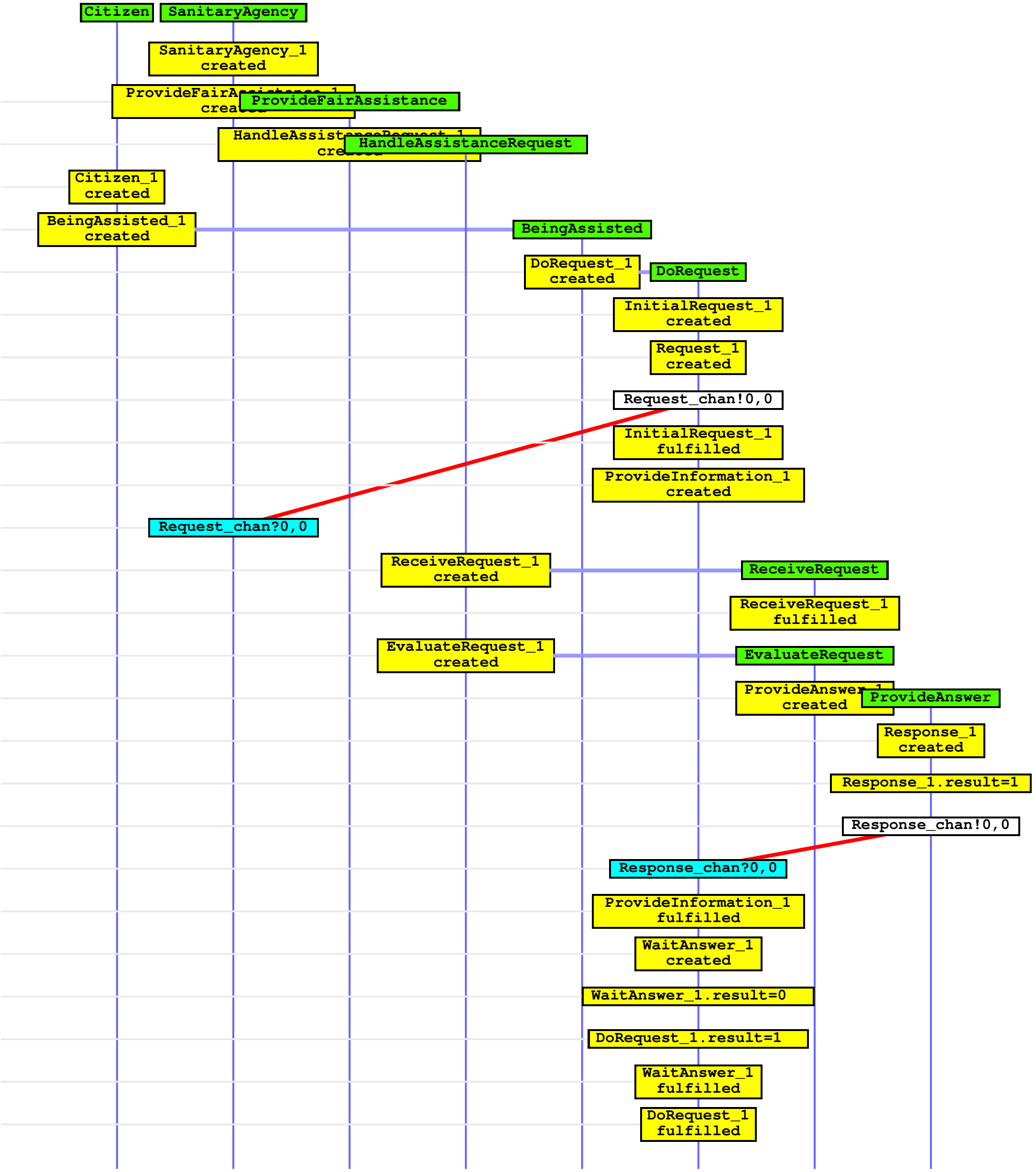
  - If a counter-example is found for such formula, it is a witness for P.

The T-TOOL



T–Tool
#
FT
FT Scenario
FT2IL
IL
IL Scenario
IL2SMV
NuSMV
Verification Engine
IL2SPIN
SPIN
Verification Engine

A counter-example produced by T-TOOL



Experimental Analysis

Logic specification translation

	Direct Translation		Compositional Translation	
	1 instance	1..2 instances	1 instance	1..2 instances
1 constraint	0,01sec	0,01sec	0,01sec	0,01sec
3 constraints	0,03sec	3,01sec	0,03sec	0,09sec
5 constraints	0,11sec	MO	0,04sec	0,14sec
10 constraints	10,65sec	SF	0,04sec	0,16sec
30 constraints	MO	SF	0,07sec	0,20sec
45 constraints	SF	SF	0,15sec	0,41sec

