Specifying and Verifying Contract-driven Composite Web Services: a Model Checking Approach

Ahmed Saleh Bataineh

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfilment of the Requirements for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Ahmed Bataineh

Entitled: "Specifying and Verifying Contract-driven Composite Web Services: a Model Checking Approach"

and submitted in partial fulfillment 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:

_____ Chair
Dr. M. Z. Kabir

_____ Examiner, External
Dr. J. Paquet (CSE) To the Program

_____ Examiner
Dr. A. Ben Hamza

_____ Supervisor
Dr. J. Bentahar

_____ Supervisor
Dr. Dr. R. Dssouli

Approved by: _____
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

_____20_____

_____
Dr. Amir Asif, Dean

ABSTRACT

As a promising computing paradigm in the new era of cross-enterprise e-applications, web services technology works as plugin mode to provide a value-added to applications using Service-Oriented Computing (SOC) and Service-Oriented Architecture (SOA). Verification is an important issue in this paradigm, which focuses on abstract business contracts and where services' behaviors are generally classified in terms of compliance with / violation of their contracts. However, proposed approaches fail to describe in details both compliance and violation behaviors, how the system can distinguish between them, and how the system reacts after each violation. In this context, specifying and automatically generating verification properties are challenging key issues. This thesis proposes a novel approach towards verifying the compliance with contracts regulating the composition of web services. In this approach, properties against which the system is verified are generated automatically from the composition's implementation. First, Business Process Execution Language (BPEL) that specifies actions within business processes with web services is extended to create custom activities, called labels. Those labels are used as means to represent the specifications and mark the points the developer aims to verify. A significant advantage of this labeling is the ability to target specific points in the design to be verified, which makes this verification very focused. Second, new translation rules from the extended BPEL into ISPL, the input language of the MCMAS model checker, are provided so that model checking the behavior of our contract-driven compositions is possible. The verification properties are expressed in the CTLC logic, which provides a powerful representation for modeling composition contracts using commitment-based multi-agent interactions. A detailed case study with experimental results are also reported ins the thesis.

ACKNOWLEDGEMENTS

All praises be to ALLAH Almighty who enabled me to complete this task successfully and my utmost respect to His last Prophet Mohammad (S.A.W.). This thesis would have never been completed without the will and blessing of Allah, the most gracious, the most merciful. AL HAMDU LELLAH. Next, my gratitude to my supervisors prof. Jamal Bentahar and prof. Rachida Dssouli, whose constant encouragement and guidelines at each step made this thesis possible. I am very grateful for their invaluable support, patience and kindness, for never lacking enthusiasm for research. I want to thank them for having so strongly believed in me and provided me with insights which helped me solve many of the problems I encountered in my research.

My family; Dad, Mom, Alaa', Roa'a , Mohammad and Sa'ed, I don't know how I can thank you for your constant moral support and your prayers. You are the people who are the closest to me and suffered most for my higher study abroad. Your support was invaluable in completing this thesis. My family, I love you too much.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

1.1 Context of the Research

In business applications, when the functional requirements become more and more complex, the possibility of finding a single web service satisfying the users' requests fades. Because of this, there should be a possibility to combine services together in order to fulfill complex requests, which strikes the need for the composition of web services (29). Web service composition is a fundamental task in which the process responsible for coordinating and integrating heterogeneous web services is defined. Web services and their compositions are a key technology that strongly underpins many modern applications such as cloud computing, which gained a considerable momentum over the past few years as a new computing paradigm for providing flexible services and dynamic infrastructures on demand (5; 78; 97). When services are combined, a significant challenge is to guarantee the correctness, integrity and robustness of the composition. A key issue in a such composition is to govern service interactions to accomplish the overall outcomes, particularly when the autonomy of services is a main perspective (90). Certain services may become completely or partially unable to provide the expected functionalities in the future. This imposes more challenges to guarantee the composition robustness against the future errors and that services will not be hurt by the errors consequences.

Providing rules to model and represent interactions among services and verifying their behaviors in the presence of these rules are two key problems in this setting. Service level agreements (SLAs) and contracts are two useful concepts to be considered and studied when

addressing such problems (52; 65; 67; 72). SLAs are rules describing agreed level of services that providers should supply when those services are invoked using particular parameters (14). Like in business settings, the concept of service contract is mainly used to specify obligations and permissions in a variety of circumstances including those produced when services are not performing as expected. Compared to SLAs, contracts consider and capture more sophisticated specifications such as human-like activities and legal-like agreements among services. A considerable number of proposals, specifically on formal verification of service compositions, has been provided. Although these proposals are all significant, web services are not fully mature yet and more attention to specification and verification of contract-driven compositions of web services is still needed. Currently, a huge number of web services are functional, but most of them have been deployed with dependability problems and are exhibiting unexpected behaviors because of lack of rigorous verification processes (12; 87; 63; 96).

