

Model Checking Commitment-Governed Compositions of Web Services

Ana Vazquez

A Thesis
in
The Department
of
The Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montréal, Québec, Canada

December 2015

© Ana Vazquez, 2015

CONCORDIA UNIVERSITY

Division of Graduate Studies

This is to certify that the thesis prepared

By: **Ana Vazquez**

Entitled: **Model Checking Commitment-Governed Compositions of Web Services**

and submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Dr. Abdessamad Ben Hamza (Chair)

_____ Dr. Juergen Rilling, CSE

_____ Dr. Rachida Dssouli, CIISE

_____ Dr. Jamal Bentahar (Supervisor)

Approved by _____

Director of CIISE

_____ 2015 _____

Dean of Engineering

ABSTRACT

Model Checking Commitment-Governed Compositions of Web Services

Ana Vazquez, MASc

Concordia University, 2015

We propose a new approach towards verifying compositions of web services using model checking. In order to perform such a verification, we transform the web service composition into a Multi-Agent System (MAS) model where the process in charge of the composition and the participating services are represented by agents. We model the behavior of the resulting MAS using the extended Interpreted Systems Programming Language (ISPL+), the dedicated language of the MCMAS+ model checker for MAS. We use commitments between agents to regulate and reason about messages between composite web services. The properties against which the compositions are verified are expressed in the Computation Tree Logic of Commitments (CTLC), an extension of the branching logic CTL that supports commitment modalities.

We describe BPEL2ISPL+, a tool we developed to perform the automatic transformation from the web service composition described in Business Process Execution Language (BPEL) into a verifiable MAS model described in ISPL+. The BPEL2ISPL+ tool is applied to a concrete BPEL web service composition and its accurate representation in ISPL+ is obtained. The CTLC properties used to verify the compositions regulated by commitments are represented along with the agents abstracting the participating web services. The MCMAS+ model checker is used to verify the model against these properties, providing thus a new approach to model check agent-based web service compositions governed by commitments.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Jamal Bentahar for giving me the opportunity to work in an interesting topic under his supervision. I am very grateful to him for his valuable suggestions throughout the preparation of this thesis. I would also like to thank Dr. Mohamed El-Menshawy for his very valuable guidance and involvement in several aspects of this thesis.

I would like to thank Dr. Juergen Rilling from CSE and Dr. Rachida Dssouli from CIISE for accepting the task of evaluating my work by being in my examination committee and Dr. Abdessamad Ben Hamza from CIISE for chairing the committee. Their time and effort are greatly appreciated.

Also, I would like to thank my lab colleagues Dr. Ehsan Kosrowshahi and Dr. Faisal Al-Saqqar for their help and support.

Finally, I am very grateful to my husband David Dery for his understanding, encouragement and endless support.

TABLE OF CONTENTS

LIST OF TABLES	vii
LIST OF FIGURES	viii
LIST OF ACRONYMS	x
1 Context and Problem of Research	1
1.1 Context	1
1.2 Motivation	4
1.3 Definition of the Problem and Methodology	5
1.4 Thesis Structure	7
2 Background	8
2.1 Web Services Composition	8
2.1.1 Service Oriented Architecture	9
2.1.2 Enterprise Processes	10
2.1.3 BPEL	12
2.2 Tools Used by BPEL2ISPL+	22
2.2.1 The DOM Parser	23
2.2.2 The DOT Drawing Tool	25
2.3 System Verification and Model Checking	28
2.3.1 Overview	28
2.3.2 Model Checking Multi-Agent Systems	32
2.3.3 Model Checking Communicative Commitments in MAS	33
2.3.4 The MCMAS+ Model Checker	37

3	Verifying Commitment-Driven Composite Web Services	41
3.1	Introduction	41
3.2	From BPEL to Agent Object Representation	42
3.2.1	Transformation of Partner Interactions Activities	47
3.2.2	Transformation of Basic Activities	48
3.2.3	Transformation of Structured Activities	49
3.2.4	Transforming Fault Handlers	62
3.2.5	Modeling Communication	62
3.3	From Agents Object Representation to ISPL and DOT	64
3.4	From Intermediate Representation to DOT	66
3.5	Verification	67
3.6	Case Study	67
3.6.1	Outline	67
3.6.2	Transformation	70
3.6.3	Verification Results	77
3.7	Related Work	78
4	Conclusion and Future Work	86
4.1	Conclusion	86
4.2	Future Work	87
	Bibliography	89
A	Extended Case Study	97
A.1	Outline	97
A.2	Transformation	99
A.3	Verification Results	105

LIST OF TABLES

2.1	Document interface methods to traverse a DOM tree	25
2.2	Document interface methods to inspect DOM nodes	26
2.3	NodeList object property and method	26
3.1	Transformation of partner interaction activities	48
3.2	Transformation of basic activities	48
3.3	Automata for communications actions	63

LIST OF FIGURES

1.1	W life cycle	2
1.2	Schematic view of our approach	6
2.1	Components of web service contract (from [17])	9
2.2	Example of service composition	10
2.3	Definition of a process service	11
2.4	BPEL2ISPL+ basic structure	22
2.5	DOM example	24
2.6	Output graph	27
2.7	Schematic view of system verification at the design phase	29
2.8	Schematic view of model checking	30
3.1	BPEL to ISPL+ transformation	43
3.2	Class diagram of agent object representation	44
3.3	Tree view of a BPEL file	45
3.4	ISPL agents depicted as automaton	46
3.5	Sequence example and its automaton	51
3.6	If example	54
3.7	While example	56
3.8	Pick example	58
3.9	Flow example	61
3.10	Implementation of shared variables	64
3.11	Sequence automaton	67
3.12	Timesheet submission service layers	68

3.13 Employee web service	69
3.14 Invoice web service	70
3.15 Timesheet submission service process	71
3.16 Invoice agent automaton	72
3.17 Employee agent automaton	73
3.18 TSP agent automaton	75
3.19 TSP agent automaton continued	76
3.20 Verification results	78
3.21 Witness of the formula one	79
3.22 Witness of the formula two	80
3.23 Witness of the formula three	81
3.24 Witness of the formula four	82
A.1 Timesheet submission service layers	98
A.2 Payroll web service	99
A.3 Project web service	100
A.4 Timesheet submission service process	101
A.5 Payroll agent automaton	102
A.6 Project agent automaton	103
A.7 TSP agent automaton	104
A.8 Verification results	106
A.9 Witness of the formula five	107
A.10 Witness of the formula six	108
A.11 Witness of the formula seven	109
A.12 Witness of the formula eight	110

LIST OF ACRONYMS

BPEL	Business Process Execution Language
ISPL	Interpreted Systems Programming Language
SOA	Service Oriented Architecture
CTL	Control Temporal Logic
CTLC	Control Temporal Logic of Social Commitments
XML	EXtensible Markup Language
DOM	Document Object Model
WSDL	Web Service Description Language
API	Application Programming Interface
HTML	HyperText Markup Language
CFSM	Communicating Finite State Machine
MAS	Multi-Agent System
LTL	Linear Temporal Logic

Chapter 1

Context and Problem of Research

1.1 Context

This thesis is about the verification of compositions of web services regulated by commitments abstracting contractual interactions. The verification is done using model checking, particularly the technique that was initially developed for Multi-Agent Systems (MAS). This section introduces the main concepts used in our approach which are: web service composition, commitments, verification techniques, and the MCMAS model checker.

Any application can publish its function on the Internet using web services. A web service presents its capabilities as operations, establishing interfaces without any ties to proprietary communication framework [18]. Through web service composition, new services can be developed by the means of coordinating the operations of existing web services, which may be running in different environments [17]. Business Process Execution Language (BPEL) is the OASIS standard language for web service composition [1]. It uses an XML-based vocabulary to specify business processes using a workflow. A web service composition uses interfaces and messages to coordinate operations. Interactions between

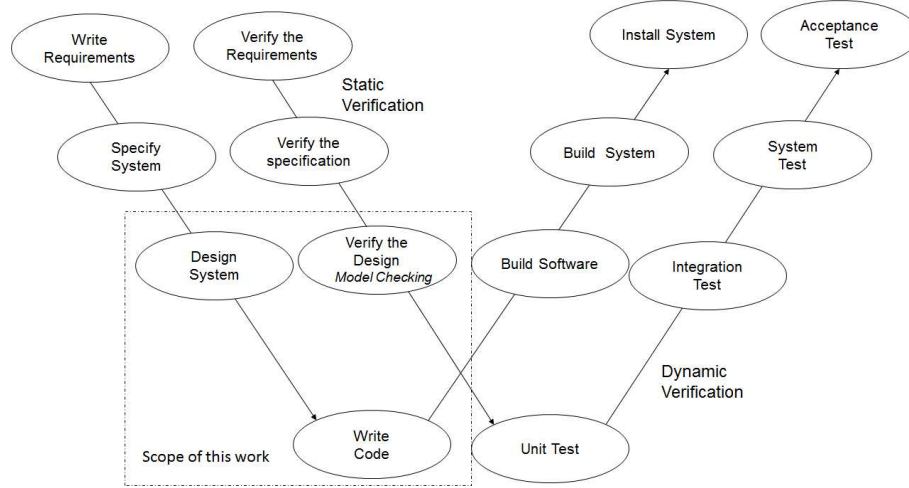


Figure 1.1: W life cycle

web services are modeled as commitments, which provide powerful abstractions of contractual obligations regulating such interactions. Verification of these interactions is a key issue to determine the quality of the composition.

Two types of verification techniques are usually used: dynamic and static. Testing is a dynamic technique deployed to verify the system execution. It runs a set of cases on the system comparing actual with expected outputs. Static techniques verify the system at early stages before producing the code. The W model [49, 30] depicted in Figure 1.1 distinguishes these two types of verification in a system development life cycle. Static verification techniques are performed in requirements, specification and design phases. While dynamic techniques can only be used after producing the system's code.

Model checking is a static verification technique used to verify the system at the design level [5]. It takes a system model as an input to characterize all possible scenarios over time and verify them against given properties. Properties capture functional and non-functional system requirements expressed in a temporal logic. Temporal logics are a special branch of symbolic logics focused on temporal propositions, i.e., statements to be evaluated to true or false depend on time. The model checking process is divided into the following

activities: 1) modeling the system under consideration; 2) formalizing the requirements into properties to be checked; and 3) running the model checker to verify the system model against the properties and analyzing the results. Model checking web service compositions has been broadly studied. However, the main properties that have been checked are limited to the safety and liveness [39]. A safety property asserts that "nothing bad will happen". A liveness property asserts that "something good will eventually happen".

On the other hand, model checking techniques have been used to verify MAS. A multi-agent system is comprised of several intelligent autonomous agents working together toward achieving a common goal or completing a joint task. An agent refers to a component of software that is capable of acting in a certain way to accomplish tasks on behalf of its user. Due to the autonomy of agents, two particular modalities have been extensively investigated: epistemic [19] and social commitments[45]. The epistemic modality deals with the knowledge that agents have and social commitments with what agents are publicly announcing.

The MCMAS model checker [32] has been developed to verify temporal and epistemic properties of MAS. MCMAS has its own language named Interpreted Systems Programming Language (ISPL) and is used to describe the system under verification. The model checker has been used to model and verify web services [33]. It was extended into MCMAS+ and ISPL+ [7, 28] to verify social commitment-based properties using CTLC, a dedicated logic extending Computation Tree Logic (CTL) with the commitment modality (C).

In this thesis, we propose to model and reason about compositions of web services using MAS as an abstraction tool when agent-based services can engage in business interactions using commitments. We assert that the interactions describing a web service composition can be verified through the verification of the commitments that model and

capture these interactions in the corresponding MAS model. To perform this verification, we transform the web service composition into a MAS model, where the participating services are transformed into agents. We model the behavior of the resulting MAS system using ISPL+. We define a set of properties using CTL_C to verify the composition through the verification of the establishment and fulfillment of commitments. The ISPL+ model is generated using BPEL2ISPL+, a tool we fully developed to automatically transform BPEL-based composition into ISPL+-based description. This tool takes as input the BPEL code of the process in charge of the composition and the behavioral description of the participating services, which are abstract BPEL processes, and produces as output the MAS model of the composition that can directly run on the MCMAS+ model checker.

1.2 Motivation

Web service composition allows companies to reuse existing applications and hence optimize their resource usage. One of the main concerns of service composition is to guarantee the quality of newly composite service. Model checking is extremely useful in this area as a formal and fully automatic technique to verify the system behavior at design phase, helping us detect defects and solve them using less resources than those that would be needed in subsequent phases [54]. This is particularly the case when the participating web services are supplied by different providers.

This work verifies the achievement of the composition's goals through the verification of the interactions among the participating services. Abstracting and transforming the problem of verifying service compositions to checking contractual interactions among services within a MAS setting provide a new and original way of applying model checking to verify services composition. A fully implemented transformation tool is provided to generate the MAS verifiable model. The main advantages of this approach are 1) to emphasize the

key elements of such a composition, namely the interaction and coordination among those services; 2) to capture the contractual obligations governing the composition through the means of commitments; and 3) to build on an existing model checker for MAS, MCMAS+, that has been proved to be efficient in terms of both time and scalability [28].

1.3 Definition of the Problem and Methodology

The problem we address in this thesis is about verifying composite web services described using the BPEL standard. The aim is to cover not only non-functional requirements, but also functional, business-oriented requirements. Our methodology is depicted in Figure 1.2. The inputs are a composition of web services shown as a BPEL-expressed workflow along with the BPEL code of the participating services. From the implementation perspective, web service composition is developed in BPEL Designer, an open source tool that enables us to specify the composition using a workflow that defines the interactions between the operations of the participant services. The BPEL code of the process in charge of the composition is automatically generated. The approach includes an automatic transformation of the composition workflow and the participating services into a verifiable MAS-based model expressed using ISPL+. This model is verified using the MCMAS+ model checker against given properties. Moreover, an automata view of the transformed model is also automatically generated.

Transforming BPEL processes into a verifiable model is a challenging task that has to be done systemically considering all the BPEL elements. Our BPEL2ISPL+ automated tool is able to perform this transformation using a set of algorithms transforming each BPEL construct into a corresponding ISPL+ code. BPEL2ISPL+ takes as input the BPEL process in charge of the composition and the behavioral description of the participating web services provided using abstract BPEL processes. In fact, the ISPL+ code defines a MAS

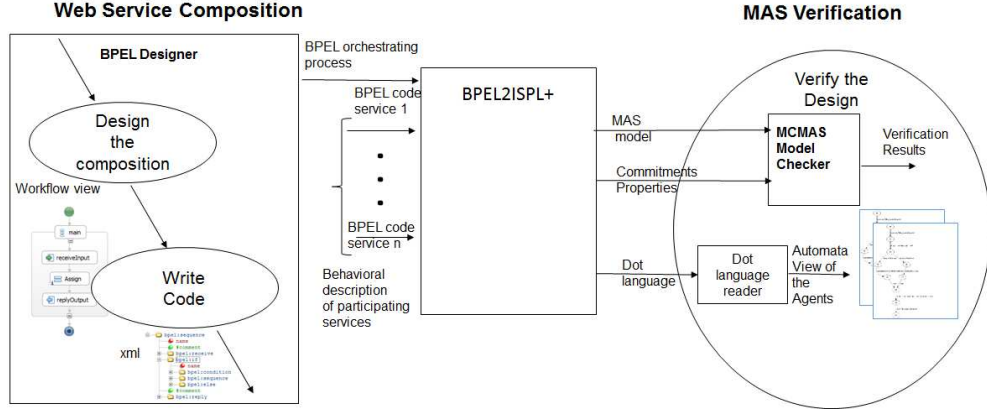


Figure 1.2: Schematic view of our approach

model which is then graphically displayed as an automaton using the dot language. The interactions among services are captured through the exchanged commitments.

To summarize, the key components of the proposed approach are:

- Transformation algorithms from BPEL constructs to MAS representations.
- A tool that generates a MAS description in ISPL+ taking as input the executable BPEL code of the process in charge of composition and the abstract BPEL processes of the participating services along with their behavioral description.
- A visual representation of the automatically generated MAS model as an additional output of the tool.
- A verification process of the web services composition through the verification of commitments between the agents representing the participating services and the agent representing the BPEL process in charge of the composition using MCMAS+ and CTLIC properties.

1.4 Thesis Structure

The rest of the thesis is organized as follows. In Chapter 2, we present a brief review of the background concepts including: Service-Oriented Architecture (SOA), web services, BPEL, verification, model checking, communicative commitments and the MC-MAS+ model checker. Chapter 3 discusses our approach in detail and provides a case study along with experimental results. In Chapter 4, we conclude our work and describe opportunities for future work.

Chapter 2

Background

The three main components of this work are, as depicted in Figure 1.2: the system under study, its transformation and its verification. The system under study is a web services composition, the transformation takes this composition to produce a verifiable model. Then, this model is used to verify commitments between agents representing web services.

This chapter devotes one section to offer background knowledge on each one of these three components. Section 2.1 introduces Web services composition. Then, Section 2.2 introduces the tools used on the transformation. Finally, Section 2.3 explains the verification technology used on this work.

2.1 Web Services Composition

This section describes web services composition in the context of Service Oriented Architecture. Then, we introduce BPEL, which is the de facto standard to develop these compositions, and the one we used to develop our case study.

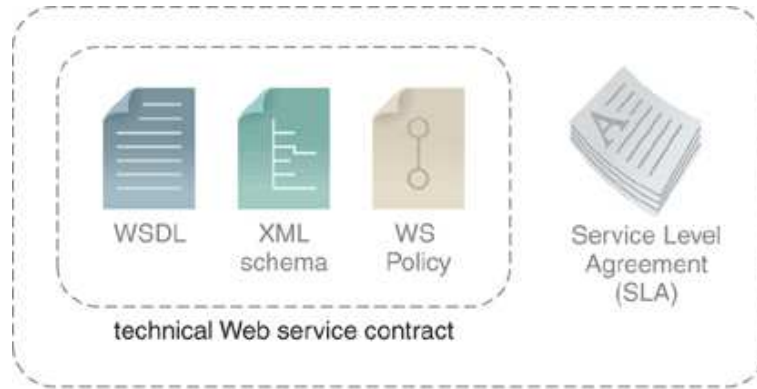


Figure 2.1: Components of web service contract (from [17])

2.1.1 Service Oriented Architecture

Service-oriented architecture is a model that describes how services could be used through description, registration and invocation. Its main objective is to enhance the agility and cost-effectiveness of business applications. It accomplishes this by making services as the main means through which solution logic is implemented [46, 47]. A service is a unit of solution logic. Each service has a functional goal, and is comprised of a set of capabilities and operations. Therefore, a service can be considered as a container of operations associated with a given goal. These operations are expressed in the service contract [17].

A web service is a piece of software describing a set of functionalities offered as a service by a machine (or electronic device) to another machine, communicating with each other via the World Wide Web. A web service exposes public capabilities as operations, establishing a technical interface without being associated to any proprietary communication framework. Each web service is described as a contract using Web Service Description Language (WSDL) definition, XML schema definition, and WS-Policy definition. A web service contract can be further comprised of human-readable documents, such as a Service Level Agreement (SLA) that describes non-functional requirements as quality features, behaviors and limitations. Figure 2.1 illustrates the components of a web service contract.

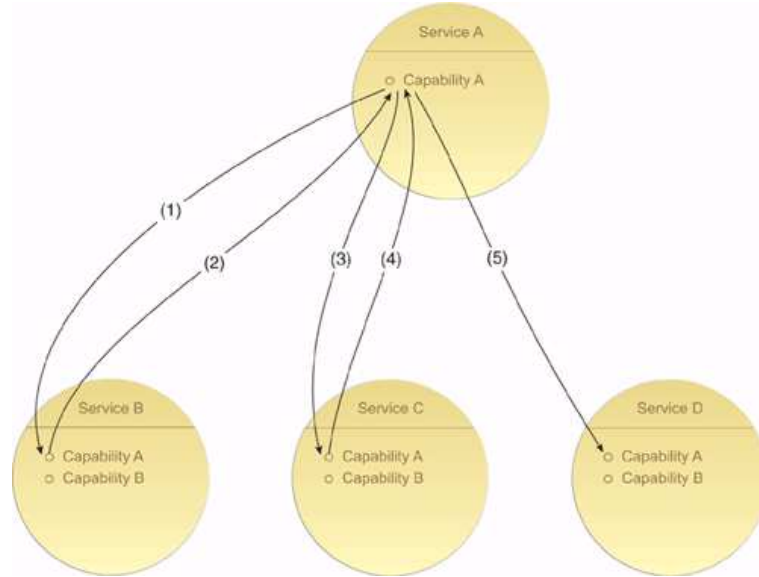


Figure 2.2: Example of service composition

Web services are designed to communicate and coordinate with each other within compositions. A service composition is an aggregation of services collectively composed to automate a particular complex function or business process that are beyond the capabilities of individual services. To qualify as a composition, two participating services or more, in addition to the composition initiator, need to take place and to be clearly specified. Otherwise, the service interaction only represents a point-to-point exchange [17]. Figure 2.2 depicts a schematic view of an example of web services composition.

2.1.2 Enterprise Processes

Enterprise objectives are implemented through processes. Both objectives and processes are constantly changing in response to external and internal influences. Nowadays, IT plays a key role in specifying, supporting and facilitating these processes. From an IT perspective, as argued in [47], enterprise processes can be divided into two main classes: business processes and applications. Usually, business processes are documented implementations of the business requirements. Business processes express these requirements, along with any

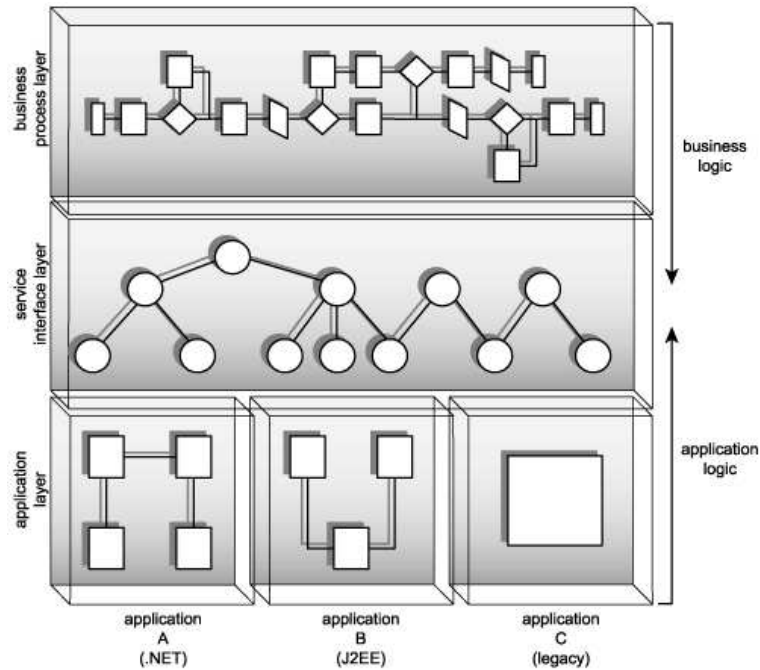


Figure 2.3: Definition of a process service

associated constraints, dependencies, and outside influences [17]. Applications are technology solutions implementing a part of these processes. These applications support business processes through developed systems within an organization's IT infrastructure covering security constraints, technical capabilities, and vendor dependencies.

In this enterprise context, web services have been highly suitable and successful in implementing and realizing SOA [47]. This is mainly thanks to the fact that services establish a form of abstraction located between traditional business and application layers. Services can encapsulate physical application logic as well as business process logic, as depicted in Figure 2.3 [17].