Experimental Analysis

Property verification results

SPIN results			
		1 instance	**1..2 instances**
A1	**HC4**	TO - 1284steps - 1382Mb	TO - 2857steps - 362Mb
	BITSTATE	Valid$^{(a)}$ - 21sec - 61Mb	TO - 3244steps - 1028Mb
	3SPIN	Valid$^{(b)}$ - 23sec - 69Mb	TO - 3207steps - 1178Mb
A2	**HC4**	TO - 1393steps - 1382Mb	TO - 2857steps - 362Mb
	BITSTATE	Invalid - 21sec - 52Mb	TO - 3244steps - 1058Mb
	3SPIN	Invalid - 24sec - 64Mb	TO - 3417steps - 1173Mb
P1	**HC4**	Valid - 27sec - 68Mb	TO - 2857steps - 362Mb
	BITSTATE	Valid - 14sec - 41Mb	TO - 3099steps - 956Mb
	3SPIN	Valid - 19sec - 56Mb	TO - 3312steps - 1143Mb

Hash factors: $^{(a)}$ 1.97 – $^{(b)}$ 3.35

Experimental Analysis

Property verification results

NuSMV results			
		1 instance	**1..2 instances**
A1	**BDD**	Valid - 9sec - 6,0Mb	TO - 235Mb
	BMC	Undec.$^{(*)}$ - 7sec - 20,4Mb	Undec.$^{(*)}$ - 106sec - 61,2Mb
A2	**BDD**	Invalid - 11sec - 6,9Mb	TO - 235Mb
	BMC	Invalid - 0,6sec - 3,8Mb	Invalid - 2sec - 11,3Mb
P1	**BDD**	Valid - 10sec - 5,8Mb	TO - 235Mb
	BMC	Valid$^{(**)}$ - 0,7sec - 5,3Mb	Valid$^{(**)}$ - 2sec - 16,0Mb

$^{(*)}$ No counter-example found up to bound length 10

$^{(**)}$ Found example of length 4 satisfying P1

Experimental Analysis

Implementation verification result

		1 instance	1..2 instances
A1	**HC4**	TO - 516steps - 1442Mb	TO - 341steps - 1282Mb
	BITSTATE	Valid$^{(a)}$ - 32sec - 83Mb	Valid$^{(b)}$ - 169sec - 316Mb
	3SPIN	Valid$^{(c)}$ - 14sec - 35Mb	Valid$^{(d)}$ - 74sec - 171Mb
A2	**HC4**	Invalid - 125sec - 206Mb	TO - 341steps - 1162Mb
	BITSTATE	Invalid - 32sec - 71Mb	Invalid - 1285sec - 2003Mb
	3SPIN	Invalid - 15sec - 32Mb	MO - 673steps - 1141sec
P1	**HC4**	Valid - 2sec - 9,1Mb	TO - 341steps - 1282Mb
	BITSTATE	Valid - 3sec - 10,1Mb	Valid - 167sec - 306Mb
	3SPIN	Valid - 3sec - 12,0Mb	Valid - 59sec - 148Mb
C	**HC4**	Invalid - 2sec - 9,1Mb	TO - 341steps - 1282Mb
	BITSTATE	Invalid - 3sec - 11,4Mb	Invalid - 166sec - 306Mb
	3SPIN	Invalid - 3sec - 12,0Mb	Invalid - 62sec - 151Mb

Hash factors: $^{(a)}$ 2.44 – $^{(b)}$ 1.66 – $^{(c)}$ 6.06 – $^{(d)}$ 1.61

Conclusions

- We have proposed a framework for the specification and verification of early requirements based on explicit state model checking and SPIN.

Conclusions

- We have proposed a framework for the specification and verification of early requirements based on explicit state model checking and SPIN.
- We have extend the scope of the verification to include the design of distributed processes defined in Promela.

Conclusions

- We have proposed a framework for the specification and verification of early requirements based on explicit state model checking and SPIN.
- We have extend the scope of the verification to include the design of distributed processes defined in Promela.
- We have proposed a novel, compositional encoding of the LTL constraints that define the valid behaviors of the requirements model in the verification tasks.

Conclusions

- We have proposed a framework for the specification and verification of early requirements based on explicit state model checking and SPIN.
- We have extend the scope of the verification to include the design of distributed processes defined in Promela.
- We have proposed a novel, compositional encoding of the LTL constraints that define the valid behaviors of the requirements model in the verification tasks.
- The preliminary experiments show that the approach is viable, even if the performance is currently a rather serious limit for its applicability.

Conclusions

- We have proposed a framework for the specification and verification of early requirements based on explicit state model checking and SPIN.
- We have extend the scope of the verification to include the design of distributed processes defined in Promela.
- We have proposed a novel, compositional encoding of the LTL constraints that define the valid behaviors of the requirements model in the verification tasks.
- The preliminary experiments show that the approach is viable, even if the performance is currently a rather serious limit for its applicability.
- Future work
  - Optimize the model generator by integrating advanced abstraction techniques that exploit, for instance, possible symmetries in the specification.
  - Deeper investigation of the compositional approach for the verification of complex LTL specifications.

The End