Many tools, models and approaches have been provided to design and implement web service compositions. Workflow technologies and languages such as Business Process Execution Language (BPEL) are widely used in this domain (2; 37). BPEL can be seen as a design tool to specify and design the composition in a perspective and as an implantation tool to execute the composition in another perspective. Modeling a system using workflows, like in all composition approaches, is a challenging activity and designers are likely to introduce errors. Guaranteeing the overall correctness and robustness of the workflow processes becomes more and more important if the workflow coordinates and controls the interactions based not on abstract variables representing those interactions, but on evaluating and understanding their real contents. Verification is generally used to prove and implement this correctness, and the majority of the verification approaches adopt the model checking techniques to reason about and verify the interactions among services (see for instance (16) for a survey).

1.2 Motivations and Research Questions

As revealed in (16), many verification approaches adopt the model checking techniques to reason about and verify the interactions among services. Two key perspectives in this context have been considered: functional and structural. In the functional perspective, the attention is focused on the functionality of services described by their inputs and outputs. According to the structural perspective, each service is described in terms of its behavior, i.e., in terms of state transitions or activities performed. The resulting contributions focus on three general aspects: 1) communication models; 2) web services' functionalities; and 3) coordination of web services roles into the compositions. It is worth noting that these approaches may not provide accurate understanding of the contracts regulating web services into compositions. The reason is that those proposals are limited to represent and reason about the contracts' obligations, but do not satisfy the web developers' interests in studying the extent to which services are in compliance with their contracts and what legal remedies the composition exhibits when some of the services are breaking their contracts in certain ways. Moreover, there is lack of approaches and tools that would enable service developers to check particular aspects of the business process and whether overall requirements are met or not. In addition, verification properties are generated manually or semi-automatically either by appending them directly in the verification frameworks or modeling the contract obligations and then extracting them semi-automatically, which is an error-prone and time-consuming process.

The main research questions this thesis aims to answer are:

- How to guarantee that service compositions comply with their contracts?
- How those contracts can be specified so that their verification in terms of compliance can be possible?
- How desirable properties to be be verified can be directly and automatically extracted from executable composition of services?

- How to make the verification of composite services focused to help developers verify specific parts of the design?

1.3 Contributions

In this thesis, we propose a novel approach towards verifying compositions of web services where services interactions are regulated by binding electronic contracts. In our approach, the composition is specified using an extension of BPEL. We implement this extension by adding custom activities called labels, which we use to mark and distinguish services' desired behaviors. We also implement a compiler that takes as input the BPEL process (or its extension) and generates a model of multi-agent systems written in Interpreted Systems Programming Language (ISPL). The ISPL program is used as input to an extended version of the symbolic model checker MCMAS (55) introduced in (9). In addition to the ISPL code, the compiler automatically generates verification properties to verify the contracts details. Those properties are written in the CTLC logic that incorporates commitments, a natural tool to express contracts' components and services desired behaviors (9). In fact, the main motivation behind using the extended MCMAS model checker is the fact that it is the only model checker that fully supports CTLC, which is expressive enough to describe contract-driven composition of interactive services. The main contributions of this thesis are as follows.

First, the proposed approach introduces a new technique to represent and reflect the contract's specifications in the implementation. In the traditional techniques such as (57), the specifications are represented completely and separately from the implementation by means of any transition-based structure or software design tool. With this traditional approach, it is hard to guarantee that the specification outcome matches the implementation. Instead of this classic way, we use the composition's implementation to represent the contract's specifications. In particular, we create labels by which the specifications are marked inside the implementation of the composition. The idea comes from our point of view that the workflow technologies are

originally tools for expressing the software specifications before using them for designing the composition. In fact, this labeling process enables web developers to verify their designs at particular points. As a result, the approach allows verifying the system partially by choosing a particular part of the design, which is inline with the general observation that web developers intend to verify parts of their design first, mainly to confine the places of mistakes. Furthermore, this labeling process is the key point behind the automatic generation of the verification properties.