Designing an IT solution requires a proper interpretation of the business process requirements, that should be collected at the beginning of the endeavor, and then implemented accurately. Usually, business processes are designed by analysts using modeling tools that

produce diagrams. These diagrams are handed over to architects and developers for implementation purposes. Diagrams and its accompanying documentation are, most of the times, the sole means of communicating how this logic should be realized within an automated solution. This traditional approaches is the source of misunderstandings and omissions between the analysis and implementation phases [46].

In SOA, this misalignment between the analysis and implementation phases is addressed by operational business modeling languages, such as BPEL. BPEL composes web services using a workflow approach and an XML-based description. The user can develop her code in two ways: writing it directly or using a visual editor. Writing the code directly requires a deep knowledge of the language, while using an editor allows the analyst to construct the BPEL processes using a drag-and-drop approach. The result is a diagram on the front end that expresses the analyst's vision of the process and a computer executable process definition on the back end that can be handed over to the development team. Although architects and developers may complete the process with technical information, there is almost no room for interpretation of the implementation.

2.1.3 BPEL

BPEL is a programming *in the large* language used for web services composition. It differs from programming *in the small* languages (such as Java, or C) which are used to implement specific functionalities. Web services described in BPEL may include several functionalities to perform the associated operations. BPEL is widely supported in development environments, such as JDeveloper, WebSphere Integration Designer, and Eclipse. To execute a BPEL process, we need a process server. Several commercial and open source BPEL process servers are available. The most well-known are Oracle SOA Suite, IBM WebSphere BPM, ActiveVOS, and Apache ODE.

A. BPEL Activities

A BPEL process consists of steps, each step is called an activity. BPEL supports three types of activities: partner interaction, basic, and structured [1, 27]. Partner interaction activities are used for receiving and sending messages to and from participating partners and are listed as follows:

- Invoking other web services using `<invoke>`.
- Waiting for the client to invoke the business process through sending a message using `<receive>` (receiving a request).
- Generating a response for synchronous operations using `<reply>`.

Basic activities are used for common tasks and are listed as follows:

- Manipulating data variables using `<assign>`.
- Indicating faults and exceptions using `<throw>` and `<rethrow>`.
- Waiting for some time using `<wait>`.
- Terminating the entire process using `<exit>`.

Structured activities aggregate basic activities to define composite flows. The most important are listed as follows:

- Sequence (`<sequence>`) for a set of activities that will be invoked in an ordered sequence.
- Flow (`<flow>`) for defining a set of activities that will be invoked in parallel.
- Conditional construct (`<if>`) for implementing branches.

- While, repeat, and for each (<while>, <repeatUntil>, <forEach>) for defining loops.
- Pick (<pick>) for selecting one of the number of alternative paths.

B. BPEL Structure

A BPEL process definition is written as an XML document using the <process> root element. <partnerLinks>, <variables> and a top-level <sequence> activity are usually part of the root element. Within the sequence, the process will first wait for the incoming message to start the process. This wait is modeled with the <receive> construct. Then, the process will perform some activities and return a response. The following listing shows an example [1]:

Listing 2.1: BPEL example

```
<process ...>
  <partnerLinks>
    <partnerLink ... />
    ...
    <partnerLink ... />
  </partnerLinks>
  <variables>
    <variable name= inputVariable... />
    ...
    <variable name = outputVariable... />
  </variables>
  <sequence>

    <!-- Wait for the incoming request to start the process -->
    <receive ... />
```



```

    <!-- Perform some activities -->
    ...

    <!-- Return the response -->
    <reply ... />
  </sequence>
</process>

```

Partner Links

Partner links denote all the interactions with the external participating services. The two possibilities of interaction are:

- The BPEL process invokes operations on other services.
- The BPEL process receives invocations from clients.

Each BPEL process has at least one client partner link that invokes the BPEL process and initiates its execution. Usually, a BPEL process will also have several invoked partner links because it will most likely invoke several services. Partner links are specified near the beginning of the BPEL process definition document, just after the `<process>` tag. Several `<partnerLink>` definitions are nested within the `<partnerLinks>` as shown in the previous listing. For each partner link, the following items should be specified:

- `name`: serves as a reference for interactions via that partner link.
- `partnerLinkType`: defines the type of the partner link. Partner link types are defined in the WSDL document.
- `myRole`: indicates the role of the BPEL process itself.
- `partnerRole`: specifies the role of the partner.

Variables

As any other programming language, BPEL does not make exception in using variables, but makes exception in the fact that these variables can hold XML elements and not only primitive types. A program requires declaration and initialization of variables, and needs at least two of them: input and output. The input variable holds the input payload (input parameters). The output variable holds the output, which is returned to the client (the one that invoked the BPEL process).

The Process Logic

The process logic is specified in the top-level `<sequence>` activity, which contains all the process flow. Each process starts waiting for the initial request message from the client, using a (`<receive>`) activity. Then uses several BPEL activities to process the received input. Most of the times, a process ends with a `<reply>` activity to return a response to the client [1].

C. Partner Interactions

Synchronous and asynchronous communication are the two general strategies to interact with web services. In synchronous interactions, both parties maintain the connection intact until the communication is over. In contrast, in asynchronous interactions, the client establishes a connection with the server, then sends the request message and closes the connection. The receiver processes the incoming message to generate a response message. Then, it establishes a connection with the requesting-party, and sends the response [27].

Each partner interaction activity identifies the interacting partners, specifying the following information:

- `partnerLink`: specifies which partner link is interacting
- `portType`: specifies the port type being used
- `operation`: specifies the name of the operation is working with

Three activities are associated with each partner interaction: receive, reply, and invoke.

Receive

This activity waits for an incoming message (operation invocation), either for starting the BPEL process, or for a callback function (after an asynchronous invocation). The business process stores the incoming message in the variable specified in the *variable* attribute.

Another attribute for the receive activity is the *createInstance* attribute, which is related to the business process life cycle, and instructs the BPEL engine to create a new instance of the process. We specify the *createInstance* = "yes" attribute with the initial receive activity of the process.

Reply

This activity is used to return the response for the synchronous BPEL operation. It is always related to the initial receive through which the BPEL process started. We define the name of the variable where the response message is stored using the *variable* attribute.

Invoke

This activity invokes operations on other services. When the business process invokes an operation on a service, it sends a set of parameters that are modeled as input messages. To specify the input message for the invocation, we use the *inputVariable* attribute. If the invoke is a synchronous request/response operation it returns a result, which is also modeled as an output message. We use the variable defined in the *outputVariable* attribute to specify the output message.

D. Faults

BPEL processes execution could lead to unexpected behaviors. These are due to communication issues, contract issues, faults produced by an external web service, or faults from the business process [27].

Communication Issues

When a business process communicates with external web services, this communication uses a network. The communication is prone to unexpected errors due to unreliability on the infrastructure. Another issue arises when the web service is unavailable, or when it has been moved to a new location without notifying the business process.

Contract Issues

The agreed communication contracts with external web services can be changed without notification, leading to the breakage of the agreed relationship.

Faults thrown from the external web service

The external web service itself can throw errors based on its execution. And those faults can be propagated back to the business process.

Faults thrown from the business process

The business process can generate faults due to its business logic. These faults can be divided in:

- Logical errors: These faults are defined by the business process developer. So, for example, if the input variable carries an unexpected value, the business process developer can declare a fault within the business logic. And then the business process should be responsible to take care of that fault. Logical errors are also known as business faults.
- Execution errors: This category of faults is generated by the BPEL runtime. Suppose, when a variable is assigned some data, the BPEL run-time generates a fault if the variable is uninitialized.

E. Fault Handlers

BPEL specification uses fault handlers to deal with faults. These faults can be either implicitly generated by the BPEL run-time or explicitly generated using the `< throw >` activity.

The following code template depicts the structure of fault handlers [27]:

Listing 2.2: Fault handlers example

```
<faultHandlers >
    <catch faultName = ... ><!-- First fault handler -->
        <!-- Perform an activity -->
    </catch>
    <catch faultName ... ><!-- Second fault handler -->
        <!-- Perform an activity -->
    </catch>
    <catchAll>
        <!-- Perform an activity -->
    </catchAll>
</faultHandlers >
```

From the previous listing, we can observe that the child elements of a fault handler are either `< catch>` or `< catchAll>`. A `< catch>` element can be used to handle a specific fault and `< catchAll>` is used to handle faults that are not handled by the other `< catch>` elements. The optional `< catchAll>` element should be located as the last element in `< faultHandler>`.

F. Abstract Business Processes

Additionally to executable business processes, BPEL supports abstract business processes. These are partially specified processes that are not intended to be executed. The syntactic characteristics of abstract processes are [48]:

- The `abstractProcessProfile` attribute must exist. Its value refers to an existing profile definition.
- All the constructs of Executable Processes are permitted. Therefore, there is no fundamental expressive power distinction between Abstract and Executable Processes.

- Certain syntactic constructs in BPEL Executable Processes may be hidden, explicitly through the inclusion of opaque language extensions, and implicitly through omission.
 - Opaque activities are allowed.
 - All BPEL expressions are allowed to be opaque.
 - All BPEL attributes are allowed to be opaque.
 - The from-spec (e.g. in (< assign>)) is allowed to be opaque.
- An Abstract Process may omit the `createInstance` activity that is mandatory for Executable BPEL Processes.

Examples of opaque language extensions are shown in the following listing. Partner links definitions contain opaque attributes. The condition within the < if > activity is opaque. The from-spec within the < assign > in the true branch of the < if > is opaque too. An opaque activity follows the < if > activity.

Listing 2.3: Example of opaque attributes [48]

```
<process name="OrderingServiceProcess"
...
  abstractProcessProfile="http://docs.oasis-open.org/wsbpel..." >
<partnerLinks>
  ...
  <partnerLink name="invoiceProcessor"
    partnerLinkType="##opaque"
    myRole="##opaque"
    partnerRole="##opaque" />
</partnerLinks>
<variables>
```

```

    ...
</variables>
<sequence>
    ...
    <if>
        <condition opaque="yes" />
        <assign>
            <copy>
                <opaqueFrom/>
                <to>orderAckMsg . OrderAckMessagePart / order : Ack</to>
            </copy>
        </assign>
        <else>
            ...
        </else>
    </if>
    <opaqueActivity>
        <documentation>
            If we receive notice that the ship has completed , update
            our ship history accordingly
        </documentation>
    </opaqueActivity>
    ...
</sequence>
</process>

```

The BPEL specification defines abstract business processes either as templates or as means to describe the externally observable service behavior [26]. Abstract processes as templates can be used to define sets of rules, without including all the execution details. Such details can be added later when the abstract process is used as a template for developing an

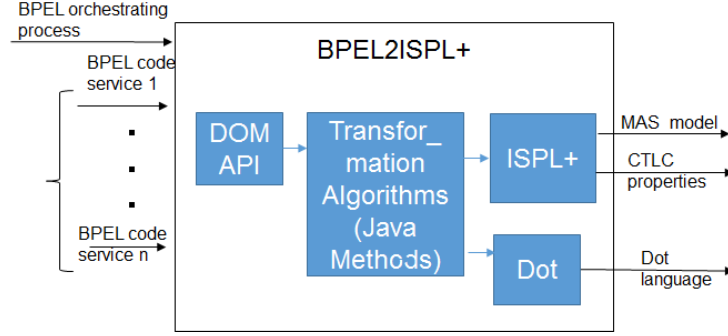


Figure 2.4: BPEL2ISPL+ basic structure

executable process. Abstract processes for describing the externally observable service behavior is useful for describing public process behavior without the exact details of how the process executes. Such abstract business processes specify public message exchange between parties only. An abstract business process should provide a complete description of external behavior relevant to a partner or several partners it interacts with. In this case, the use of opacity is concentrated in those features associated with data handling.

2.2 Tools Used by BPEL2ISPL+

BPEL2ISPL+ is a Java program that performs the transformation from BPEL into a verifiable MAS model. This section introduces the tools used by BPEL2ISPL+ to achieve its objective. As illustrated in Figure 2.4, BPEL2ISPL+ takes several BPEL files as input. Then, BPEL files are read using the DOM parser, while translating them into an Agent Object Representation (AOR) following a set of algorithms. Out of the the object representation, two outputs are produced. The main output is the verifiable model in ISPL+, the second output is a file in the DOT language. The latter one allows us to produce a graphical view of the main components of the model. Both outputs are text files.

In fact, BPEL2ISPL+ uses: BPEL, the DOM parser, Java classes to build the AOR, ISPL+ and the DOT language. BPEL was already introduced in the previous section, Java is

a well known language and the verifiable model in ISPL+ [28] is discussed in the next section. In this section, we introduce the DOM parser (Section 2.2.1), and the DOT language (Section 2.2.2).

2.2.1 The DOM Parser

BPEL2ISPL+ uses an XML parser to read the input BPEL files. The two most popular APIs used to parse XML documents are: the Document Object Model (DOM) and the Simple API for XML (SAX). DOM is an official recommendation of the W3C (available at <http://www.w3.org/TR/REC-DOM-Level-1>), while SAX is a de facto standard created on the XML-DEV mailing list (<http://lists.xml.org/archives>) [50].

SAX uses memory efficiently, but it reads documents sequentially. DOM uses more memory than SAX, but allows us to reach elements randomly. Additionally, it facilitates access to the siblings of an element. We decided to use DOM because we needed random access to the document's elements, and did not have important memory restrictions. DOM builds a tree view of the XML document that is contained in a single element, which becomes the root of the tree. The DOM specification defines several language-neutral interfaces including:

- **Node**. This interface is the base datatype of the DOM. `Document`, `Element`, `Attr`, `Text`, `Comment`, and `ProcessingInstruction` all extend the `Node` interface.
- **Document**. This object contains the DOM representation of the XML document.
- **Element**. This interface represents an element in an XML document.
- **Attr**. This interface represents an attribute of an element in an XML document.
- **Text**. This interface represents a piece of text from the XML document. Any text in your XML document becomes a `Text` node. This means that the text of a DOM object

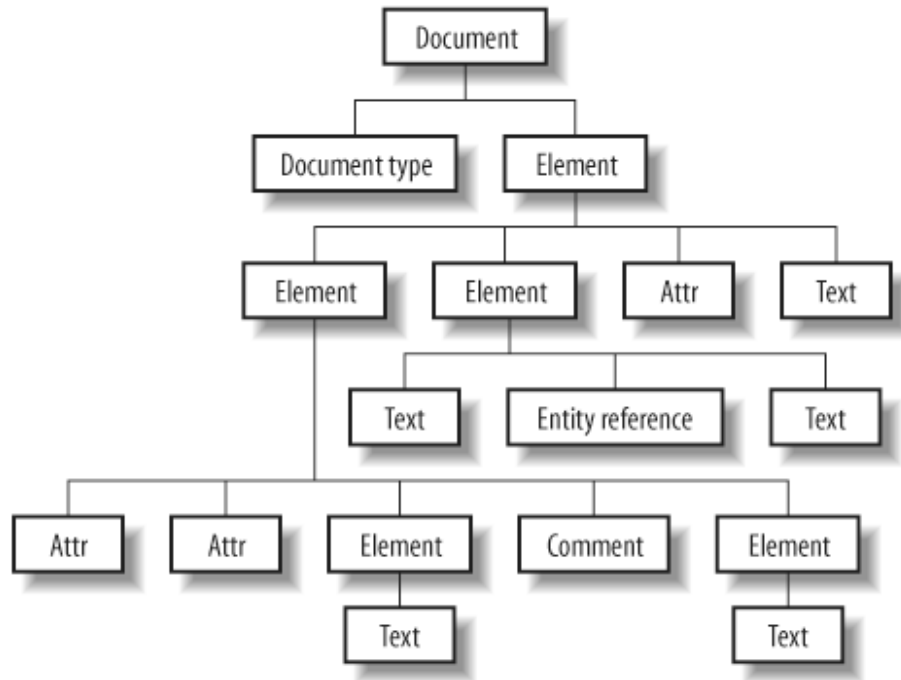


Figure 2.5: DOM example

is a child of the object, not a property of it. The text of an `Element` is represented as a `Text` child of an `Element` object; the text of an `Attr` is also represented that way.

- `Comment`. This interface represents a comment in the XML document.
- `ProcessingInstruction`. This interface represents a processing instruction in the XML document.

After parsing an XML document, the DOM parser builds an in-memory representation of the document as represented in Figure 2.5 [36]. Given a `Document` object, one can get the root of the tree with the `getDocumentElement()` method. From the root, we can move through the tree to find all elements, attributes, text, comments, processing instructions, etc.

There are a number of methods provided in the `Document` interface to access the nodes in the tree, some of them are listed in Table 2.1. These methods return either a `Node`

Method name	Description
<code>getDocumentElement()</code>	Allows direct access to the root element of the Document.
<code>getElementsByTagName(String)</code>	Returns a <code>NodeList</code> of all the elements with the given tag name, in the order in which they are encountered in the tree.
<code>getChildNodes()</code>	Returns a <code>NodeList</code> that contains all children of this Node.
<code>getParentNode()</code>	Returns the parent of this Node.
<code>getFirstChild()</code>	Returns the first child of this Node.
<code>getLastChild()</code>	Returns the last child of this Node.
<code>getPreviousSibling()</code>	Returns the Node immediately preceding this Node.

Table 2.1: Document interface methods to traverse a DOM tree

or a `NodeList` (ordered collection of nodes) [51].

In a simple DOM application, the `getChildNodes()` method can be used to recursively traverse the DOM tree. The `NodeList.getLength()` method can then be used to find out the number of nodes in the `NodeList`. In addition to the tree traversal methods, the Node interface provides the methods shown in Table 2.2 to investigate the contents of a node. `NodeList` objects are collections of Nodes such as those returned by `Node.childNodes()`. They have one property and one method (see Table 2.3).

2.2.2 The DOT Drawing Tool

A visual representation of the agents defined in ISPL+ language [28] facilitates the comprehension and review of the verifiable model. The verification tool MCMAS+ [7, 28] used in this thesis (see Section 2.3.4) doesn't include a visual representation. Thus, we used the Graphviz graph visualization software to implement it [41]. Graphviz is an open source software that has several programs to provide different type of layouts (hierarchical, radial, circular, etc). Graphviz's programs take descriptions of graphs in a simple text language, and make diagrams in useful formats for inclusion in other documents or display in an

Method name	Description
<code>getAttributes()</code>	Returns a <code>NamedNodeMap</code> containing the attributes of a <code>Node</code> if it is an <code>Element</code> or null if it is not.
<code>getNodeName()</code>	Returns a string representing the name of this <code>Node</code> (the tag).
<code>getNodeType()</code>	Returns a code representing the type of the underlying object.
<code>getNodeValue()</code>	Returns a string representing the value of this <code>Node</code> . If the <code>Node</code> is a <code>Text</code> node, the value will be the contents of the <code>Text</code> node. For an attribute <code>Node</code> , it will be the string assigned to the attribute. For most node types, there is no value and a call to this method will return null.
<code>getNamespaceURI()</code>	The namespace URI of this <code>Node</code> .
<code>hasAttributes()</code>	Returns a boolean to indicate whether this <code>Node</code> has any attributes.
<code>hasChildNodes()</code>	Returns a boolean to indicate whether this <code>Node</code> has any children.

Table 2.2: Document interface methods to inspect DOM nodes

Property/Method	Description
<code>length()</code>	Returns the number of nodes in a <code>NodeList</code> .
<code>item</code>	Returns the <code>Node</code> at the specified index in a <code>NodeList</code> .

Table 2.3: `NodeList` object property and method

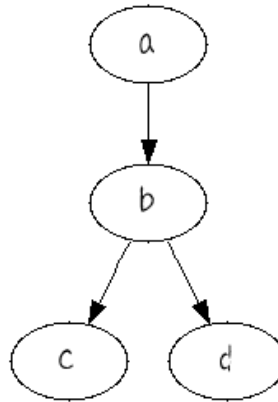


Figure 2.6: Output graph

interactive graph browser [41].

The DOT program accepts inputs in the DOT language and produces directed graphs. This language describes three types of objects: graphs, nodes, and edges. In the following listing, we give an example. The first line defines the type of graph and its name. The second and third line define nodes and edges. DOT creates a node when its name first appears. DOT creates Edges when nodes are joined by the edge operator `->`.

Listing 2.4: DOT program example [48]

```
digraph graphname {  
    a -> b -> c;  
    b -> d;  
}
```

DOT files are read by several tools that complement Graphviz, such as graph generators, post processors and interactive viewers. The output provided by one of this tools is depicted in Figure 2.6.

2.3 System Verification and Model Checking

This section starts with an overview of verification and model checking in Section 2.3.1, followed by an introduction to its application to MAS in Section 2.3.2. Then, we present the logic developed to verify commitments in MAS in Section 2.3.3, and finally we describe the MCMAS+ model checker in Section 2.3.4.

2.3.1 Overview

System verification is the process of checking that a system meets its specification, and that it fulfills its intended purpose. This specification prescribes what the system has to do and what not, constituting the basis for any verification activity. A defect is found when the system does not fulfill one of the specification's properties. The system is "correct" when it satisfies all properties obtained from its specification. So correctness is always relative to a specification, and is not an absolute property of a system.

Verification techniques are divided into two types: dynamic and static. Testing is a dynamic technique that is used to verify the system. It runs a set of cases on the system comparing actual with expected outputs. Static techniques verifies the systems at early stages before producing code. The W model [49] in Figure 1.1 distinguishes these two types of verification in a system development life cycle. This model has two Vs, one dedicated to the system development and another devoted to its verification. Static verification techniques are performed in requirements, specification and design phases. While dynamic techniques can only be used after producing the system's code. The coverage of testing increases along with the code is produced. It goes from unit test, to integration test, to system test and finalizes with acceptance test. A schematic view of verification at the design phase is depicted in Figure 2.7 [5].

Model checking is a static technique used at the design phase to verify automatically

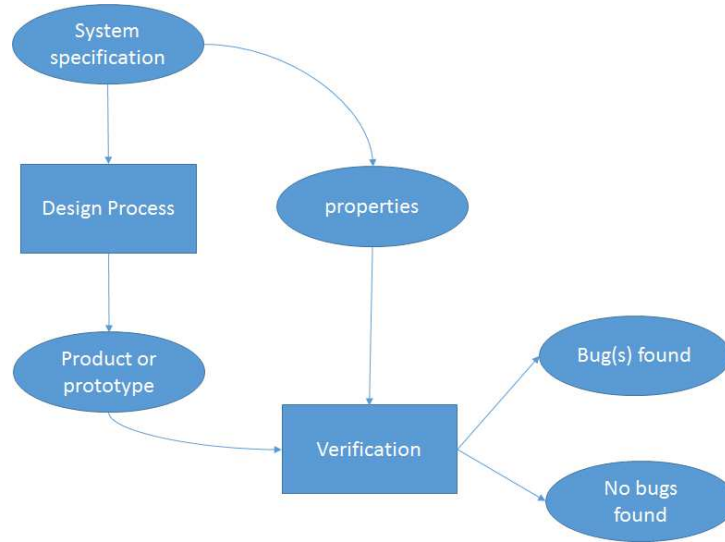


Figure 2.7: Schematic view of system verification at the design phase

finite state systems. The verification involves an exhaustive search of the state space of the design, to check if a given property is satisfied or not. The procedure is guaranteed to terminate with a yes/no answer, if sufficient computational resources are available. In order to apply model checking to a given system, it needs to be expressed in a formalism susceptible to model checking. Then, it is necessary to state the requirements that the system must satisfy. These requirements are typically expressed as a set of properties in a suitable logical formalism [35]. A schematic view on model checking is depicted in Figure 2.8 [5].

Each model checker has a model description language to specify the system, and a property specification language to formalize its requirements. In applying model checking to a design, the following different steps must be followed:

- Modeling the system under consideration, using the model description language of the selected model checker. Simulations should be run to verify that the model represents accurately the system's design.
- Formalizing the properties to be checked, using the property specification language.

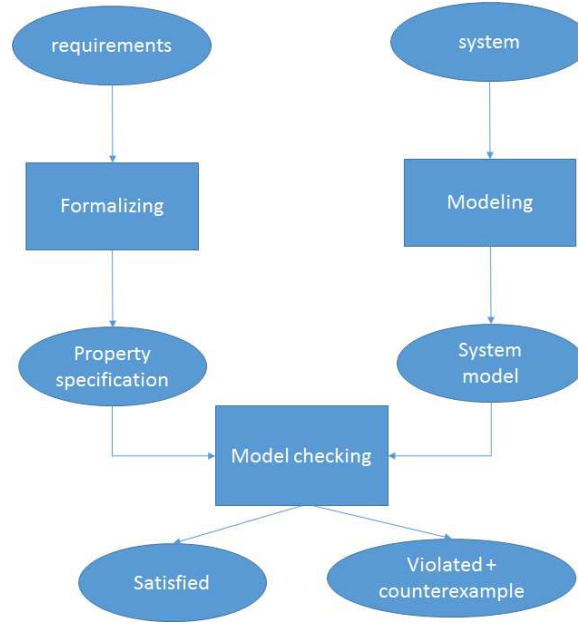


Figure 2.8: Schematic view of model checking

- Running the model checker, to check the validity of the property in the system model.
- Analyzing the results. If the property is satisfied, then check the next property. If the property is violated, then analyze the counterexample. Using analysis results, remove the defect either by refining the design, the model or even the property. Run the verification again after performing the modifications.

A. System Models

These models describe the behavior of systems in an accurate and unambiguous way. They are mostly expressed using finite state automata, consisting of a finite set of states and a set of transitions. States comprise information about the current values of variables. Transitions describe how the system evolves from one state into another [5].

One of the most popular formalisms to model systems in model checking are Kripke structures defined formally as follows: Let AP be a set of atomic propositions. A Kripke structure over AP is a triple $M = (S, R, K)$, where:

- S is a set of states.
- $R \subseteq S \times S$ is a *transition relation* that is total, that is, $\forall s \in S (\exists t \in S) ((s, t) \in R)$.
- $K : S \rightarrow 2^A P$ is a *labelling function*.

A Kripke structure models the state transition graph, where outputs are functions of current state variables. The labeling function K associates each state with a set of atomic propositions that are true in that state.

B. Properties

The target of model checking are mostly dynamic systems. Dynamic systems have a state component that changes over time. Temporal logics are a suitable formalism for describing requirements or properties of such systems for model checking. Temporal logic expresses system behavior over time without explicitly bringing in the notion of time [35].

Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) are among the most popular temporal logics in model checking. These logics allow us the specification of a broad range of relevant system properties, such as: functional correctness (does the system do what it is supposed to do?), reachability (is it possible to end up in a deadlock state?), safety ("something bad never happens"), liveness ("something good will eventually happen"), fairness (does, under certain conditions, an event occur repeatedly?), and real-time properties (is the system acting in time?) [5].

Computation Tree Logic

CTL is a branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined. Different paths in the future are available, any one of which might be the "actual" path.

We define CTL formulas inductively via a Backus Naur form:

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid EX\varphi \mid E(\varphi_1 U \varphi_2) \mid EG\varphi$$

where $p \in AP$ is an atomic proposition.

CTL distinguishes between state and path formulae. Intuitively, state formulae express a property of a state, while path formulae express a property of a path. Formula $X\phi$ holds for a path if ϕ holds at the next state in the path, $\phi_1 U \phi_2$ holds for a path if there is some state along the path for which ϕ_2 holds, and ϕ_1 holds in all states prior to that state, and $G\phi$ holds for a path if ϕ holds in all states. Path formulae can be turned into state formulae by prefixing E (pronounced "for some path"). Note that the linear temporal operators X , U and G are required to be immediately preceded by E to obtain a legal state formula. Other operators may be defined as the path quantifier A (pronounced "for all paths").

2.3.2 Model Checking Multi-Agent Systems

Different approaches have been proposed for conducting verification in multi-agent systems using model checking [8, 9, 13, 29, 37]. These approaches include the verification of temporal [53], epistemic (knowledge) [3, 40] and commitments properties [3, 7]. This thesis focuses on CTLC, a logic of social commitments that extends CTL to make the reasoning about the commitments establishment between two agents possible [7].

Before describing CTLC, we provide an introduction to MAS in order to understand the context under which this logic was developed. An agent refers to a component of software and/or hardware that is capable of acting in a certain way to accomplish tasks on behalf of its user. Agents generally look for complete automation of complex processes through artificial intelligent techniques acting on behalf of human users. An agent is a computational entity that satisfies the following criteria [2]:

- Acts on behalf of other entities in an autonomous fashion.
- Performs its actions with some level of proactivity and/or reactivity.

- Exhibits several characteristics such as: autonomy, cooperation, learning and reactivity.
- Enjoys social ability, which refers to the ability to interface and interact with other agents via a communication language.

A multi-agent system is comprised of several intelligent agents working together toward a goal or completion of a task. This system is called for when complex problems require the coordination of multiple agents with diverse capabilities and needs. A multi-agent system cannot exist without interactions between intelligent agents. In cooperative models, several agents try to combine their efforts to accomplish as a group what the individuals cannot, while in competitive models, each agent tries to get what only some of them can have [2].

MAS objectives in cooperative models are achieved through collaboration. Collaboration is realized when an agent commits to perform a task, and fulfills its commitment. Castelfranchi et al. stated that "Social commitment results from the merging of a strong delegation and the corresponding strong adoption: reciprocal social commitments constitute the most important structure of groups and organizations" [10].

2.3.3 Model Checking Communicative Commitments in MAS

The model used for this logic extends the interpreted system formalism to account for communication that occurs during the execution of MAS. This extension provides an intuitive semantics for social commitments that are established through communication between interacting agents [7].

A. Interpreted Systems

Interpreted systems was first adopted as a formal tool to model MAS characteristics by Fagin et al. [19]. The system is defined by:

- A set of n agents $A = \{1, \dots, n\}$, each agent $i \in A$ has:
 - a non-empty and countable set L_i of local states. Each local state of an agent represents the complete information about the system that the agent has at his disposal at a given moment;
 - a set Act_i of local actions. It is assumed that $null \in Act_i$ for each agent i , where $null$ refers to the silence action;
 - an action selection mechanism given by the notion of local protocol $P_i : L_i \rightarrow 2^{Act_i}$. That is P_i is a function giving the set of enabled actions that may be performed by i , in a given local state;
 - an evolution function τ_i that determines the transitions for an individual agent i between its local states and is defined as follows: $\tau_i : L_i \times Act_i \rightarrow L_i$.
- A set of global states is denoted by S , a global state $s \in S$:
 - gives the configuration of all agents in the system at a given time;
 - is a tuple $g = (l_1, \dots, l_n)$ where each element $l_i \in L_i$ represents a local state of agent i . Thus, the set of all global states $S \subseteq L_1 \times \dots \times L_n$ is a non-empty subset of the cartesian product of all local states of n agents.
- A set of initial global states for the system $I \subseteq S$.
- A global evolution defined as follows: $\tau : \times ACT \rightarrow S$, where $ACT = Act_1 \times \dots \times Act_n$ is a joint action, which is a tuple of actions (one of each agent).
- A set Φ_p of atomic propositions and a valuation function $v : S \rightarrow 2^{\Phi_p}$

To provide an "intuitive semantics" for social commitments that are established through communication between inter-acting agents, Bentahar et al. [7] extended the formalism of interpreted systems as follows:

- Each agent $i \in A$ is associated with a set of local variables Var_i of n local boolean variables. These variables represent the communication channels that can be used for sending and receiving messages.
- A shared variable between two interacting agents i and j exists iff $Var_i \cap Var_j \neq \emptyset$, which means the existence of a communication channel between the two agents.
- The value of a variable x in the set Var_i at local state $l_i(s)$ is denoted by $l_i^x(s)$.
- For the shared variable $x \in Var_i \cap Var_j$, $l_i^x(s) = l_j^x(s')$ means that the values of variable x for the agent i in $l_i^x(s)$ are equal to the values of variable x for agent j in $l_j^x(s')$
- If $l_i(s) = l_i(s')$ then $l_i^x(s) = l_i^x(s')$ for all $x \in Var_i$.
- For the unshared variables $(y), \forall y \in Var_j - Var_i$ we have $l_i^y(s) = l_i^y(s')$

Therefore, a model $M_C = (S, I, R_t, \{\sim_{i \rightarrow j}\}_{(i,j) \in A^2}, V)$ is a tuple, where:

- $S \subseteq L_1 \times \dots \times L_n$ is a set of reachable global states for the system.
- $I \subseteq S$ is a set of initial global states for the system.
- $R_t \subseteq S \times S$ is the transition relation defined by $(s, s') \in R_t$ iff there exists a joining action $(a_1, \dots, a_n) \in ACT$ such that $\tau(s, a_1, \dots, a_n) = s'$
- For each pair $(i, j) \in A^2$, $\sim_{i \rightarrow j} \subseteq S \times S$ is the social accessibility relation defined by $s \sim_{i \rightarrow j} s'$ iff $l_i(s) = l_i(s')$ and $\exists! x \in Var_i \cap Var_j$ s.t. $l_i^x(s) = l_j^x(s')$ and $\forall y \in Var_j - Var_i$ we have $l_j^y(s) = l_j^y(s')$ and s' is reachable from s using transitions from the transition relation R_t .
- $v : S \rightarrow 2^{\Phi_p}$ is a valuation function, where Φ_p is the set of atomic propositions.

B. Computation Tree Logic of Commitments

The CTLC logic extends CTL with two operators: commitment and fulfillment. A commitment $C_{i \rightarrow j} \varphi$ is described by a debtor i , a creditor j and the commitment content φ , meaning that the debtor commits to the creditor to bring about the content φ . The fulfillment of this commitment by the debtor is denoted by: $Fu(C_{i \rightarrow j} \varphi)$. The syntax of CTLC is defined as follows [7]:

$$\varphi ::= p \mid \neg \varphi \mid \varphi \vee \psi \mid EX \varphi \mid E(\varphi U \psi) \mid EG \varphi \mid C_{i \rightarrow j} \varphi \mid Fu(C_{i \rightarrow j} \varphi)$$

A CTLC formula φ in a global state s , denoted by $(M_C, s) \models \varphi$ is recursively defined as follows:

- $(M_C, s) \models p$ iff $p \in v(s)$;
- $(M_C, s) \models \neg \varphi$ iff $(M_C, s) \not\models \varphi$
- $(M_C, s) \models \varphi \vee \psi$ iff $(M_C, s) \models \varphi$ or $(M_C, s) \models \psi$
- $(M_C, s) \models EX \varphi$ iff there exists a path π starting at s such that $(M_C, \pi(1)) \models \varphi$;
- $(M_C, s) \models E(\varphi U \psi)$ iff there exists a path π starting at s such that for $k > 0$, $(M_C, \pi(k)) \models \psi$ and $(M_C, \pi(j)) \models \varphi$ for all $0 \leq j < k$;
- $(M_C, s) \models EG \varphi$ iff there exists a path π starting at s such that $(M_C, \pi(k)) \models \varphi$ for all $k > 0$;
- $(M_C, s) \models C_{i \rightarrow j} \varphi$ if for all global states $s' \in S$ such that $s \sim_{i \rightarrow j} s'$, we have $(M_C, s') \models \varphi$;
- $(M_C, s) \models Fu(C_{i \rightarrow j} \varphi)$ if there exists $s' \in S$ such that $s' \sim_{i \rightarrow j} s$ and $(M_C, s') \models C_{i \rightarrow j} \varphi$

2.3.4 The MCMAS+ Model Checker

In this thesis, the MCMAS+ model checker [7, 28] is used to verify commitments capturing the interactions between web services within compositions. This model checker is an extension of MCMAS, a model checker for MAS [32]. MCMAS+ allows us to verify CTL_C properties. It takes two inputs: a MAS specification and a set of formulae to be verified. It evaluates the truth value of these formulae, producing counterexamples for false formulae, and witnesses for true formulae when it is possible. MCMAS+ allows us to perform the verification of a number of modalities, including CTL operators.

MCMAS+ can also run interactive, step by step simulations. It provides a graphical interface as an Eclipse plug-in that includes a graphical editor and a graphical analyzer for counterexamples. MAS are described in MCMAS+ using ISPL+, which is an extended dedicated programming language derived from the formalism of extended interpreted systems. This language characterizes agents by means of variables, and represents their evolution using Boolean expressions [28]. It distinguishes between two kinds of agents: standard agents and the environment agent, which are modeled similarly. The environment agent is practically used to define and code infrastructures shared by traditional agents. This agent is optional since not all models require one. As in MCMAS, in MCMAS+ each agent is characterized by [32]:

- A set of local states, defined using an enumerated variables.
- A set of actions.
- A protocol describing which action can be performed by an agent in a given local state.
- An evolution function describing how the local states of the agents evolve based on their current local state and on other agents' actions.

The ISPL+ specification also contains the definition of initial states, evaluation (propositions), and formulae to be checked. The following listing shows the general structure of a program.

Listing 2.5: ISPL+ program structure

```

Agent agent1
  Vars:
    state = {s0,s1, s2 ..};
    ...
  end Vars
  Actions = {a1, a2, ...};
  Protocol:
    s0:{a1,...}
    ...
  end Protocol
  Evolution:
    state = s1 if state = s0 and action = a1;
    ...
  end Evolution
end Agent
Evaluation
  label = agent.state = s1;
  ....
end Evaluation
InitStates
  agent1.state = s0 and agent1.state =s0 and
  ...
end InitStates
Formulae
  EF (label);
  ...
end Formulae

```

Evaluation function

An evaluation function consists of a group of atomic propositions, which are defined over global states. Each atomic proposition is associated with a boolean formula over local variables of standard agents and observable variables in the environment agent. The proposition is evaluated to true in all the global states that satisfy the boolean formula. Every variable involved in the formula has a prefix indicating the agent the variable belongs to [32].

Initial states

Initial states are defined by a boolean formula over variables.

Definition of formulae to be checked

A formula to be verified is defined over atomic proposition. Some of the forms that it can have are:

formula ::= (formula)

!formula **and** formula

!formula **or** formula

!

!formula -> formula

!AG formula

!EG formula

!AX formula

!EX formula

!AF formula

!EF formula

!A (formula **U** formula)

MCMAS+ allow us to verify commitment properties using CTL logic. From the user perspective, the additional features are: defining shared variables, using these variables to model communication between agents, and writing CLTC properties to verify commitments [7].

Shared variables

Shared variables are used to model communication channels between two agents. They have to be declared in the *Vars* section of each of the two agents that use it to model their

communication. The name of the variable has to be exactly the same in both agents and it has to start with a lowercase letter.

Commitment properties

Properties are defined in the Formulae section and they are stated as follows:

- $C(i, j, \varphi)$ where agent i is the creditor, agent j is the debtor and φ is the commitment. The commitment has to be defined in the Evaluation section. Meaning agent j commits to agent i to φ .
- $Fu(i, j, \varphi)$, this formulae verifies the fulfillment of the commitment defined above.

In the next chapter, we describe our approach to verify compositions of web services using commitments and the MCMAS+ model checker.

Chapter 3

Verifying Commitment-Driven Composite Web Services

3.1 Introduction

As discussed in the introductory chapter, this work uses MAS as abstraction to reason about compositions of web services. A MAS model is being generated and used for this purpose. The service in charge of the composition as well as the participating services are automatically transformed into agents. The behavior of these agents is modeled using the BPEL code of the corresponding service. Service invocations by the process in charge of the composition are treated as commitments which capture and regulate the interactions between services. Thus, verifying these interactions within a composition scenario can be done through the verification of the underlying commitments.

The proposed framework takes as input the BPEL code of the process in charge of the web service composition and the BPEL code of the participating services, either in executable or abstract mode. Our BPEL2ISPL+ tool transforms these inputs into an equivalent MAS verifiable model specified in ISPL+, and recommend properties in CTL_C to verify

commitments between two services. BPEL2ISPL+ also provides an automata-based view of the system in the DOT language.

Properties in CTLC intend to verify commitments establishment and their fulfillment. Commitments are set up between the agent-based service in charge of the composition and one of the participating agent-based services. Several commitments may be established between two services according to the business logic of the composition. The properties are verified using the MCMAS+ model checker. Out of these results, we can conclude whether services invocations respond as expected or not.

BPEL2ISPL+ transforms each BPEL process (executable or abstract) into an agent. BPEL2ISPL+ uses an intermediate in-memory Agent Object Representation (AOR) to perform this transformation. The reason behind using this intermediate representation is because it is not feasible to develop the final agent's ISPL+ code while reading the BPEL file sequentially due to some decisions that cannot be made immediately before scanning the whole BPEL file. For example, an agent description in ISPL+ starts defining the set of local states, and we cannot know in advance how many states are being used until we finish processing the corresponding BPEL file. The object representation allows us to create all the objects needed to describe the agent before writing its description in the ISPL+ text file. Therefore, BEL2ISPL+ has two parts as illustrated in Figure 3.1. In the first part, AOR objects are created while processing each BPEL code. In the second part, objects are used to write the ISPL+ code.

3.2 From BPEL to Agent Object Representation

In order to create the AOR, the agent description in ISPL+ is first analyzed. The corresponding UML class diagram of this description is then obtained. This class diagram is used to guide the creation of agent objects from BPEL. An example of ISPL+ agent description is

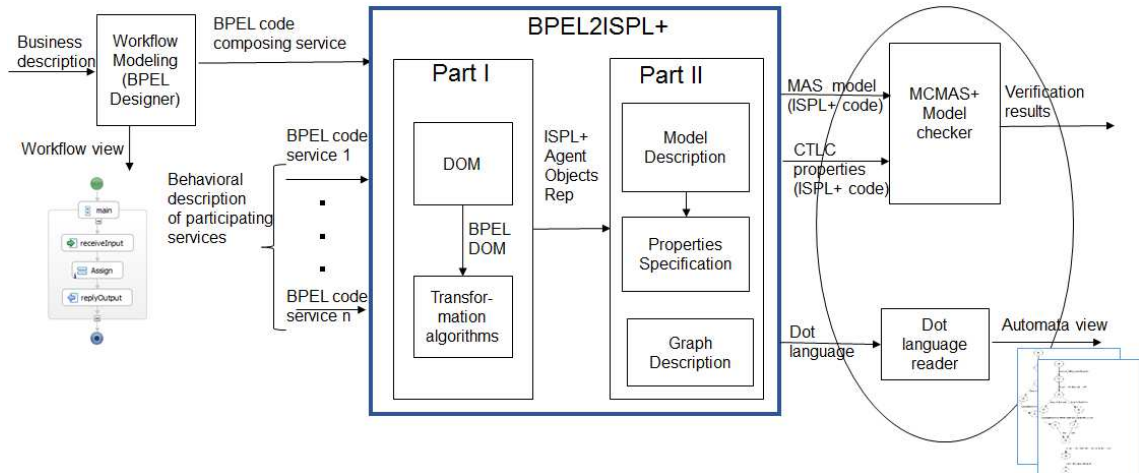


Figure 3.1: BPEL to ISPL+ transformation

shown in the following listing.

Listing 3.1: Agent Definition

```

Agent Process
  Vars :
    State : {SE0, SE1, SE2, SE3};
  end Vars;
  Actions = {Activity1, Activity2, Activity3};
  Protocol
    State = SE0 : {Activity1};
    State= SE1 : {Activity2};
    State= SE2: {Activity 3};
  end Protocol
  Evolution
    State = SE1 if state = SE0 and Action = Activity1;
    State = SE2 if state = SE1 and Action = Activity2;
    State = SE3 if state = SE2 and Action = Activity3;
  end Evolution
end Agent

```

The classes identified in this agent description are: agent, state, action, protocol and evolution. These classes and their relationships are represented in the class diagram depicted in Figure 3.2. The diagram shows the following:

- *Agent* has the attribute name, and has one or many actions, one or many states and one or many evolutions.
- *State* has the attributes name and a label (the label is used in the evaluation section as

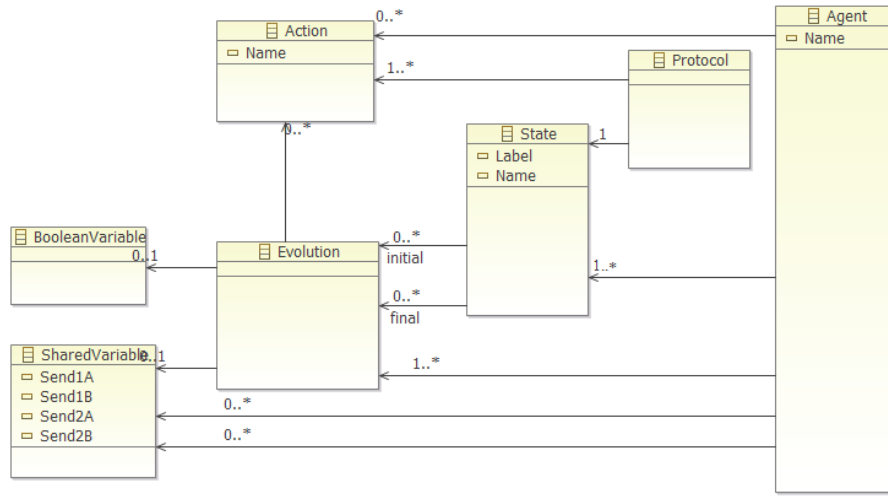


Figure 3.2: Class diagram of agent object representation

described in Section 3.3).

- *Action* has the attribute name.
- *Protocol* is related with one and only one state and with one or more actions.
- *Evolution* is related with a final state from an initial state and one or more actions and one or more Boolean variables.
- *Shared variable* has Send1A, Send1B, Send2A, Send2B. This is part of the implementation of CTLC logic. Its usage is explained in Section 3.2.5.

Part I transforms a BPEL process at a time. It starts creating an agent object for the corresponding process, and converting the BPEL file into a DOM. DOM builds a tree view of the BPEL code (which is an XML document). An example of a tree view of a BPEL file is given in Figure 3.3. In fact, Part I creates agent's objects (Actions, States, Protocols, etc.) by processing each activity in the DOM tree with the algorithms corresponding to each BPEL activity. The algorithms for < sequence >, < if >, < while >, < pick > and < flow > are recursive. In the example provided in Figure 3.3, the root of the tree is the

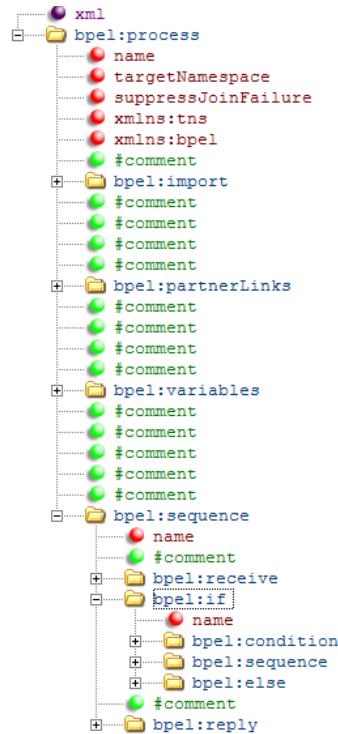


Figure 3.3: Tree view of a BPEL file

< process > clause. Each < process > has to have one BPEL activity among its children, which is a < sequence > in this example. The transformation starts applying the algorithm defined for the < sequence > activity. This algorithm calls the algorithms of its children, namely: < receive >, < if > and < reply >.

Part II is divided into three sub-parts. The first sub-part describes ISPL+ agents using the AOR. The second one completes the description of the MAS, defining the initial states and the evaluation section. The last sub-part proposes commitment properties using the communication between agents.

As explained in Chapter 2, BPEL activities are divided into three groups: partner interaction, basic and structured. Specifically, our transformation tool includes: < invoke >, < receive > and < reply > partner interaction activities; < assign >, < throw > and < opaqueActivity > basic activities; and the < sequence >, < flow >, < if >, < pick > and <

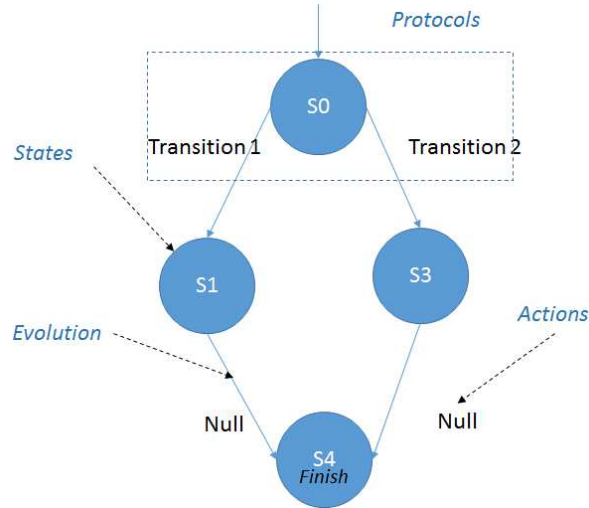


Figure 3.4: ISPL agents depicted as automaton

while> control structured activities. It is important to highlight that all control structured activities are recursive.

Automata are used to visualize the agents representation in ISPL+ produced by the transformation of each BPEL process. Automata are constructed using the following equivalences: agent's states are equivalent to states, agent's evolutions are equivalent to transitions, and agent's protocols define the activities that can be executed from a given state. The equivalences between automata and agents representation are shown in Figure 3.4. To clarify the description of our transformation algorithms, we define the following notations (items 1 and 2) and methods (items 3 to 6):

1. Each agent object representation (AOR) A has a set s of local states, set a of local actions and set e of local evolutions.
2. Each evolution object representation (EOR) e contains: initial state $i \in s$, final state $f \in s$, and action $x \in a$.
3. The *create_new_action(activity)* procedure creates a new action's object (an instance of the class 'Action', see Figure 3.2) using the *activity* data as input.

4. The *create_new_state* procedure creates a new state's object S_y of the class 'State' where y is a consecutive identification number.
5. The *create_new_evolution(initial_state, final_state, action|boolean_expression)* procedure creates a new *evolution*'s object as a transition from *initial_state* to *final_state* labeled with *action* or *boolean_expression*.
6. The *get_last_state* procedure returns a pointer to the agent's last state object.

When starting to transform a BPEL process, an agent with the same name is created along with an initial state S_0 .

3.2.1 Transformation of Partner Interactions Activities

As discussed in Section 2.1.3, a BPEL process can interact with a web service using three distinct interaction activities: *< receive >*, *< reply >* and *< invoke >*. The *< receive >* activity waits for an incoming message. The *< reply >* activity returns a response. The *< invoke >* activity invokes operations on other services. When the business process invokes an operation, it sends a set of parameters that are transformed into messages. If we invoke a synchronous request/response operation, it also returns a message.

Table 3.1 describes the automata representations of partner interaction activities. From the table, we can observe that the activities are transformed into a mix of "Receive" and "Send" actions. This is because from the agent perspective, both *replying* to a request and *invoking* a service can be represented by a "Send" action. On the other hand, receiving a message derived from an invoke can be represented as a "Receive" action. These activities are transformed into AOR using Algorithm 1, when transforming *< invoke >* one state and one evolution were added to model the "Receive" action in synchronous mode.

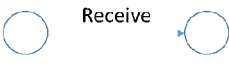
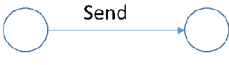


Activity	Garded Automata
Receive	
Reply	
Invoke (one-message)	
Invoke in synchronous way	

Table 3.1: Transformation of partner interaction activities

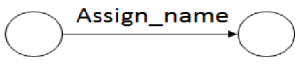

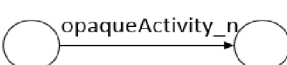
Activity	Garded Automata
Assign	
Throw	
OpaqueActivity	

Table 3.2: Transformation of basic activities

3.2.2 Transformation of Basic Activities

As discussed in Section 2.1.3, the $\langle assign \rangle$ activity manipulates data variables, the $\langle throw \rangle$ activity is responsible for throwing faults, and $\langle opaqueActivity \rangle$ hides functionality in abstract processes. Table 3.2 shows the transformation of these activities, where each one has an initial state, a final state and an evolution transition connecting between them. Each evolution transition has an action name called either *assign_name* in the case of the $\langle assign \rangle$ activity, *fault_name* in the case of the $\langle throw \rangle$ activity, and a number *n* in the case of $\langle opaqueActivity \rangle$. Algorithm 1 describes the transformation of these activities. The algorithm responsible for transforming $\langle throw \rangle$ will be discussed when describing the fault-handlers processing.

Algorithm 1: Transformation of basic activity

Input: BPEL non-recursive activity A

Output: AOR A

- 1 $c := A.get_last_state$
 - 2 $a_i := A.create_new_action(A)$
 - 3 $s_i := A.create_new_state$
 - 4 $e_i := A.create_new_evolution(c, A.s_i, A.a_i)$
-

3.2.3 Transformation of Structured Activities

Each structured activity has a transformation description that provides: an example of the BPEL construct under transformation and the algorithm that describes how to perform the transformation into AOR followed by its execution description using the given example as an input, along with the automaton that describes the resulting AOR. Processes examples are taken from [11] and created using Eclipse BPEL Designer [21].

A. Sequence Transformation

The `<sequence>` construct runs activities sequentially. An example is shown in the left side of Figure 3.5. This workflow starts receiving an input from *client* using `<receive>`. The value is saved into the "input" variable, then it is assigned to the "output" variable using an *assign*, and then this is replied to *client* with the `<reply>` activity. The BPEL code is shown in the following listing.

Listing 3.2: Sequence example

```
<bpel:sequence name="main">
  <bpel:receive name="receiveInput" partnerLink="client"
    portType="tns:HellowWorld"
    operation="process" variable="input"
    createInstance="yes"/>
  <bpel:assign validate="no" name="Assign">
    ...
  </bpel:assign>
```

```

    <bpel:reply name="replyOutput"
      partnerLink="client"
      portType="tns:HellowWorld"
      operation="process"
      variable="output" />
  </bpel:sequence>

```

Algorithm 2 describes how to transform the *< sequence >* BPEL activity into its corresponding AOR. Figure 3.5 shows the flow of the transformation. The algorithm's input is a BPEL *< sequence >* activity *S* that has one or more activities t_1, t_2 , etc.

Algorithm 2: Sequence transformation

Input: BPEL sequence activity *S*

Output: AOR *A*

```

1 iniState := A.get_last_state
2 foreach  $t_i \in S$  do
3   | call algorithm for  $t_i$ 
4 end

```

Execution description

The input *< sequence >* has three activities: *< receive >*, *< assign >* and *< reply >*. The algorithm starts obtaining the initial state *S0*. Then calls the basic algorithm to transform the *< receive >*. When the algorithm is executed, it creates the action "receive", the state *S1*, and one evolution from *S0* to *S1* with the action "receive" on it. The following activity is *< assign >* and therefore the basic algorithm is called again. This algorithm creates the action "assign", the state *S2* and one evolution form *S1* to *S2* with the action "assign" on it. Finally, the basic algorithm is called to transform the *< reply >*. This algorithm creates the action "send", the state *S3*, and one evolution form *S2* to *S3* with the action "send" on it . The automaton is depicted in the right side of Figure 3.5.

B. If Transformation

The intent of the *< if >* construct is to branch from a single activity to exactly one of

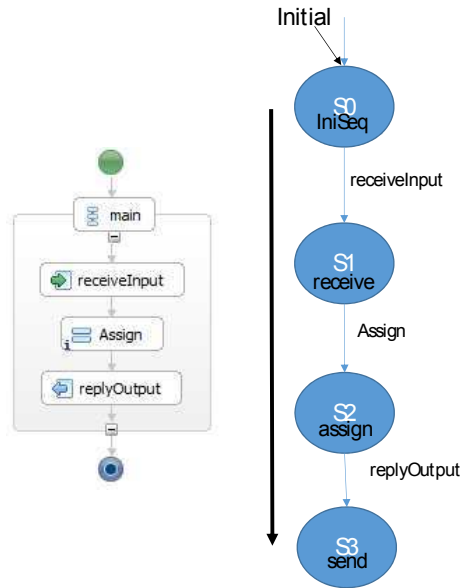


Figure 3.5: Sequence example and its automaton

several branches based on the evaluation of a condition. Branches converge to a single activity, which starts executing when the chosen branch completes.

The left side of Figure 3.6 shows an example. The workflow starts receiving a *name* from *client*, then the value of *name* is compared with "John". If this condition evaluates to **true**, then the output variable is set to "Hello John". On the other hand, if the condition evaluates to **false**, the output variable is set to "You are not John". The output variable is replied to the *client*. The BPEL code is shown in the following listing.

Listing 3.3: If example

```
<bpel:sequence name="main">
  <bpel:receive name="receiveInput" partnerLink="client" .../>
  <bpel:if name="If">
    <bpel:condition>
      <- name == "John"->
      ...
    </bpel:condition>
    <bpel:assign
```

```

        <!-- output = "Hello John" -->
        ...
    </bpel:assign>
    <bpel:else>
        <bpel:assign
            <- output == " Yo are not John" ->
            ...
        </bpel:assign>
    </bpel:else>
</bpel:if>
    <bpel:reply name="replyOutput" partnerLink="client" .../>
</bpel:sequence>

```

Algorithm 3 describes how to transform the $\langle if \rangle$ BPEL construct into its corresponding AOR. The right side of Figure 3.6 shows the flow of the transformation. The algorithm's input is a BPEL $\langle if \rangle$ activity denoted by I that contains one *condition* C , one *activity* t_1 for the **true** branch and one *activity* t_2 for the **false** branch.

Execution description

The input's condition is ($name == "John"$), the **true** branch has an $\langle assign \rangle$ activity and the **false** branch has another $\langle assign \rangle$ activity. The algorithm starts obtaining the initial state, that is S_1 , then processes the **true** branch. The **true** branch creates the state S_2 , one evolution with the guard C from S_1 to S_2 , and calls the algorithm for processing the $\langle assign \rangle$. This algorithm creates the action "assign", the state S_3 , and one evolution with the "assign" action from S_2 to S_3 . When the control returns to the *If algorithm*, it creates a new state S_4 and one evolution with the guard *true* from S_3 to S_4 . After processing the *true* branch, the algorithm processes the *false* branch. This branch creates the state S_5 , one evolution with the guard $\neg C$ from S_1 to S_5 , and calls the algorithm for processing the $\langle assign \rangle$. This algorithm creates the action "assign1", the state S_6 , and one evolution

Algorithm 3: IF transformation

Input: BPEL If *I*

Output: AOR *A*

```
1 iniState := A.get_last_state
2 true branch
3 A.si := create_new_state
4 A.ei := create_new_evolution (iniState, A.si, I.C)
5 call algorithm for t1
6 lastPathState := A.get_last_state
7 A.si := create_new_state
8 A.ei := create_new_evolution (lastPathState, A.si, true)
9 finState = si
10 false branch
11 A.si := create_new_state
12 ei := A.create_new_evolution (iniState, A.si, !I.C)
13 call algorithm for t2
14 lastPathState := A.get_last_state
15 ei := A.create_new_evolution (lastPathState, finState, true)
```

with the action “assign1” from S5 to S6. When the control returns to the *If algorithm*, it creates one evolution with the guard *true* from S6 to S4. The resulting automaton is depicted in Figure 3.6.

C. While Transformation

The `< while >` construct supports repeated executions of a contained activity, which runs as long as the Boolean `< condition >` evaluates to **true** at the beginning of each iteration. The left side of Figure 3.7 shows an example. This workflow receives an input string, creates a new string concatenating 5 times the input and returns the new string to the client. The following listing shows its BPEL code.

Listing 3.4: While example

```
<bpel:sequence name="main">
  <bpel:receive name="receiveInput" .../>
  <bpel:assign validate="no" name="Assign">
    ...
```

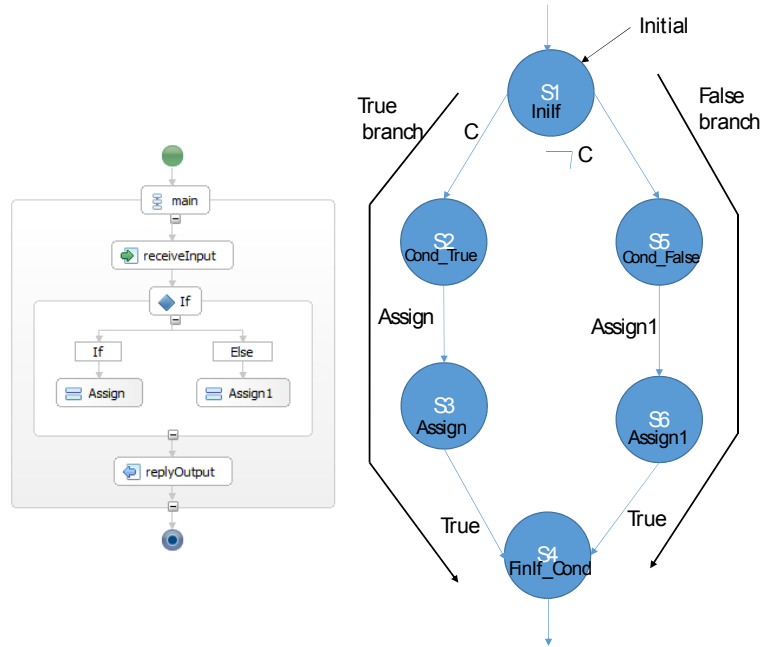


Figure 3.6: If example

```

</bpel:assign>
<bpel:while name="While">
    <bpel:condition ... (iterator < 5)... </bpel:condition>
    <bpel:assign validate="no" name="Assign1">
        ...
    </bpel:assign>
</bpel:while>
<bpel:assign validate="no" name="Assign2">
    ...
</bpel:assign>
<bpel:invoke name="callbackClient" ... />
</bpel:sequence>

```

Algorithm 4 describes how to transform the *< while >* BPEL construct into its corresponding AOR. The flow of the transformation is drawn in the right side of Figure 3.7. The algorithm's input is a BPEL *< while >* activity denoted by *W* which contains one condition

C and one activity t .

Algorithm 4: While transformation

Input: BPEL While W

Output: AOR A

```
1  $iniState := A.get\_last\_state$ 
2 true branch
3  $A.s_i := create\_new\_state$ 
4  $A.e_i := create\_new\_evolution (iniState, s_i, W.C)$ 
5 call algorithm for  $t_i$ 
6  $s_i := A.get\_last\_state$ 
7  $A.e_i := create\_new\_evolution (s_i, iniState, true)$ 
8 false branch
9  $A.s_i := create\_new\_state$ 
10  $A.e_i := create\_new\_evolution (iniState, s, !W.C)$ 
```

Execution description

The input's condition is $(iterator < 5)$ and the **true** branch has an $< assign >$ activity. Thus, the algorithm starts obtaining the initial state that is S_2 , then processes the **true** branch. This branch creates the state S_3 , one evolution from S_2 to S_3 with the guard C , and calls the algorithm for processing the $< assign >$ activity. This algorithm creates the action "assign1", the state S_4 , and one evolution from S_3 to S_4 with the "assign1" action. When the control returns to the *While algorithm*, it creates one evolution with the guard "true" from S_4 to S_2 . After processing the **true** branch, the algorithm processes the **false** branch. This branch creates the state S_5 and one evolution from S_2 to S_5 with the guard $!C$. Figure 3.7 depicts the resulting automaton.

D. Pick Transformation

This activity allows us to specify which branch would be executed according to the received message. Once one of the branches is activated, the other alternative branches are withdrawn. It is important to note that the decision about which branch to take is delayed until a message is received. The left side of Figure 3.8 displays an example. This workflow replies "hello" if it receives "say hello", or replies "goodbye" if it receives "say goodbye".

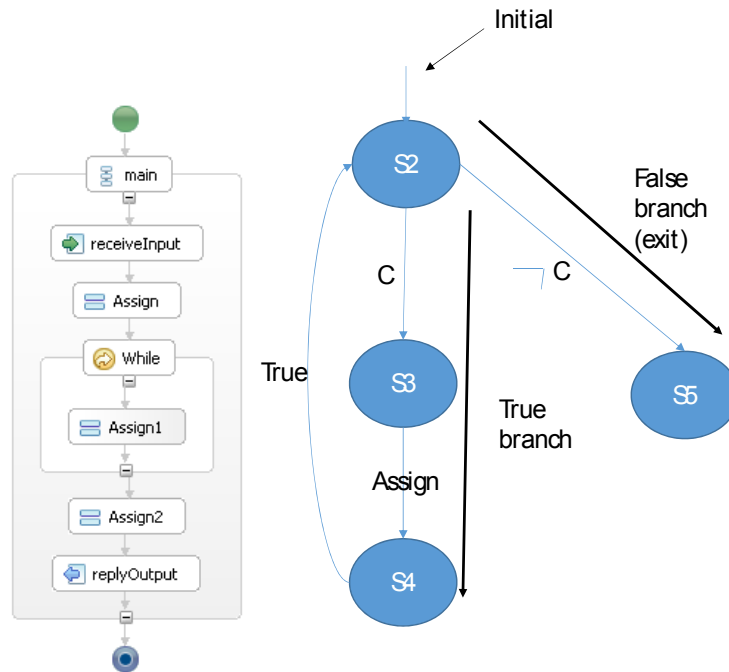


Figure 3.7: While example

The following listing shows the BPEL code.

Listing 3.5: Pick example

```
<bpel:sequence name="main">
  <bpel:pick name="Pick" createInstance="yes">
    <bpel:onMessage partnerLink="client" operation="SayHello" ...>
      <bpel:sequence>
        <bpel:assign validate="no" name="Assign">
          ...
        </bpel:assign>
        <bpel:reply name="replyOutput" partnerLink="client" ... />
      </bpel:sequence>
    </bpel:onMessage>
    <bpel:onMessage partnerLink="client" operation="SayGoodBye" ...>
      <bpel:sequence>
        <bpel:assign validate="no" name="Assign1">
```

```

...
</bpel:assign>
<bpel:reply name="Reply" partnerLink="client"
operation="SayGoodBye" />
</bpel:sequence>
</bpel:onMessage>
</bpel:pick>
</bpel:sequence>

```

Algorithm 5 describes how to transform the $\langle pick \rangle$ BPEL construct into its corresponding AOR. The right side of Figure 3.8 shows the flow of the transformation. The algorithm's input is a BPEL $\langle pick \rangle$ activity denoted by P that has one or more *on-message* denoted by o_j . Each *on-message* o_j has one *activity* denoted by t_i .

Algorithm 5: Pick transformation

Input: BPEL Pick P
Output: AOR A

```

1  $iniState := A.get\_last\_state$ 
2 foreach  $o_j \in P$  do
3    $A.s_i := create\_new\_state$ 
4    $A.a_i := create\_new\_action(o_j)$ 
5    $A.e_i := create\_new\_evolution(iniState, s_i, a_i)$ 
6   call algorithm for  $t_i$ 
7    $s_i := A.get\_last\_state$ 
8   if  $j == 0$  then
9      $finState := A.create\_new\_state$ 
10     $e_i := A.create\_new\_evolution(s_i, finState, true)$ 
11  else
12     $e_i := A.create\_new\_evolution(s_i, finState, true)$ 
13  end
14 end

```

Execution description

The input $\langle pick \rangle$ activity has two branches. The first branch ($j := 0$) is the message "say hello" and it is related to a $\langle sequence \rangle$ that has two activities. The first activity

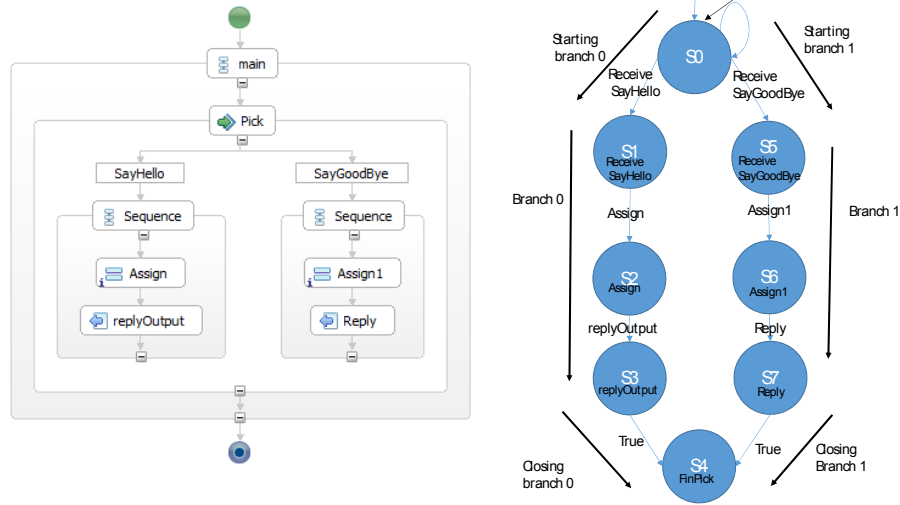


Figure 3.8: Pick example

is an *< assign >* to copy "hello" to an output variable. The second activity replies this variable to the client. The second branch is the message "say goodbye" and it is related to a *< sequence >* that has two activities. The first activity is an *< assign >* to copy "goodbye" to an output variable and the second activity replies this variable to the client. The algorithm starts obtaining the initial state that is S0, then processes the first branch. The first branch creates the state S1, a new "receive" action, one evolution with the "receive" action from S0 to S1, and calls the algorithm for processing the *< sequence >*. When the control returns to the *Pick algorithm*, it gets the last state, that is S3. Then, the condition ($j == 0$) evaluates to **true**. Therefore, it creates a new state S4 and its value is assigned to the variable finState. Thereafter, it creates one evolution with the guard "true" from S3 to S4. The second branch ($j := 1$) is processed in the same way that the first one; the only difference is that the condition ($j == 0$) evaluates to **false**. Then, one evolution is created with the guard "true" from S7 to S4(finState). Figure 3.8 illustrates the resulting automaton.

E. Flow Transformation

The *< flow >* activity sets up parallel activity execution and activity synchronization where

the order of execution is unpredictable. The flow waits for each contained activity to complete before exiting. The left side of Figure 3.9 reveals an example. In this workflow, an input is received, then two *< assign >* activities are executed in parallel inside the *< flow >* activity. Once this is done, an output is replied. The following listing shows the BPEL code.

Listing 3.6: Flow example

```
<bpel:sequence name="main">
  <bpel:pick name="Pick" createInstance="yes">
    <bpel:onMessage partnerLink="client" operation="SayHello" ...>
      <bpel:sequence>
        <bpel:assign validate="no" name="Assign">
          ...
        </bpel:assign>
        <bpel:reply name="replyOutput" partnerLink="client" ... />
      </bpel:sequence>
    </bpel:onMessage>
    <bpel:onMessage partnerLink="client" operation="SayGoodBye" ...>
      <bpel:sequence>
        <bpel:assign validate="no" name="Assign1">
          ...
        </bpel:assign>
        <bpel:reply name="Reply" partnerLink="client"
                                operation="SayGoodBye" ...>
      </bpel:sequence>
    </bpel:onMessage>
  </bpel:pick>
```

Algorithm 6 describes how to transform the *< flow >* BPEL construct into its corresponding AOR. The flow of the transformation is drawn in the right side of Figure 3.9. The algorithm's input is a BPEL *< flow >* activity denoted by *F* that contains *n activities*

t_i where $n > 0$. The procedure *Permutation* ($a_1, a_2 \dots a_n$) produces a set of permutations P_j of the input activities.

Execution description

Algorithm 6: Flow transformation

Input: BPEL Flow F
Output: AOR A

```

1  $iniState := A.get\_last\_state$ 
2 foreach permutation  $P_j(t_1, t_2, \dots t_n)$  do
3   | call sequence algorithm with the activities of  $P_j$  as input
4   |  $lastPermutationState := A.get\_last\_state$ 
5   | if  $j=0$  then
6   |   |  $finState := A.create\_new\_state$ 
7   |   |  $e_i := A.create\_new\_evolution(lastPermutationState, finState, true)$ 
8   | else
9   |   |  $e_i := A.create\_new\_evolution(lastPermutationState, finState, true)$ 
10  | end
11 end

```

The input $\langle flow \rangle$ activity has two branches. The first branch has the activity t_0 which is $\langle assign \rangle$. The second branch has the activity t_1 which is $\langle assign1 \rangle$. The algorithm starts obtaining the initial state $S1$, then obtains the permutations of these activities, which are $P_0 : (t_0, t_1)$ and $P_1 : (t_1, t_0)$. Thereafter, the $\langle sequence \rangle$ algorithm is called using P_0 as input. Inside the sequence, the algorithm for non recursive activities is called twice. The first time, it creates the action "assign", the state $S2$ and an evolution form $S1$ to $S2$ with the action "assign" on it. The second time, it creates the action "assign1", the state $S3$ and an evolution form $S2$ to $S3$ with the action "assign1" on it. When the control returns, the state $S4$ is created and assigned to $finState$. Then, it creates an evolution form $S3$ to $S4$ with **true** as a guard. The $\langle sequence \rangle$ algorithm is called again using P_1 as input and creates the state $S5$, an evolution form $S1$ to $S5$ with the activity "assign1", the state $S6$ and an evolution form $S5$ to $S6$ with the action "assign" on it. When the control returns, an evolution is created form $S6$ to $S4$ with **true** as a guard. Figure 3.9 depicts the resulting automaton.

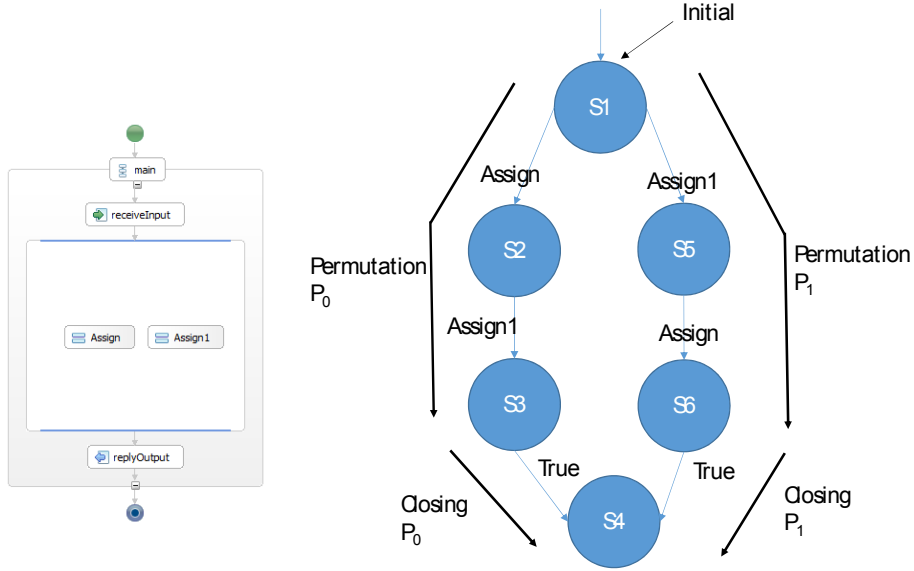


Figure 3.9: Flow example

This algorithm generates the alternatives of execution for the $\langle flow \rangle$ activity performing permutations, taking each branch activity as an element. When processing structured activities, a more accurate representation of the actual processing would be taking each basic activity inside each structured activity as an element to generate the alternatives of execution following the defined order for each branch activity, but interchanging elements of all branches. The algorithm to implement this approach is called *shuffle product* [52]. However, we did not use the *shuffle product* approach because from the verification perspective, it does not add any value compared to the transformation using permutations. The alternatives created using permutations follow the ordering within the $\langle flow \rangle$ branches activities and basic activities of other branches are executed before and after a given branch. This is enough to verify that there is no dependency between a given basic activity of a branch and other branches's elements. Additionally, the *shuffle product* would multiply the number of branches on the automaton, which may cause confusions for the users.

3.2.4 Transforming Fault Handlers

Fault handling allows a BPEL process to deal with run-time faults. Each fault has a fault handler and an activity, which is executed when the associated fault occurs. We transform the fault-handling section after processing the constructs in the main body of the BPEL process. The transformation is done in two steps:

- For each fault-handler, we create a new state's object and label it with the fault's name. Then, we transform its associated activity using the algorithms described earlier.
- After transforming all the fault-handlers, we look for the AOR states objects labeled with a *< throw >* and create an evolution from each one of them to the state labeled with the name of the corresponding fault.

3.2.5 Modeling Communication

We addressed communication transformation between the BPEL process and its partners in Section 3.2.1. However, two more steps are needed. The first one is to synchronize the sending and the corresponding receiving actions. The second one is to include shared variables to model communication channels as required by the extension of the formalism of interpreted systems to use CTL_C. Both steps have to be done after transforming all the input processes into the corresponding AORs.

A. Synchronization

In the AOR "send" and "receive", actions include the name of the partner and the operation they are conducting. To synchronize receiving and sending actions, the corresponding partner action is added to each transition that contains one of these activities. A partner action for a "send" action is a "receive" action of the partner with the same operation. The partner

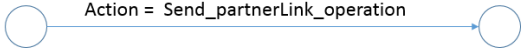
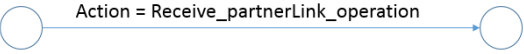
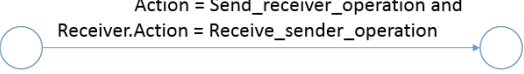
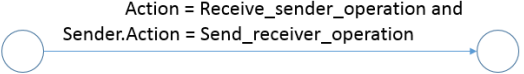
Action	Automata
Send to a non BPEL partner	
Receive from a non BPEL partner	
Send to a BPEL partner	
Receive from a BPEL partner	

Table 3.3: Automata for communications actions

action is included using an *and*. Since some partners may not have a behavioral description, their communication actions are not modified. Table 3.3 shows the automata obtained for communications actions when synchronization is being applied.

B. Shared Variables

In order to apply CTL logic, we model communication channels using shared variables between two services. An enumerated variable is defined in each pair of agents that communicate with each other. An initial value and two values to model communication directions in both ways are enumerated for each variable. The shared variable is initialized when the process starts. Before sending a message, one of the values is assigned to the shared variable. The receiver assigns the same value to the shared variable after receiving the message. Two values are required to differentiate two consecutive messages.

Figure 3.10 depicts an example where the shared variable *sv2* is used to model communication between agent *A* and agent *B*. This variable holds one of the following values: *v0*, *v1*, *v2*, *v3* and *v4*. The initial value is *v0*. Values *v1* and *v2* are used to model communication from agent *B* to agent *A*. Variables *v3* and *v4* are used to model communication from agent *A* to agent *B*. Agent *A* changes the value of the variable before sending a message to agent *B*. Agent *B* changes the value of the variable after receiving the message.

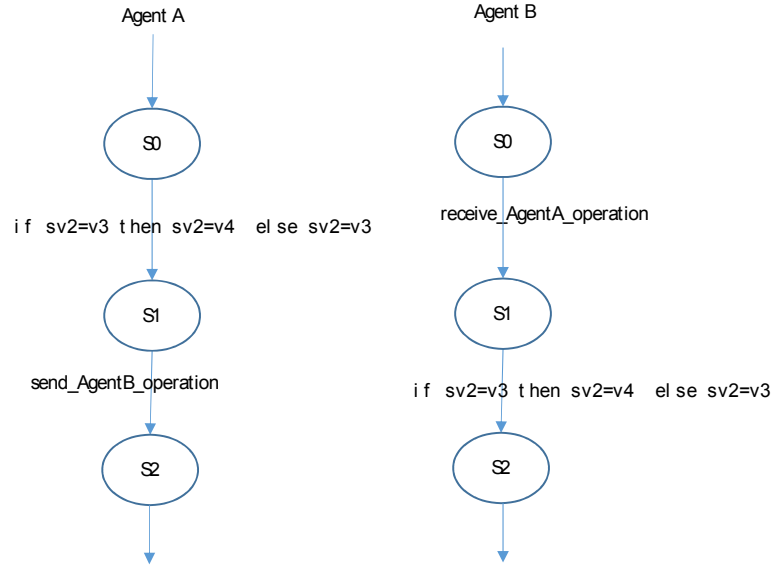


Figure 3.10: Implementation of shared variables

3.3 From Agents Object Representation to ISPL and DOT

A. Agent Description

As explained earlier in Section 2.3.4, an agent description in ISPL+ consists of the following fixed sections: Vars, Actions, Protocol and Evolution. In order to write an ISPL+ agent description, we use each AOR as described in the following steps:

1. declare the agent using the agent's object name;
2. define the state variable using the objects of the state class;
3. define actions section taking all entries from the action class;
4. define protocol section taking all entries from the protocol class and the related state and actions; and
5. define evolution taking all the objects from the evolution class, the initial and target states, and the action.

Moreover, in order to complete the MAS description in ISPL+, three sections are needed: IniStates, Evaluation, and Formulae.

B. Initial States

Each initial state is a Boolean expression indicating that the agent is in its starting point. Shared variables are also initialized in initial states.

C. Evaluation Section

An evaluation is defined for each state object of the agent. During the transformation, a label is saved on each state based on the name of the transition that ends on it. The format of the evaluation is as follows:

label if Agent.State = Sx;

When all the agents are being processed, an evaluation section for communication between agents is written. Each occurrence is performed matching the sending state from one agent with the receiving state of its partner using the labels saved on each state object. The format of the evaluation is as follows:

Agent1_Agent2_OperationName if Agent1.State = Sx and Agent2.State = Sy;

D. Formulae Section

As seen in Section 2.3.4, we define commitment properties and the corresponding fulfillment as follows:

- C(agent2, agent1, commitment1);
- Fu(agent2, agent1, commitment1);

The transformation assumes that each invocation from the agent in charge of the orchestration to a participating agent establishes a commitment. Therefore, for each invocation a commitment property and its fulfillment is generated. The format of the transformation is as follows:

- EF C(agentInCharge, participatingAgent1, commitment1); – meaning there exists a computation so that in its future agentInCharge commits toward participatingAgent1 to bring about commitment1.
- EF Fu(agentInCharge, participatingAgent1, commitment1)); – meaning there exists a computation so that in the future the fulfillment of the commitment will hold.

3.4 From Intermediate Representation to DOT

In order to provide a graphical visualization of each agent, a guarded automaton is drawn using its AOR. A DOT file for each agent is written, transforming each evaluation object into the following format:

IniState ->FinalState [label = evolution]

In the following listing we show the output for the sequence example in Section 3.2.3.

Listing 3.7: DOT example

```
Digraph{
s0->s1 [label="receive_client_SayHello "];
s1->s2 [label="assign_Assign "];
s2->s3 [label="send_client_SayHello "];
s3->s4 [label="null "];
s0->s5 [label="receive_client_SayGoodBye "];
s5->s6 [label="assign_Assign1 "];
s6->s7 [label="send_client_SayGoodBye "];
s7->s4 [label="null "];
}
```

This file creates the automaton displayed in Figure 3.11 using a DOT visualization software.

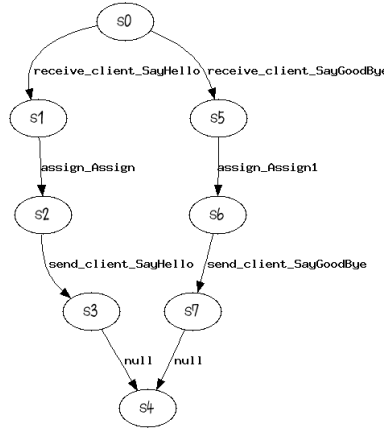


Figure 3.11: Sequence automaton

3.5 Verification

We perform the verification loading the code generated in ISPL+ into the MCMAS+ model checker. The graphic description of the automata of the agents are useful to understand the transformation results. After running the validation we can obtain the results of each property validation. The tool also provides a witness when the property evaluates to **true** and a counter example when the formula evaluates to **false**.

3.6 Case Study

3.6.1 Outline

The "Time Sheet Submission Service Process" (TSP) orchestrates several services. In this section, we show the example of two web services: Employee and Invoice to create a new service as depicted in Figure 3.12. However, these two services include all the constructs of BPEL that we considered in this thesis. The objective is to show through a simple example the feasibility of our transformation-based verification framework. Extending the example to include more services is straightforward. Appendix A shows the example of composing

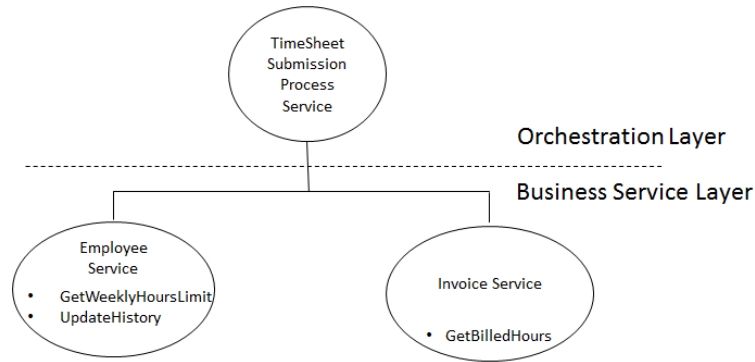


Figure 3.12: Timesheet submission service layers

5 services (4 services plus the orchestrator).

A. Participating Services

Employee web service

The Employee service has two operations: **GetWeeklyHoursLimit** and **UpdateHistory**. **GetWeeklyHoursLimit** operation receives the employee identification data, and returns the largest amount of hours that he can work on a given week. In order to get this amount, it performs an addition of the largest amount of hours that he can work each day of the week. With regard to **UpdateHistory** operation, it receives information related to the employee and saves it on the historical data. The abstract BPEL Process that describes this service behavior is described in Figure 3.13.

Invoice web service

The Invoice service has only one operation: **GetBilledHours**. This operation receives as parameters the employee identification and a given week, and returns the number of billed hours. The abstract BPEL Process that describes its behavior is described in Figure 3.14.

B. Orchestrating Web Service

This service receives a time-sheet as an input, then verifies that the information contained is valid. If the validation fails, it is recorded in the employee history. This process can be done with two **< if >** construct. However, it was done using the **< flow >** construct in

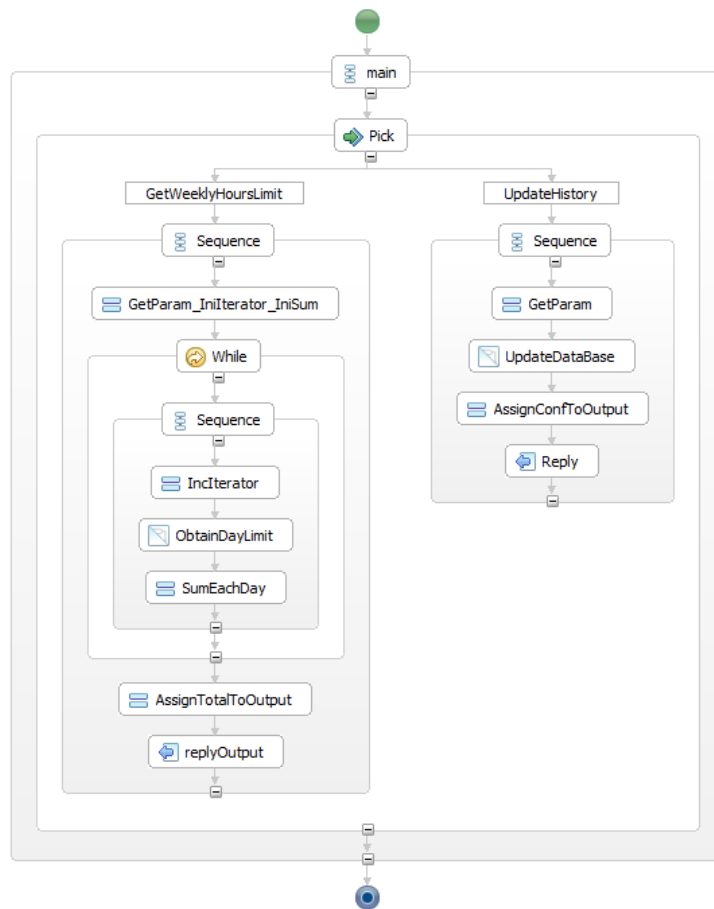


Figure 3.13: Employee web service

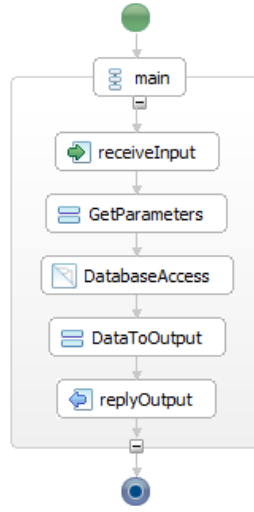


Figure 3.14: Invoice web service

order to show the fault handler functionality. The process is depicted in Figure 3.15.

3.6.2 Transformation

BPEL2ISPL+ transformed the BPEL code of the process in charge of the orchestration and the abstract BPEL code of the participating web services. BPEL2ISPL+ provided two files as outputs: *name.ispl* containing the MAS description in ISPL+, and *name.gv* containing the graphical description of the agents. The files are named as the process in charge of the orchestration.

The Invoice automaton is depicted in Figure 3.16. It has one path (from s_0 to s_7) resulting from the $\langle sequence \rangle$ construct transformation. The activities inside this construct are transformed into sequential actions as described by Algorithm 2. Each basic activity is transformed using Algorithm 1. The resulting actions are: *receive_TSPa_GetBilledhours* on the evolution from S_0 to S_1 , *assign_GetParameters* on the evolution from S_2 to S_3 , *opaque_Activity_DatabaseAccess* on the evolution from S_3 to S_4 , *assign_DataToOutput* on the evolution from S_4 to S_5 and *send_TSPa_GetBilled_hours* on the evolution from S_6

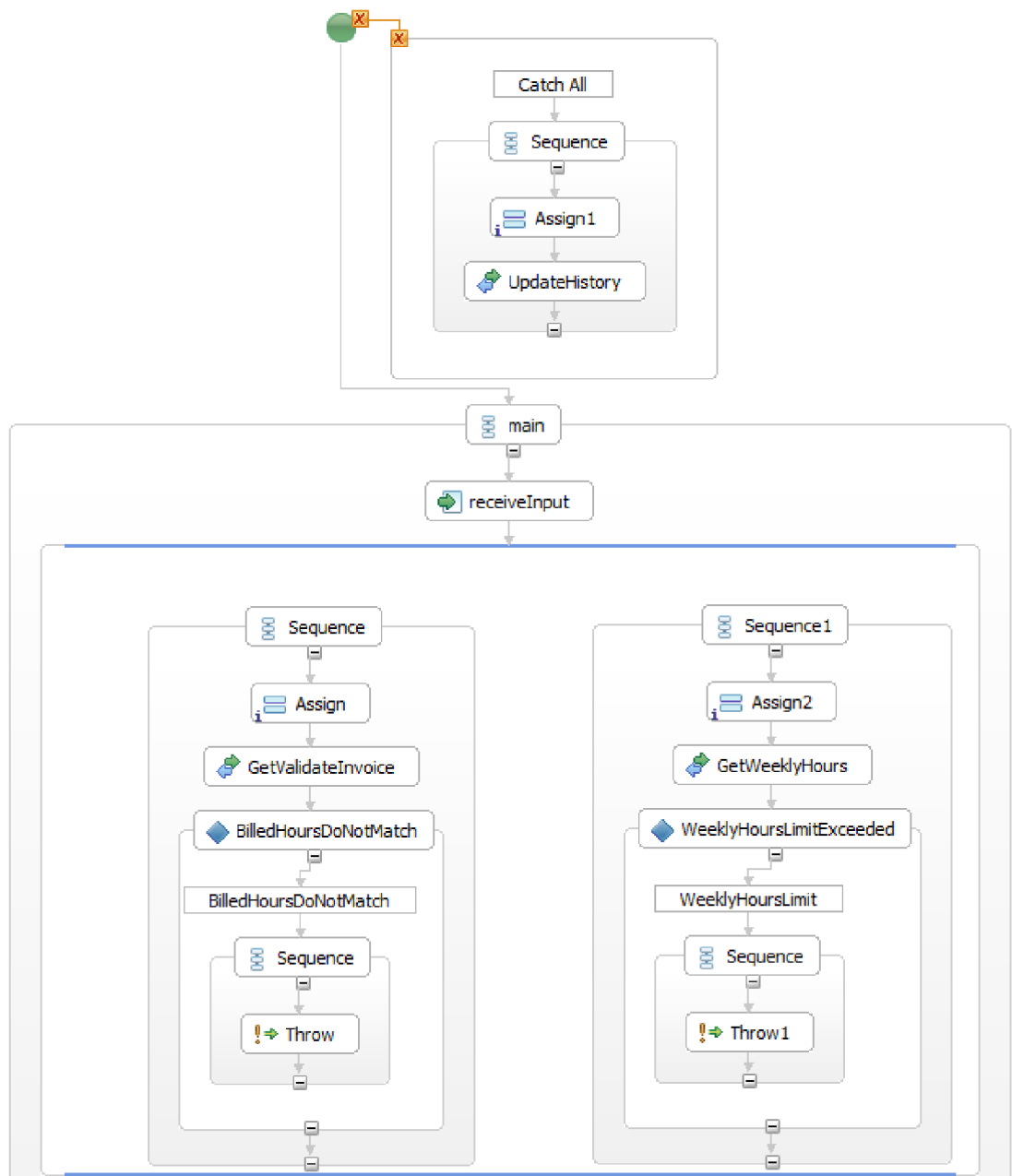


Figure 3.15: Timesheet submission service process

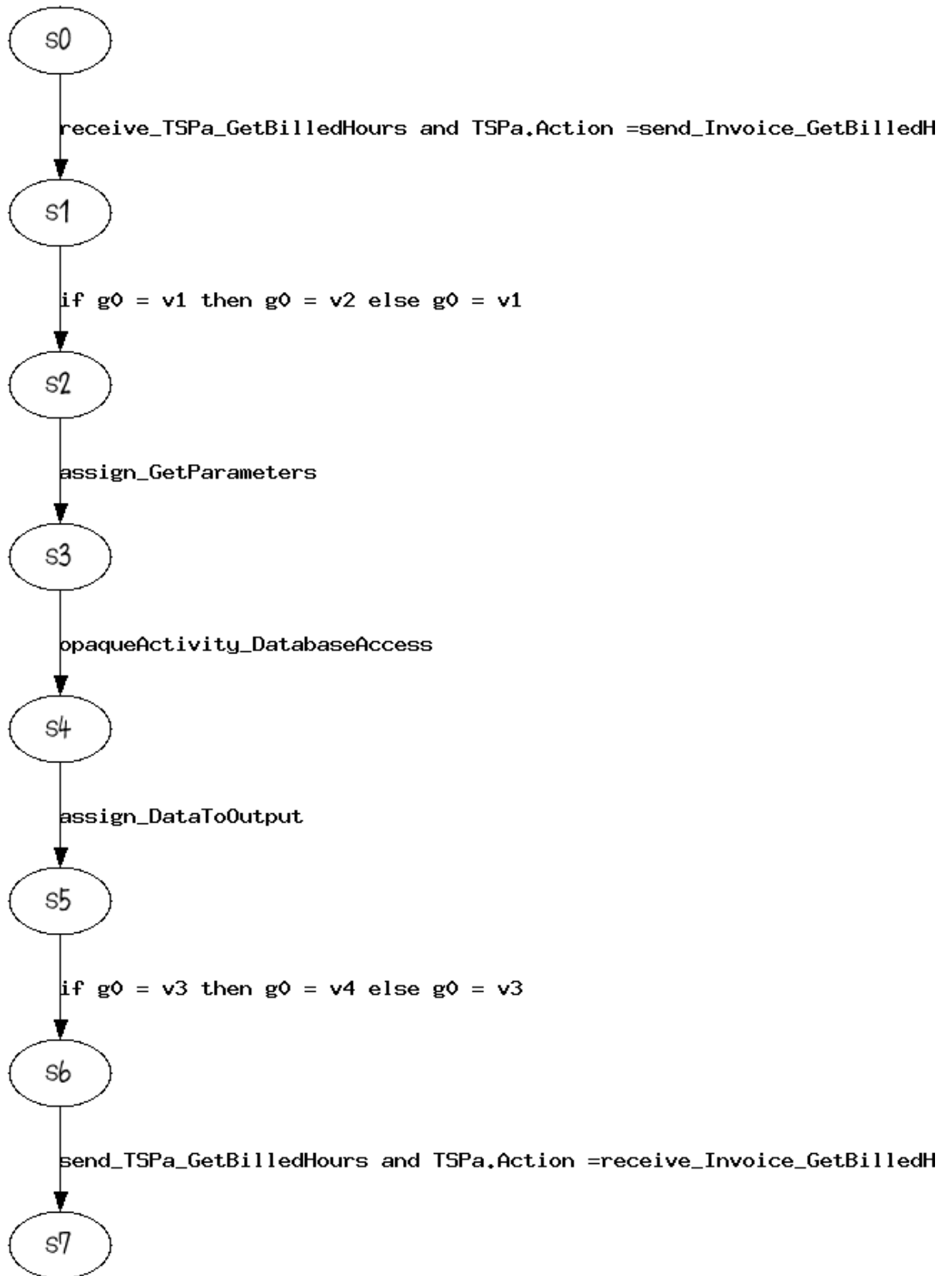


Figure 3.16: Invoice agent automaton

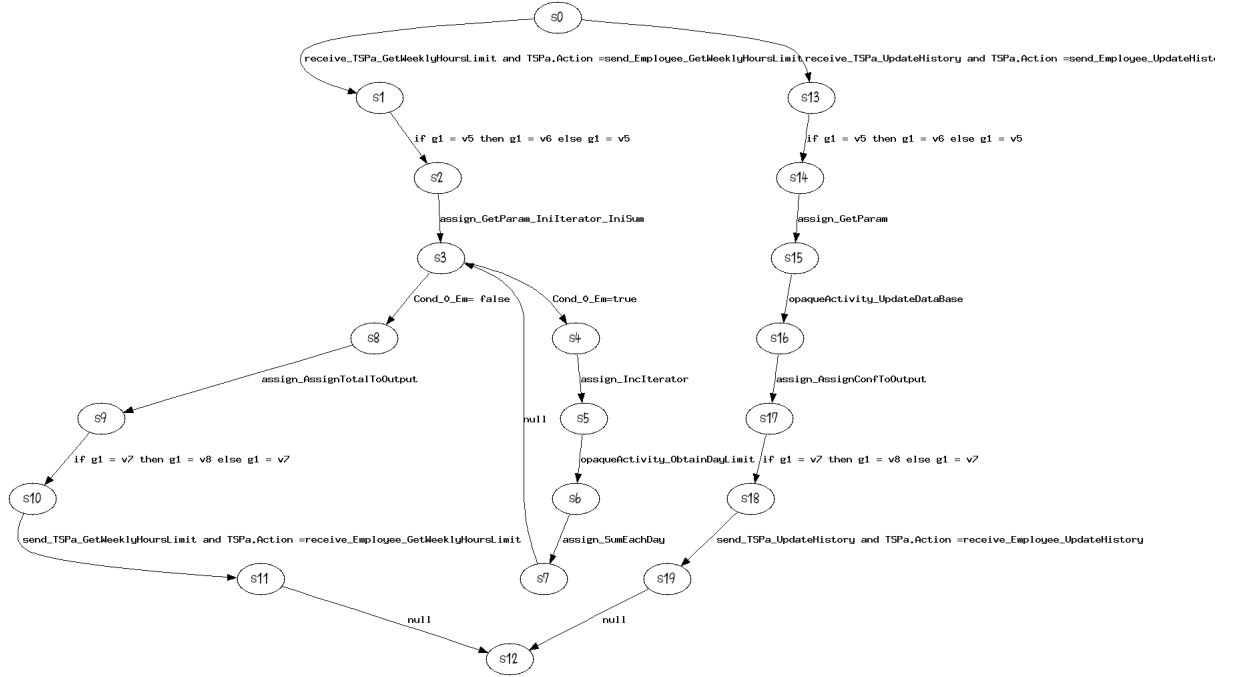


Figure 3.17: Employee agent automaton

to S7.

In order to model communication, as explained in 3.2.5, the receive action is synchronized with a send action from the corresponding agent and operation, and the send action is synchronized with a receive action from the corresponding agent and operation. Additionally, two evolutions are added to assign values to the shared variable $g0$ with the TSP agent. The values $v1$ and $v2$ are used when this agent receives information from the TSP agent. One of these values is assigned after receiving information. The values $v3$ and $v4$ are used when information is sent to the TSP agent. One of these values is assigned before sending information. As discussed earlier, two values are used to differentiate consecutive actions in the same direction. The initial value of the shared variable $g0$ is $v0$.

The Employee automaton is depicted in Figure 3.17. It starts with two paths resulting from the $\langle \text{pick} \rangle$ construct transformation. Each path starts with a receive action. The first path starts with the evolution from $s0$ to $s1$ with the action `receive_TSPa_GetWeeklyHoursLimit`

and finishes with the evolution from s11 to s12 with the label **true**. The second path starts with the evolution from s0 to s13 with the action `receive_TSPa_UpdateHistory`. It ends with an evolution from s19 to s12 with the guard **true**.

The first path continues with the result from the $\langle sequence \rangle$ construct transformation. The evolution from s2 to s3 has the action `receive_TSPa_GetWeeklyHoursLimit`. Then, states S3 to S8 implement the while transformation, followed by the evolution from s8 to s9 with the action `assign_AssignTotalToOutput` and the evolution from s10 to s11 with the action `receive_TSPa_GetWeeklyHoursLimit`. Algorithm 1 is used to perform the $\langle while \rangle$ construct transformation. It has two branches: one when the condition evaluates to **true** and the other when the condition evaluates to **false**. The **true** branch starts with the evolution from s3 to s4 when the condition evaluates to **true**, then transforms the $\langle sequence \rangle$ construct inside the while. The evolution s4 to s5 has the action `assign_incIterator`, the evolution s5 to s6 has the action `opaqueActivity_ObtainDayLimit` and the evolution from s6 to s7 has the action `assign_SumEachDay`. An evolution from s7 to s3 with the **true** guard finishes the **true** branch. The evolution from s3 to s8 with the guard `Cond_0_em` is **false** implements the **false** branch. The second path has four additional actions resulting from the $\langle sequence \rangle$ construct transformation, which are `assign_GetParam`, `opaqueActivity_UpdateDatabase`, `assign_AssignConfToOutput` and `send_TSPa_UpdateHistory`. Finally, service communication is modeled in the same way as in the Invoice service.

The TSP automaton is depicted in Figure A.2. The automaton starts with the evolution from s0 to s1 with the `receiveInput` action, then it transforms the $\langle flow \rangle$ construct that has two $\langle sequence \rangle$ constructs: *Sequence* and *Sequence1* using Algorithm 6. As a result, the automaton has two branches out of the the permutations of *Sequence* and *Sequence1*. The branch s1 to s15 is the result of the transformation of the first permutation,

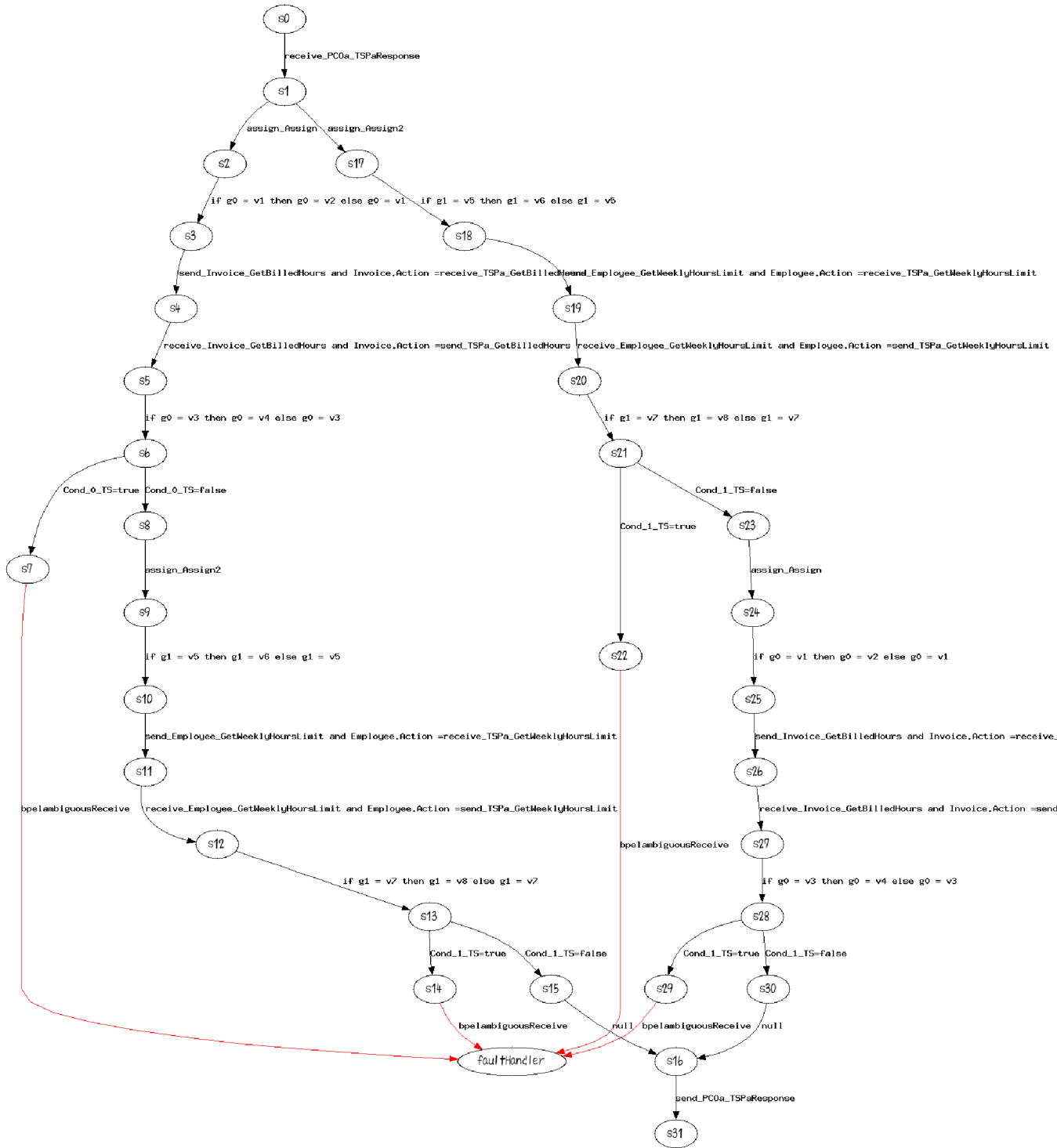


Figure 3.18: TSP agent automaton

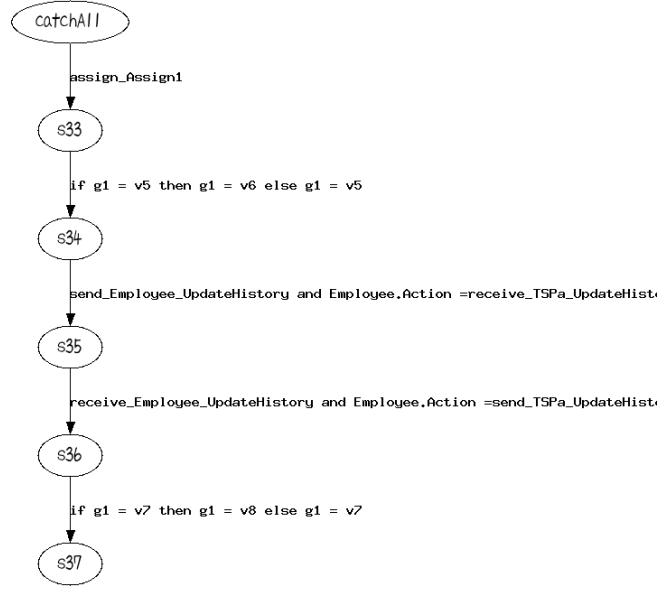


Figure 3.19: TSP agent automaton continued

which starts with Sequence followed by Sequence1. The branch from s1 to s30 is the result of the transformation of the second permutation, which starts with Sequence1 followed by Sequence. Both branches close with an evolution to s16 with **true** as guard. The automaton ends with an evolution from s16 to s31 with the action send.

The branch from s1 to s15 begins with the Sequence *< sequence >* construct transformation, which starts with an assign_Assign action on the evolution from s1 to s2. Then continues with the invoke transformation from s3 to s4 and from s4 to s5. Thereafter, it transforms the *< if >* construct using Algorithm 3. The **true** branch starts with the evolution from s6 to s7 with **true** as guard. Since the following activity is a *< throw >*, s7 has a variable indicating which fault is thrown. The evolution will be completed after processing fault-handlers. The false branch is implemented by the evolution from s6 to s8. Up to now, the transformation of Sequence is completed. Thus, we continue with the transformation of Sequence1. It starts with the action assign_Assign2 evolution from s8 to s9. It continues with the invoke transformation from s10 to s12. Then, it transforms the *< if >* construct.

The **true** branch starts with the evolution from s13 to s14 with **true** as label. Since the following activity is a *< throw >*, s14 has a variable indicating which fault is thrown. The false branch is implemented by the evolution from s13 to s15. The branch from s1 to s30 that results from the second permutation starts transforming Sequence1 followed by Sequence. The steps to transform them are the same as the first permutation. Finally, the communication part is modeled in the same way as for the Employee and Invoice services.

The fault handler section is transformed as explained in Section 3.2.4 and is depicted in Figure 3.19. This case has the *CatchAll* Fault Handler. The program creates a new state s32 and label it as the beginning of the *CatchAll* fault handler. Then, it transforms the nested *< sequence >* construct, which ends in state s37. Once this is done, the program jumps back to the main process transformation and looks for all states that have a label indicating that a fault is thrown from it. An evolution to s32 from s7, s14, s22 and s29 is created with the name of the corresponding fault handler on it. To improve readability, when depicting the automaton for the fault handler, states are divided in two: fault state and fault handler state.

3.6.3 Verification Results

Once the ISPL+ file is generated, the next step is to load it into MCMAS+. The properties checked are stated as follows:

1. EF C(TSPa, Invoice, opaqueActivity_DatabaseAccess)
2. EF Fu(TSPa, Invoice, opaqueActivity_DatabaseAccess)
3. EF C(TSPa, Employee, assign_AssignTotalToOutput)
4. EF Fu(TSPa, Employee, assign_AssignTotalToOutput)

Property one checks if there is a possible computation where the Invoice agent commits to TSPa agent to perform DatabaseAccess. Property two checks if this commitment

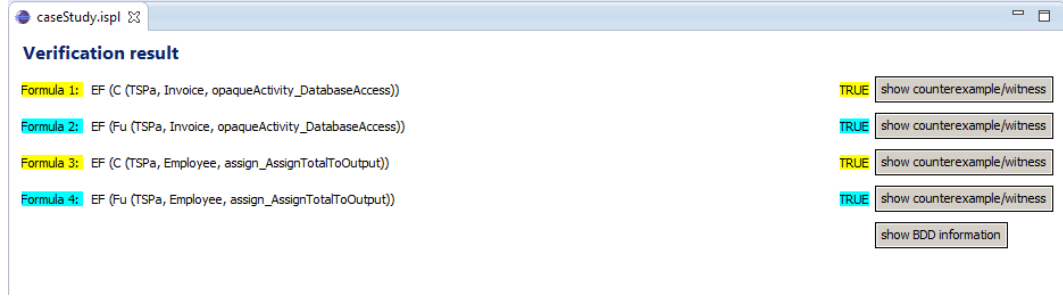


Figure 3.20: Verification results

is fulfilled. Property three checks if it is possible in some future run that Employee agent commits to TSPa agent about AssignTotalToOutput. Property four checks if this commitment is fulfilled. As we can see in the verification results shown in Figure 3.20, all the properties hold.

Figure 3.21 shows a witness for property one, which means that Invoice agent commits to TSPa agent to perform DatabaseAccess in global state three. Figure 3.22 shows a witness for property two, which means that the commitment established in property one is fulfilled in global state eight. Figure 3.23 shows a witness for property three, which means that Employee agent effectively commits to TSPa agent to perform AssignTotalToOutput in global state three. Figure 3.24 is a witness for property four, meaning that the commitment established in property three is fulfilled in global state twenty.

3.7 Related Work

Morimoto recently surveyed the work undertaken the formal verification of BPEL processes [39] using the transformation process. He categorized current approaches according to the transformed formal models into *Automata*, *Petri-net* and *Process Algebra*. In this thesis, we extend Morimoto's taxonomy [39] with a new category called *Interpreted System models* and discuss the following approaches.

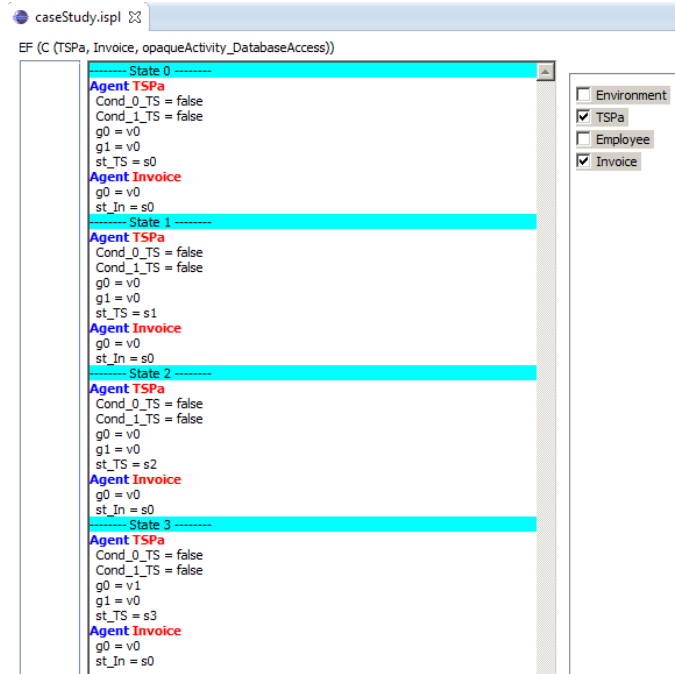


Figure 3.21: Witness of the formula one

An automaton generally comprises of a set of states, transitions between states, an initial state, and actions. Specification models, which specify system behavior can be derived from automata. The approaches lie in this category transform BPEL processes into automata in order to perform formal verification using either NuSMV [12], UPPAL [6] or SPIN [24] model checkers. Among these approaches, there is only one approach that develops the BPEL2SMV tool to automatically perform the transformation from BPEL processes into an automaton model, which is then transformed into a SMV model [38]. The correctness of the resulting SMV model is verified against properties formalized in CTL using NuSMV. However, the approach does not present and discuss the transformation rules that they use in the transform process in order to be able to compare with our rules.

Petri net is an approach to model concurrent systems. The Petri net model comprises of places, transitions, and arcs. The arcs connect between places and transitions and they are directed. In this category, there are several approaches that transform BPEL processes into

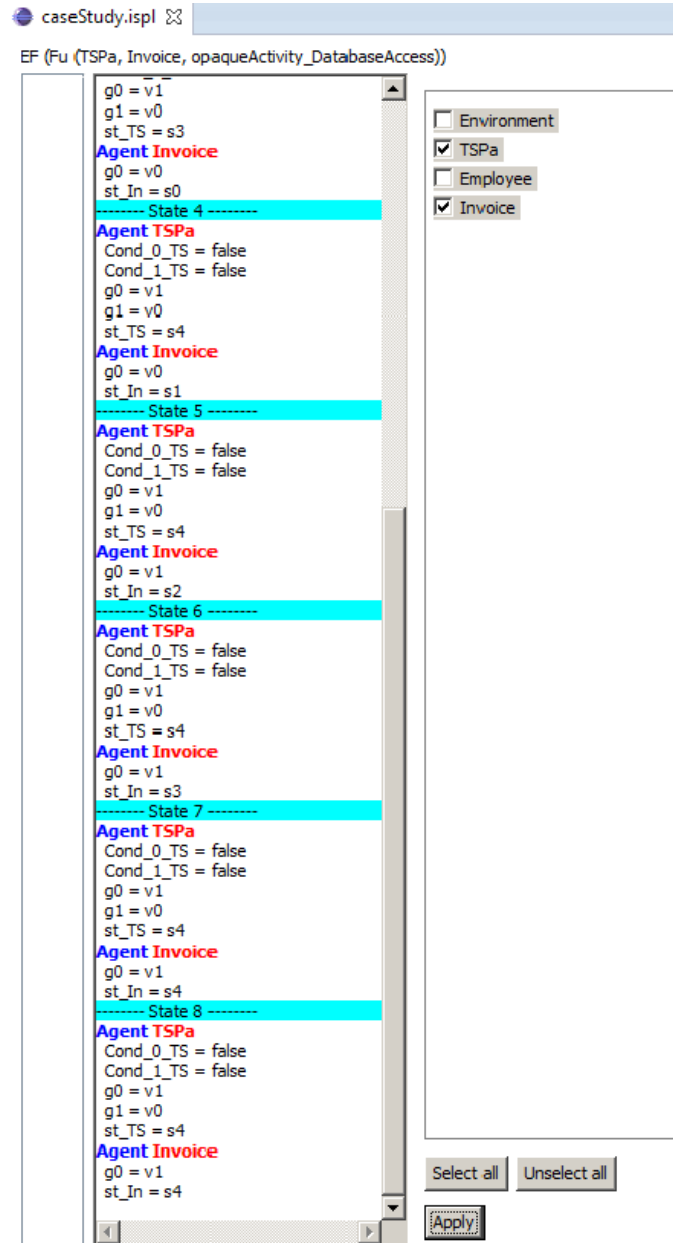


Figure 3.22: Witness of the formula two

caseStudy.ispl

EF (C (TSPa, Employee, assign_AssignTotalToOutput))

----- State 0 -----

Agent TSPa

Cond_0_TS = false
 Cond_1_TS = false
 g0 = v0
 g1 = v0
 st_TS = s0

Agent Employee

Cond_0_Em = false
 g1 = v0
 st_Em = s0

----- State 1 -----

Agent TSPa

Cond_0_TS = false
 Cond_1_TS = false
 g0 = v0
 g1 = v0
 st_TS = s1

Agent Employee

Cond_0_Em = false
 g1 = v0
 st_Em = s0

----- State 2 -----

Agent TSPa

Cond_0_TS = false
 Cond_1_TS = false
 g0 = v0
 g1 = v0
 st_TS = s17

Agent Employee

Cond_0_Em = false
 g1 = v0
 st_Em = s0

----- State 3 -----

Agent TSPa

Cond_0_TS = false
 Cond_1_TS = false
 g0 = v0
 g1 = v5
 st_TS = s18

Agent Employee

Cond_0_Em = false
 g1 = v0
 st_Em = s0

☐ Environment

☒ TSPa

☒ Employee

☐ Invoice

Figure 3.23: Witness of the formula three

EF (Fu (TSPa, Employee, assign_AssignTotalToOutput))

The interface displays a list of states, each with a set of conditions for two agents: TSPa and Employee. The states are numbered 16 through 20. Each state entry is preceded by a blue header for the TSPa agent and a red header for the Employee agent. The conditions listed for each state are:

- State 16:** Cond_0_TS = false, Cond_1_TS = true, g0 = v3, g1 = v5, st_TS = s11. Agent Employee: Cond_0_Em = false, g1 = v5, st_Em = s2.
- State 17:** Cond_0_TS = false, Cond_1_TS = true, g0 = v3, g1 = v5, st_TS = s11. Agent Employee: Cond_0_Em = false, g1 = v5, st_Em = s3.
- State 18:** Cond_0_TS = false, Cond_1_TS = true, g0 = v3, g1 = v5, st_TS = s11. Agent Employee: Cond_0_Em = false, g1 = v5, st_Em = s8.
- State 19:** Cond_0_TS = false, Cond_1_TS = true, g0 = v3, g1 = v5, st_TS = s11. Agent Employee: Cond_0_Em = false, g1 = v5, st_Em = s9.
- State 20:** Cond_0_TS = false, Cond_1_TS = true, g0 = v3, g1 = v5, st_TS = s11. Agent Employee: Cond_0_Em = false, g1 = v5, st_Em = s9.

On the right side, there is a selection panel with checkboxes for 'Environment', 'TSPa', 'Employee', and 'Invoice'. The 'TSPa' and 'Employee' checkboxes are checked. Below the checkboxes are two buttons: 'Select all' and 'Unselect all'. At the bottom right, there is an 'Apply' button.

Figure 3.24: Witness of the formula four

Petri net models. Some of these approaches perform an automatic transformation. Among these approaches, Lohmann et al. [31] developed the BPEL2oWFN tool that transforms BPEL processes into Petri net models. The authors claimed that the Petri net models can be checked given temporal properties (LTL and CTL) using common model checker tools without considering certain model checker tool.

It is known that process algebras can formally model concurrent systems and their grounded semantics are based on the automata theory. Several derivative algebras have been defined such as Milner's Calculus of Communicating Systems (CCS), Hoare's Calculus of Sequential Processes (CSP) and the Language of Temporal Ordered Systems (LOTOS). While, there are several approaches that can transform BPEL processes into process Algebra models [43] [20], their transformation process is not performed in an automatic manner.

All the above approaches are not considered the multi-agent system paradigm, which can effectively model and reason about the services composition. For this reason, we introduce the interpreted system models. The formalism of interpreted systems were first proposed by Fagin et al.[19] to model distributed systems. Recently, this formalism has been adopted as a formal tool to model multi-agent systems and their characteristics. In this formalism, each agent is defined using a set of local states, local actions, local protocol and local evolution function and initial state. The semi-automatic transformation from BPEL processes to the ISPL models was done by Lousmicio et al. [34], which in turn is based in the transformation proposed by Fu et al. [22]. However, there are at least four technical differences with our approach. The first difference is that our transformation produces ISPL+ code instead of ISPL, which allows us to verify commitments using CTLC. The second difference is in the objective of the thesis: they focus on the contract service

level agreements among Web services instead of Web services themselves and their composition, while our work focuses on commitments verification between the service in charge of the composition and participating services. The third difference is that they state rules, while we describe algorithms that provide implementation details. The fourth difference is in the transformation of the BPEL constructs. Within the language constructs, we include `<throw>` and `<opaqueActivity>` and modify the transformation of `<pick>` and `<flow>` introduced in [34]. The inclusion of `<throw>` allows us to model the fault-triggering to handle exceptions at run time, and the inclusion of `<opaqueActivity>` allows us to include opaque activities. In the modified version of the `<pick>` transformation, we translate every `<on-message>` as the “receive” activity to model interactions among services in a reasonable way. The transformation rule of the `<flow>` activity in the Lousmicio et al.’ proposal is not feasible in the automaton theory.

Discussion: Although interactions between invoked Web services can be described in the automaton, Petri net, process algebra and ISPL models, we cannot express properties that can check the direct interactions among Web services (or agents). In a matter of fact, the interaction among Web services and agents plays a fundamental role in the composition process, as shown in [16, 15]. In the reviewed approaches, the compliance of the transformed models is verified with specifications formalized in pure temporal logics (LTL and CTL). The pure temporal logics waive temporal communication modalities that we can use to directly model, reason and verify Web service (agent) interaction with each other. Additionally, the LTL and CTL formulas are added manually in the transformed models.

In our approach, we add transformation rules to model communication into the shared variables in the ISPL+ models. These shared variables model communication channels between two pair of agents in the commitment logic CTLC introduced in [7]. In fact, CTLC is an extension of CTL with social commitment and fulfillment modalities. The commitment

modality can be use to model and reason about the interaction among Web services enacted by agents. Moreover, we automatically extract atomic propositions from the transformed models. These atomic propositions are used to automatically express different temporal properties.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

In this thesis, we proposed a new approach towards verifying compositions of web services using 1) MAS and commitments as abstractions; and 2) model checking as a formal and automatic verification technique. To perform the verification, we transformed the web services composition into a MAS model where the process in charge of the composition and the participating services were transformed into agent models. We coded the behavior of the resulting MAS using ISPL+, which is the dedicated language of the MCMAS+ model checker. In fact, to represent contractual communications between services within compositions, we used commitments, which are powerful abstractions that allow us to reason about services interactions and capture useful properties by the means of operations on these commitments. CTL_C, the commitment-extension of CTL is used to formally express these properties.

From the implementation perspective, we developed a tool called BPEL2ISPL+ to perform the automatic transformation of the web service composition and its participating services into a MAS verifiable model in ISPL+. We included a graphical representation

that depicts the resulting model in order to facilitate its understanding. We tested our tool with a web service composition case study and loaded the resulting ISPL+ model into the MCMAS+ model checker. In fact, BPEL2ISPL+ automatically generated the ISPL+ model in less than one second, the task that would take several hours if manually done. A careful analysis of the generated model revealed that it is error free, a result that cannot be guaranteed with the manual transformation. Furthermore, The visual representation of the ISPL+ model was extremely helpful when verifying the accuracy of the model. Moreover, we defined a set of properties to verify the soundness of the composition by verifying the commitments established between the participating services including the service in charge of the composition. Verification results showed how these commitments are created and fulfilled, capturing thus the interaction-based composition.

4.2 Future Work

In recent years, companies have migrated to cloud computing to decrease the cost devoted to the management of hardware and software resources. High demand on cloud services has contributed to increase the amount of cloud services offerings [25]. Some approaches have proposed BPEL to perform cloud computing service composition to provide composite functionality out of the combination of several services [4, 14]. Our first future direction is to extend and adapt our approach to verify cloud service compositions. The challenge is to capture, model and then verify both horizontal (same cloud layer) and vertical (cross cloud layers) compositions.

On the other hand, for several years, researchers have proposed approaches where software agents interact with web services [23, 42, 44]. The plan is to expand our approach to systems where agents and web services collaborate. Our current proposal already support the web services behavioral description in ISPL+, which was originally proposed for agents.

In order to include concrete software agents, and not only abstract ones as in the current proposal, their behavioral description including their beliefs and epistemic status should be integrated and then coded in ISPL+. A logic combining knowledge and commitments, for instance the one proposed in [3], should be used to express the model and properties to be model-checked.

Regarding the CTLC logic, currently it verifies fulfillment of commitments when the creditor stays in the local state where the commitment was established. In other words, the creditor has to wait in the state until the debtor accomplishes the commitment. Investigating the removal of this condition from the commitment logic and analyzing its impact on the model checking approach is another direction for further research.

Bibliography

- [1] Web Services Business Process Execution Language version 2.0 (WS-BPEL 2.0). https://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm_Toc166509724.
- [2] Rajendra Akerkar and Priti Sajja. *Knowledge-Based Systems*. Jones & Bartlett Publishers, 2010.
- [3] Faisal Al-Saqqar, Jamal Bentahar, Khalid Sultan, Wei Wan, and Ehsan Khosrowshahi Asl. Model checking temporal knowledge and commitments in multi-agent systems using reduction. *Simulation Modelling Practice and Theory*, 51:45–68, 2015.
- [4] Tobias Anstett, Frank Leymann, Ralph Mietzner, and Steve Strauch. Towards bpm in the cloud: Exploiting different delivery models for the execution of business processes. In *Services - I, 2009 World Conference on*, pages 670–677, 2009.
- [5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.
- [6] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Proceedings of the third International Conference on the Quantitative Evaluation of Systems*, pages 125–126, 2006.

- [7] Jamal Bentahar, Mohamed El-Menshawy, Hongyang Qu, and Rachida Dssouli. Communicative commitments: Model checking and complexity analysis. *Knowledge-Based Systems*, 35:21–34, 2012.
- [8] Mustapha Bourahla and Mohamed Benmohamed. Formal specification and verification of multi-agent systems. *Electr. Notes Theor. Comput. Sci.*, 123:5–17, 2005.
- [9] Mustapha Bourahla and Mohamed Benmohamed. Model checking multi-agent systems. *Informatica*, 29(2), 2005.
- [10] Cristiano Castelfranchi. Commitments: From individual intentions to groups and organizations. In *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, pages 41–48, 1995.
- [11] Nguyen Ngoc Chan. TPs about BPEL process design and execution. http://www-inf.int-evry.fr/cours/WebServices/TP_BPEL/, 2015.
- [12] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404, pages 359–364. Springer, 2002.
- [13] Mika Cohen, Mads Dam, Alessio Lomuscio, and Francesco Russo. Abstraction in model checking multi-agent systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 945–952. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

- [14] Tim Dornemann, Ernst Juhnke, and Bernard Freisleben. On-demand resource provisioning for bpel workflows using amazon’s elastic compute cloud. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, pages 140–147, 2009.
- [15] Warda El-Kholy, Jamal Bentahar, Mohamed El-Menshawy, Hongyang Qu, and Rachida Dssouli. Modeling and verifying choreographed multi-agent-based web service compositions regulated by commitment protocols. *Expert Systems with Applications*, 41:7478–7494, 2014.
- [16] Warda El-Kholy, Mohamed El-Menshawy, Jamal Bentahar, Hongyang Qu, and Rachida Dssouli. Verifying multiagent-based web service compositions regulated by commitment protocols. In *Proceedings of 21th IEEE International Conference on Web Services (ICWS)*, pages 49–56. IEEE, 2014.
- [17] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2005.
- [18] Thomas Erl, Anish Karmarkar, Priscilla Walmsley, Hugo Haas, L. Umit Yalcinalp, Kevin Liu, David Umit Orchard, Andre Tost, and James Pasley. *Web Service Contract Design and Versioning for SOA*. Prentice Hal, 1 edition, 2008.
- [19] Ronald Fagin, Joseph Halpern, Yoram Moses, and Moshe Vardi. *Reasoning about knowledge*. MIT Press, 1995.
- [20] Andrea Ferrara. Web services: A process algebra approach. In Marco Aiello, Mikio Aoyama, Francisco Curbera, and Mike P. Papazoglou, editors, *Proceedings of the Second International Conference on Service-Oriented Computing*, pages 242–251. ACM, 2004.

- [21] The Eclipse Foundation. BPEL Designer Project. <https://eclipse.org/bpel/>, 2015.
- [22] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting BPEL web services. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web*, pages 621–630. ACM, 2004.
- [23] Dominic Greenwood, Margaret Lyell, Ashok Mallya, and Hiroki Suguri. The iee fipa approach to integrating software agents and web services. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 276. ACM, 2007.
- [24] Gerard J. Holzmann. Software model checking with SPIN. *Advances in Computers*, 65:78–109, 2005.
- [25] Amin Jula, Elankovan Sundararajan, and Zalinda Othman. Cloud computing service composition: A systematic literature review. *Expert Systems with Applications*, 41(8):3809 – 3824, 2014.
- [26] Matjaz B. Juric and Marcel Krizevnik. *WS-BPEL 2.0 for SOA Composite Applications with Oracle SOA Suite 11g*. Packt Publishing, 2010.
- [27] Matjaz B. Juric and Denis Weerasiri. *WS-BPEL 2.0 Beginner’s Guide*. Packt Publishing, 2014.
- [28] Warda El Kholy, Jamal Bentahar, Mohamed El-Menshawy, Hongyang Qu, and Rachida Dssouli. Conditional commitments: Reasoning and model checking. *ACM Trans. Softw. Eng. Methodol.*, 24(2):9:1–9:49, 2014.

- [29] Ryan Kirwan and Alice Miller. Model checking multi-agent systems. In *Automated Reasoning Workshop 2010 Bridging the Gap between Theory and Practice ARW 2010*, page 18, 2010.
- [30] Kung-Kiu Lau, Faris M Taweel, and Cuong M Tran. The w model for component-based software development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 47–50. IEEE, 2011.
- [31] Niels Lohmann. A feature-complete petri net semantics for WS-BPEL 2.0. In Marlon Dumas and Reiko Heckel, editors, *Web Services and Formal Methods, 4th International Workshop, WS-FM*, volume 4937, pages 77–91. Springer, 2008.
- [32] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: A model checker for the verification of multi-Agent systems. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *LNCS*, pages 682–688, 2009.
- [33] Alessio Lomuscio, Hongyang Qu, Marek Sergot, and Monika Solanki. *Verifying temporal and epistemic properties of web service compositions*. Springer, 2007.
- [34] Alessio Lomuscio, Hongyang Qu, and Monika Solanki. Towards verifying contract regulated service composition. *Autonomous Agents and Multi-Agent Systems*, 24(3):345–373, 2012.
- [35] Mukul Prasad Masahiro Fujita, Indradeep Ghosh. *Verification Techniques for System-Level Design*. Morgan Kaufmann, 2010.
- [36] Brett McLaughlin and Justin Edelson. *Java and XML, 3rd Edition*. O’Reilly Media, Inc., 2006.

- [37] Yuan Mengting and Yu Chao. Model checking multi-agent systems. In *Service Systems and Service Management, 2007 International Conference on*, pages 1–5. IEEE, 2007.
- [38] Marina Mongiello and Daniela Castelluccia. Modelling and verification of BPEL business processes. In Ricardo Jorge Machado, João M. Fernandes, Matthias Riebisch, and Bernhard Schätz, editors, *Proceedings of the Joint Meeting of The Fourth Workshop on Model-Based Development of Computer-Based Systems and The Third International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, pages 144–148. IEEE Computer Society, 2006.
- [39] Shoichi Morimoto. A survey of formal verification for business process modeling. In *Computational Science ICCS 2008*, volume 5102 of *Lecture Notes in Computer Science*, pages 514–522. Springer Berlin Heidelberg, 2008.
- [40] Wojciech Penczek and Alessio Lomuscio. Verifying epistemic properties of multi-agent systems via bounded model checking. *Fundam. Inform.*, 55(2):167–185, 2003.
- [41] Jesse Russell and Ronald Cohn. *Graphviz*. VDM Publishing, 2010.
- [42] Jarogniew Rykowski and Wojciech Cellary. Virtual web services: Application of software agents to personalization of web services. In *Proceedings of the 6th International Conference on Electronic Commerce, ICEC '04*, pages 409–418. ACM, 2004.
- [43] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. In *Proceedings of the IEEE International Conference on Web Services*. IEEE Computer Society, 2004.
- [44] Omair Shafiq, Ying Ding, and Dieter Fensel. Bridging multi agent systems and web services: towards interoperability between software agents and semantic web services.

- In *Enterprise Distributed Object Computing Conference, 2006. EDOC '06. 10th IEEE International*, pages 85–96, 2006.
- [45] Munindar P. Singh. A social semantics for agent communication languages. In *Issues in agent communication*, pages 31–45. Springer, 2000.
 - [46] Munindar P. Singh. *Practical Handbook of Internet Computing*. Chapman & Hall/CRC Press,, 2004.
 - [47] Munindar P. Singh and Michael N. Huhns. *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley & Sons, Ltd., 2005.
 - [48] Committee Specification. *Web Services Business Process Execution Language Version 2.0*. OASIS, 2007.
 - [49] Andreas Spillner and H Bremenn. The w-model strengthening the bond between development and test. In *Int. Conf. on Software Testing, Analysis and Review*, pages 15–17, 2002.
 - [50] Doug Tidwell. *XSLT, 2nd Edition*. O'Reilly Media, Inc., 2008.
 - [51] W3C. Document object model. <http://www.w3.org/DOM/>, 2015.
 - [52] Andreas Wombacher, Peter Fankhauser, and Erich Neuhold. Transforming BPEL into annotated deterministic finite state automata for service discovery. In *Web Services, 2004. Proceedings. IEEE International Conference on*, pages 316–323. IEEE, 2004.
 - [53] Michael Wooldridge, Michael Fisher, Marc-Philippe Huget, and Simon Parsons. Model checking multi-agent systems with mable. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2, AAMAS '02*, pages 952–959. ACM, 2002.

- [54] Michael Zhivich and Robert K. Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

Appendix A

Extended Case Study

A.1 Outline

The "Time Sheet Submission Service Process" (TSP) discussed in Chapter 3 orchestrates two participating web services: Employee and Invoice. In this appendix, we extend the example with two additional participating web services: Payroll and Project, to create an extended service as depicted in Figure A.1.

A. Participating Services

Payroll web service

The Payroll service has one operation: UpdatePayroll. This operation receives as parameters the employee identification and the number of hours worked in a given week. If the company pays the employee monthly, it adds the hours worked to the monthly records. Otherwise, the company pays the employee according to the hours worked. The abstract BPEL Process that describes this behavior is illustrated in Figure A.2.

Project web service

The Project service has one operation: SetBilledHours. This operation receives as parameters the employee identification, project name, a given week and the number of hours

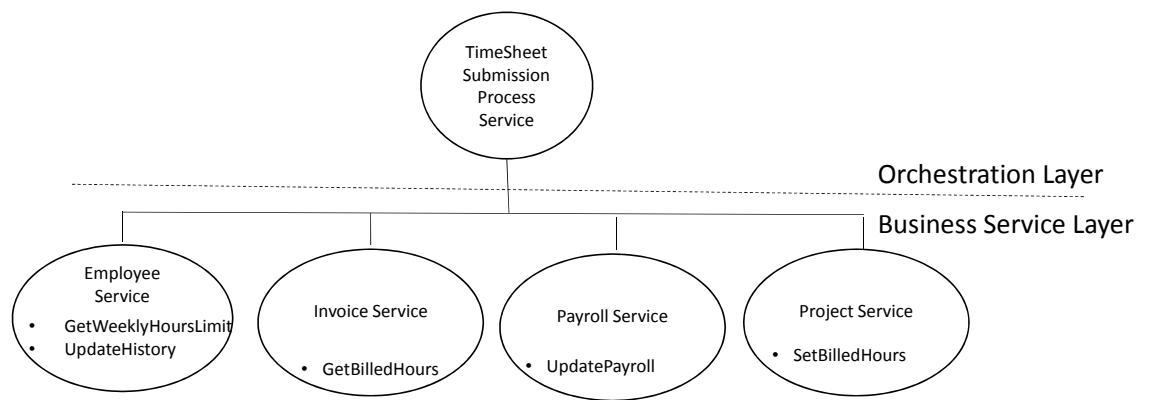


Figure A.1: Timesheet submission service layers

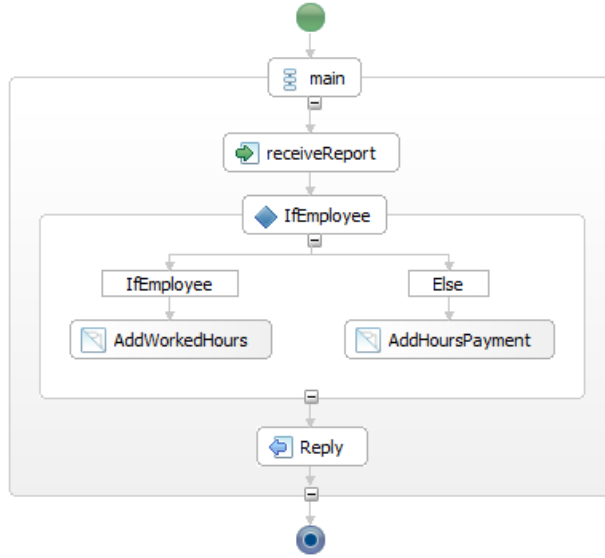


Figure A.2: Payroll web service

worked. It updates the project cost according to the hours worked. The abstract BPEL Process that describes this behavior is shown in Figure A.3.

B. Orchestrating Web Service

As described in Chapter 3, this service receives a time-sheet as an input, then verifies that the information contained is valid. If the validation fails, it is recorded in the employee history. The process is extended after the validation by invocations of Payroll and Project web services. The process is depicted in Figure A.4.

A.2 Transformation

BPEL2ISPL+ is used to transform the BPEL code of the process in charge of the orchestration and the abstract BPEL code of the participating web services. The orchestration of Invoice and Employee web services is the same as described in Chapter 3. Therefore, their corresponding agents are as previously presented. Agents of Payroll and Project web

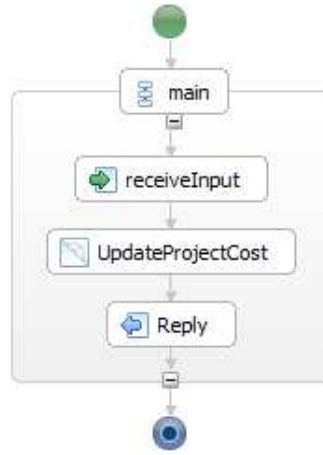


Figure A.3: Project web service

services are described in the appendix as well as the extended version of TSP.

The Payroll automaton is depicted in Figure A.5. It has an $\langle if \rangle$ construct transformation and the nested activities inside this construct are transformed into two branches as described by Algorithm 3. Each basic activity is transformed using Algorithm 1. The resulting actions are: `receive_TSPa_UpdatePayroll` on the evolution from S_0 to S_1 , `opaqueActivity_AddWorkedHours` on the evolution from S_3 to S_4 if `Cond_0_Pa` evaluates to true and `opaqueActivity_AddHoursPayment` on the evolution from S_6 to S_7 if `Cond_0_Pa` evaluates to false, and `send_TSPa_UpdatePayroll` on the evolution from S_8 to S_9 .

In order to model communication, as explained in 3.2.5, the receive action is synchronized with a send action from the corresponding agent and operation, and the send action is synchronized with a receive action from the corresponding agent and operation. Additionally, two evolutions are added to assign values to the shared variable `g2` with the TSP agent. The values `v9` and `v10` are used when this agent receives information from the TSP agent. One of these values is assigned after receiving information. The values `v11` and `v12` are used when information is sent to the TSP agent. One of these values is assigned before

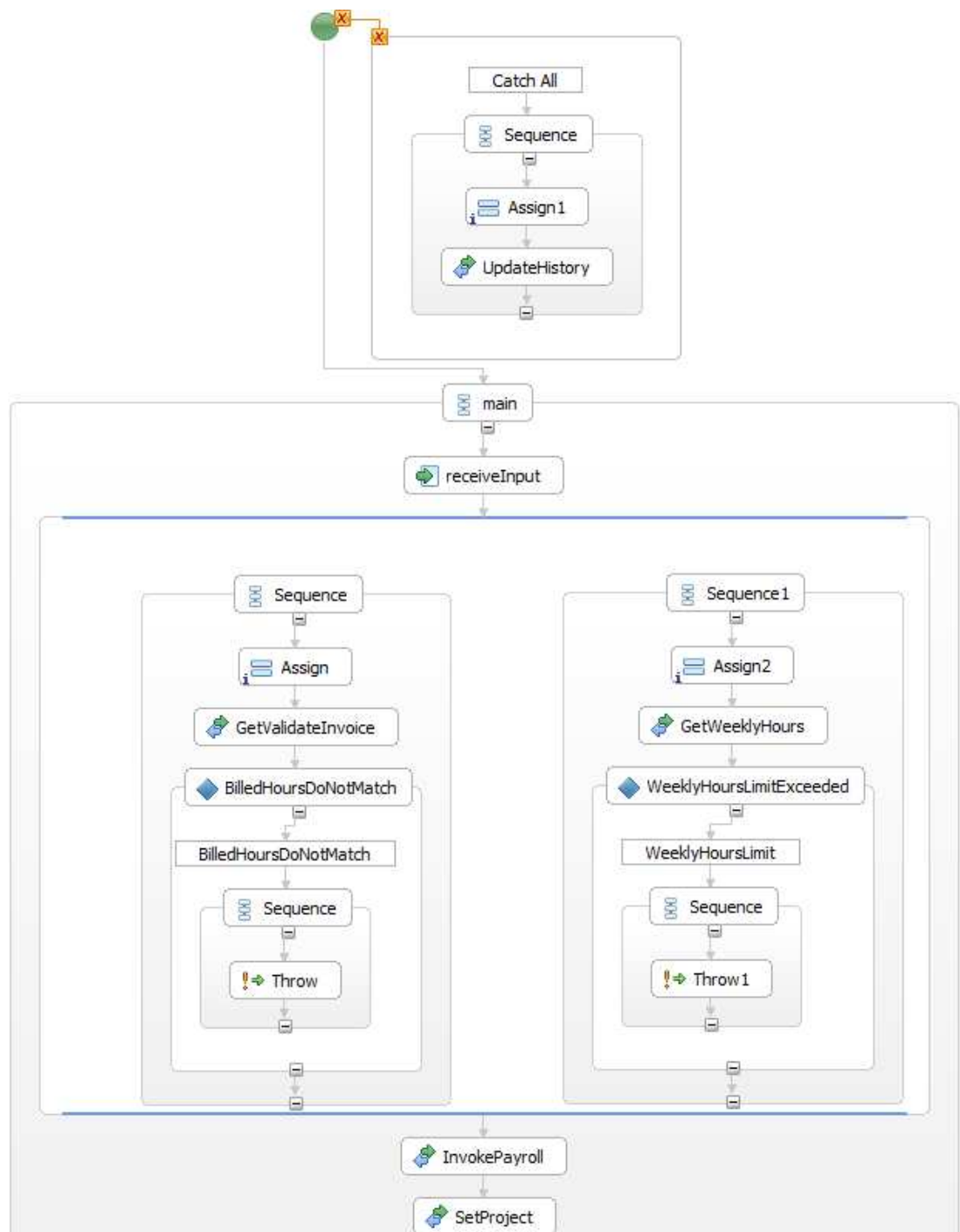


Figure A.4: Timesheet submission service process

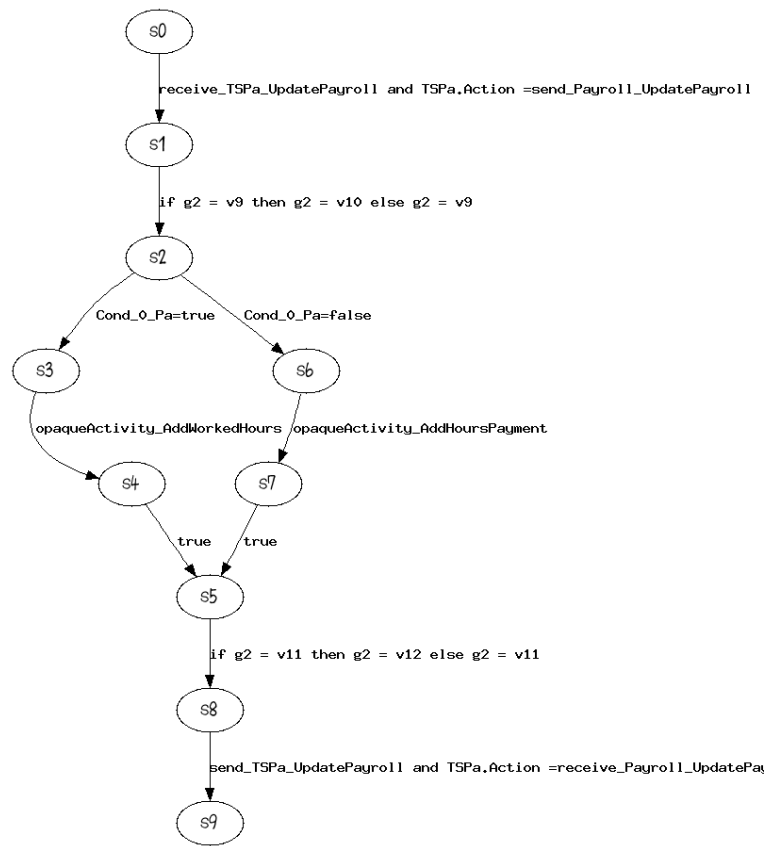


Figure A.5: Payroll agent automaton

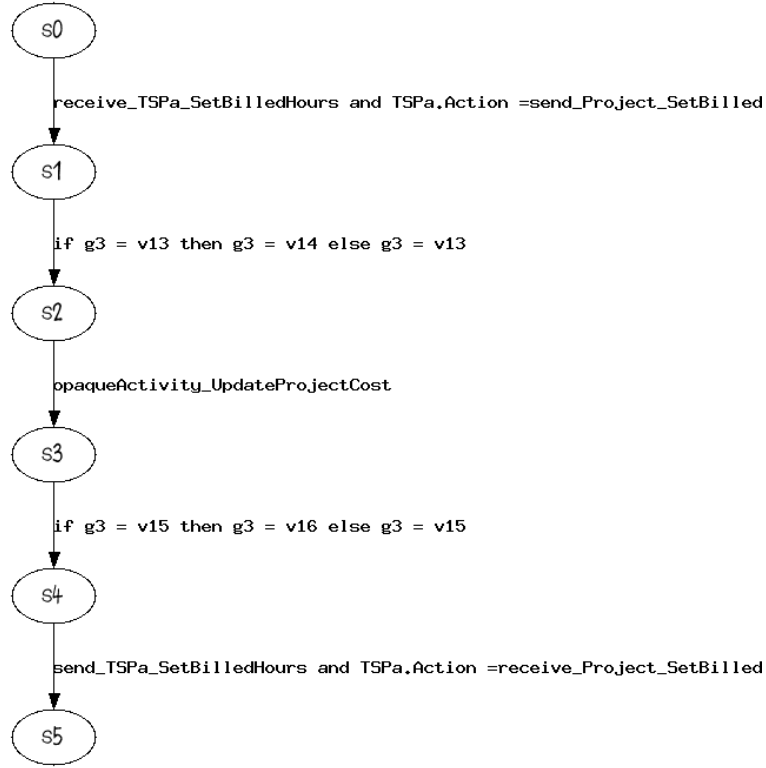


Figure A.6: Project agent automaton

sending information. As discussed earlier, two values are used to differentiate consecutive actions in the same direction. The initial value of the shared variable $g2$ is $v0$.

The Project automaton is depicted in Figure A.6. It has one path (from $s0$ to $s5$) resulting from the *< sequence >* construct transformation. The activities nested within this construct are transformed into sequential actions as described by Algorithm 2. Each basic activity is transformed using Algorithm 1. The resulting actions are: `receive_TSPa_SetBilledhours` on the evolution from $S0$ to $S1$, `opaque_Activity_UpdateProjectCost` on the evolution from $S2$ to $S3$, and `send_TSPa_SetBilledHours` on the evolution from $S4$ to $S5$. Moreover, the communication is modeled in a similar way as for the Payroll service.

The TSP automaton is the same as the one depicted in Figure until global state 16, its extension starting at the transition from state 16 to state 31 is depicted in Figure A.7. The actions corresponding to the invocation to Payroll and Project web services

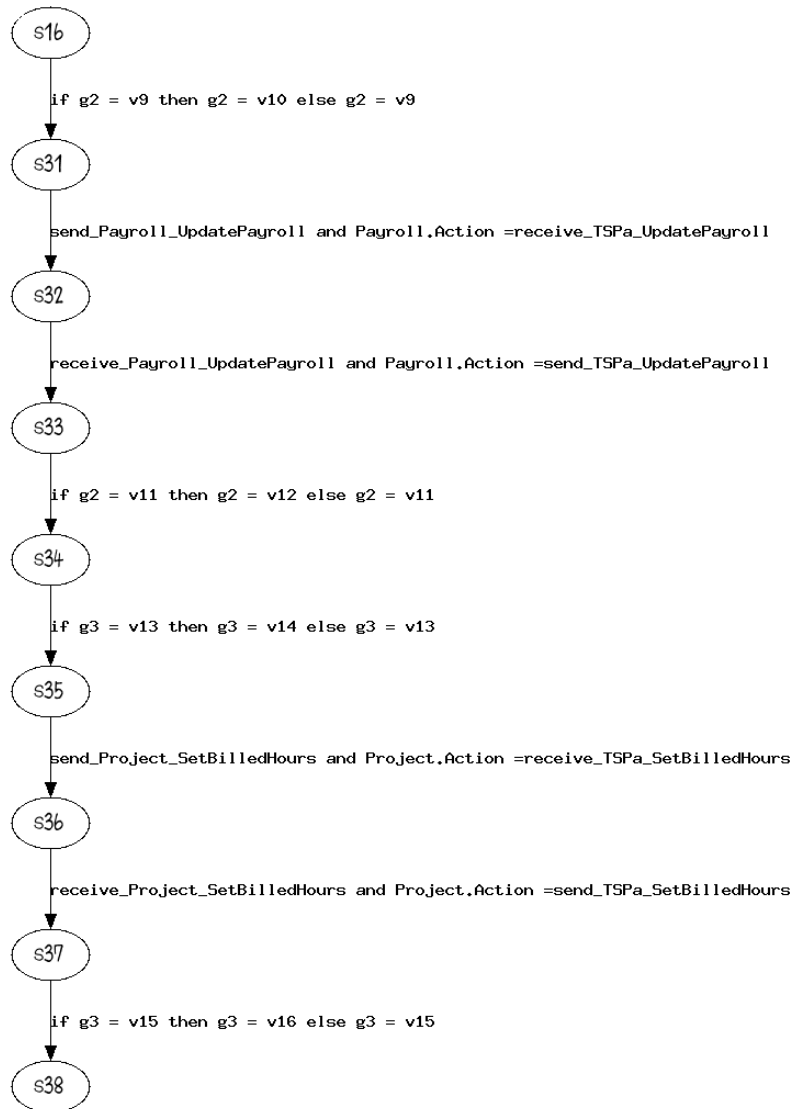


Figure A.7: TSP agent automaton

were added after the transformation of the $\langle flow \rangle$ construct. The resulting actions are: send_Payroll_UpdatePayroll on the evolution from S31 and S32, receive_Payroll_UpdatePayroll on the evolution from S32 and S33, send_Project_SetBilledHours on the evolution from S35 and S36 and receive_Project_SetBilledHours on the evolution from S36 and S37. The Fault Handler section do not have changes.

A.3 Verification Results

After being generated, the ISPL+ file is loaded into MCMAS+. Additional properties involving Payroll and Project services are stated as follows:

5. EF C(TSPa, Payroll, FinIf_Payroll)
6. EF Fu(TSPa, Payroll, FinIf_Payroll)
7. EF C(TSPa, Project, opaqueActivity_UpdateProjectCost)
8. EF Fu(TSPa, Project, opaqueActivity_UpdateProjectCost)

Property five checks if there is a possible computation where the Payroll agent commits to TSPa agent to perform FinIf. Property six checks if this commitment is fulfilled. Property seven checks if it is possible in some future run that Project agent commits to TSPa agent about AssignTotalToOutput. Property eight checks if this commitment is fulfilled. As we can see in the verification results shown in Figure A.8, all the properties hold.

Figure A.9 shows a witness for property five, which means that Payroll agent commits to TSPa agent to perform FinIf in global state twenty six. Figure A.10 shows a witness for property six, which means that the commitment established in property five is fulfilled in global state thirty one. Figure A.11 shows a witness for property seven, which means that Project agent effectively commits to TSPa agent to perform UpdateProjectCost in global

Verification result

Formula 1:	EF (C (TSPa, Invoice, opaqueActivity_DatabaseAccess))	TRUE	show counterexample/witness
Formula 2:	EF (Fu (TSPa, Invoice, opaqueActivity_DatabaseAccess))	TRUE	show counterexample/witness
Formula 3:	EF (C (TSPa, Employee, assign_AssignTotalToOutput))	TRUE	show counterexample/witness
Formula 4:	EF (Fu (TSPa, Employee, assign_AssignTotalToOutput))	TRUE	show counterexample/witness
Formula 5:	EF (C (TSPa, Payroll, FinIf_Payroll))	TRUE	show counterexample/witness
Formula 6:	EF (Fu (TSPa, Payroll, FinIf_Payroll))	TRUE	show counterexample/witness
Formula 7:	EF (C (TSPa, Project, opaqueActivity_UpdateProjectCost))	TRUE	show counterexample/witness
Formula 8:	EF (Fu (TSPa, Project, opaqueActivity_UpdateProjectCost))	TRUE	show counterexample/witness
			show BDD information

Figure A.8: Verification results

state thirty six. Figure A.12 is a witness for property eight, meaning that the commitment established in property seven is fulfilled in global state forty three.

EF (C (TSPa, Payroll, FinIf_Payroll))

Agent Payroll

Cond_0_Pa = false
g2 = v0
st_Pa = s0

----- State 23 -----

Agent TSPa

Cond_0_TS = false
Cond_1_TS = false
g0 = v3
g1 = v7
g2 = v0
g3 = v0
st_TS = s28

Agent Payroll

Cond_0_Pa = false
g2 = v0
st_Pa = s0

----- State 24 -----

Agent TSPa

Cond_0_TS = false
Cond_1_TS = false
g0 = v3
g1 = v7
g2 = v0
g3 = v0
st_TS = s30

Agent Payroll

Cond_0_Pa = false
g2 = v0
st_Pa = s0

----- State 25 -----

Agent TSPa

Cond_0_TS = false
Cond_1_TS = false
g0 = v3
g1 = v7
g2 = v0
g3 = v0
st_TS = s16

Agent Payroll

Cond_0_Pa = false
g2 = v0
st_Pa = s0

----- State 26 -----

Agent TSPa

Cond_0_TS = false
Cond_1_TS = false
g0 = v3
g1 = v7
g2 = v9
g3 = v0
st_TS = s31

Agent Payroll

Cond_0_Pa = false
g2 = v0
st_Pa = s0

☐ Environment

☒ TSPa

☐ Employee

☐ Invoice

☒ Payroll

☐ Project

Select all Unselect all

Apply

Figure A.9: Witness of the formula five

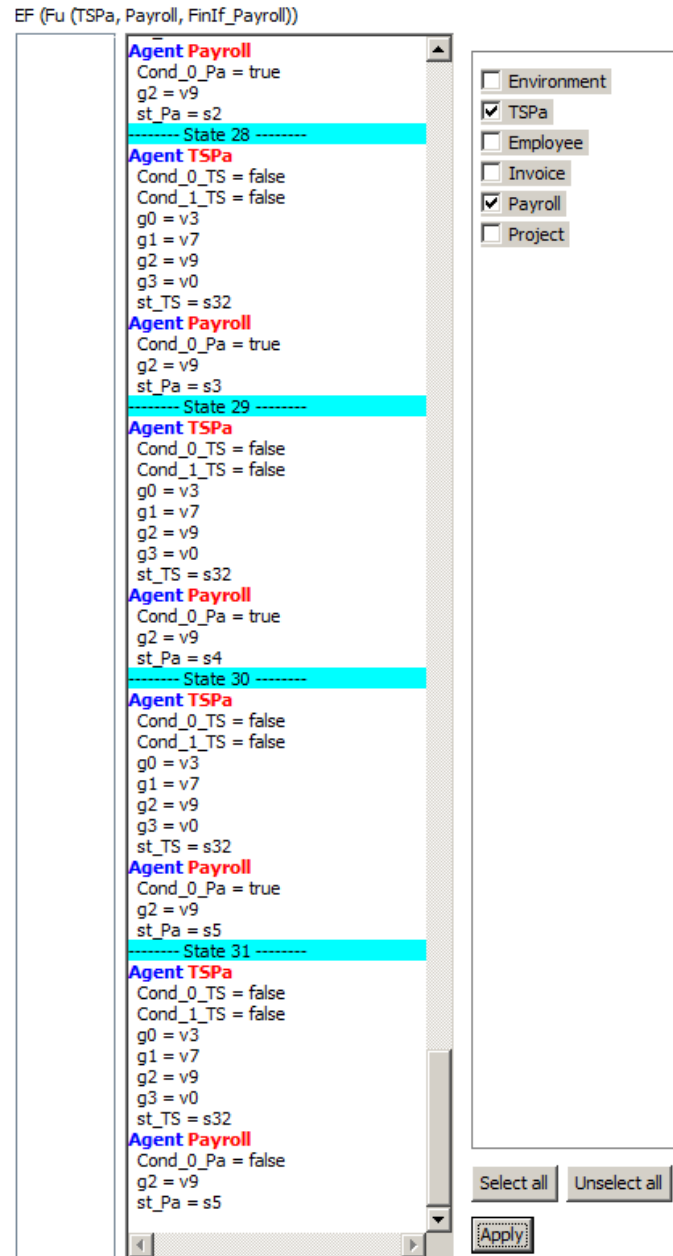


Figure A.10: Witness of the formula six

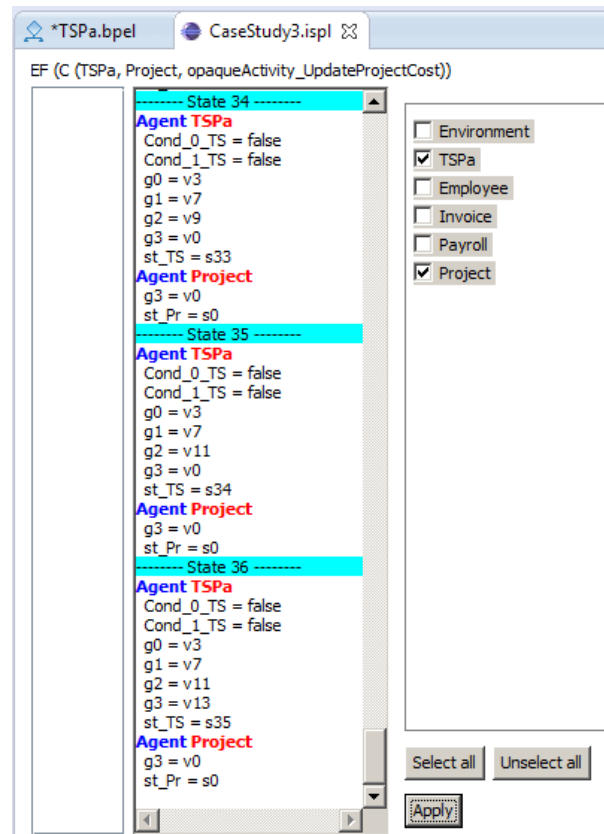


Figure A.11: Witness of the formula seven

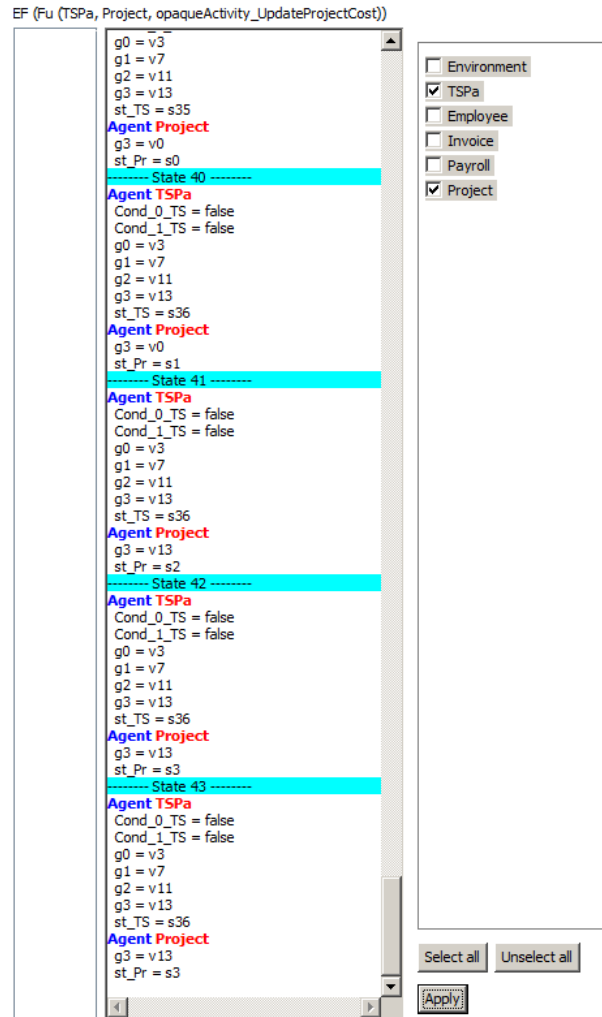


Figure A.12: Witness of the formula eight