Second, unlike other proposals that focus only on the correctness of the composition by verifying the composition against given properties, the proposed approach goes a step further by incorporating the recovery process to verify the robustness of the composition. This has been made possible thanks to the deep analysis of all the contracts' details. Specifically, the approach pays attention not only to the violations, but also to the commitments made by services to each other. New mechanism is provided to show the recoverable and unrecoverable violations. It shows in details the reactions of the system against each violation and consequences of such a violation. Studying in depth all services' behaviors in presence of their contracts gives us the chance not only to study the recoveries, but also to measure the possibility of future errors the system can encounter. Furthermore, our approach provides a powerful framework to verify different aspects of contract-driven compositions of web services, including the communication models, messages contents and their relations to services' behaviors.

1.4 Related Work

Commitments are employed in many approaches used to define a formal semantics for agent communication languages, which successfully provide a powerful representation for modeling multi-agent interaction. Several commitment formalisms have been proposed in computer science over more than ten years (9; 30; 69; 61; 60; 11; 10; 92; 91; 7; 79; 26; 25; 24). The presentation given here follows that of (9). Our approach depends partially on the line of

distinguishing between ideal and actual (possibly incorrect) behaviors in the context of computing systems.

The notion of compliance has strongly been considered in the past few years in the area of agent computing. Several efforts have been made in this field. In particular, Giunchiglia et al. (40) dealt with persistence by specifying acceptable and disallowed transitions using an action language. In (23), the authors labeled the transitions rather than states and emphasize on the system modelling. Unlike these two approaches, ours labels the states rather than the transitions and emphasizes on efficient verification. Ägotnes and his collaborators (41) present a normative system by a subset of a transition system in terms of allowed and disallowed transitions. The authors focus on the meta logical properties of the logic and the theoretical complexity of the resulting model checking problem. In contrast, the main point of our approach is the practical verification of contract-regulated composite web services. Lomuscio and his group (57) proposed a novel approach to (semi-) automatically compile and verify contract-regulated service compositions implemented as multi-agent systems. The proposed approach uses temporal epistemic logic to verify the agents behaviors against their contracts in terms of compliance and violation. However, their formalism cannot represent in details the contracts clauses. Due to this limitation, unlike our proposal, their approach shows whether there is a recovery or not when a service breaks its contract but it cannot express which clause is violated. Moreover, the properties are generated manually by exploring the generated ISPL code. These two problems are solved in our proposed approach. More specifically, instead of separating the contract (e.g., compliant behaviors) from the composition, and then using the contract model to verify the composition implementation, in our approach we highlight the contract's specifications in the composition implementation directly using our marking process. Moreover, we differ from (57)'s proposal in the way we represent the services behaviors and the composition itself. Thus, unlike (57)'s approach that represents the composition by building a BPEL process for each participating service, we represent the composition by one BPEL process from which we automatically extract the services behaviors.

The problem of WS-BPEL modeling and verification has been studied by many researchers. Solaiman et al. (83) proposed an approach in which contracts are mapped into finite state machines (FSM). Then, the model checker SPIN has been used to verify the contracts. While this work is relevant, it uses the LTL properties to verify the contracts and focuses on the contract themselves only. Unlike our work, this proposal does not discuss the case where agents break their contracts and how the system can recover from this. Fu et al. (39) verified automatically web services compositions using SPIN. They modeled the interactions of composite web services as conversations and services as peers interacting via asynchronous messages. A compiler has been developed to translate the conversation protocol implemented by BPEL to PROMELA, the input language of the SPIN model checker. Properties expressed in LTL have been used to verify the conversation protocol. Walton (88) defined a lightweight protocol language which represents the interactions among web services in form of dialogue. In the proposed approach, the SPIN model checker has also been used to perform the verification on this language, where only LTL is being exploited. Unlike our logic that can express contracts by means of commitments, LTL can only express classic temporal protocol properties such as deadlock and reachability.

Kazhamiakin et al. (50) discussed an approach to verify the communication models in the composition. The proposal provided a parametric model to capture a hierarchy of communication models, and then used NuSMV and SPIN to perform the verification. Su, Bultan, and Fu (85), formalized the web service interaction models into a conversation concept using FIFO queues. They showed the impact of asynchronous communication on the conversation behavior. Some abstract strategies for both bottom-up and top-down design approaches are outlined. However, neither analysis nor implementation of these strategies was provided. Hull et al. (47) proposed the Mealy conversation model as a composition model to analyze the web services. Propositional Linear Temporal Logic has been used for specifying properties. The proposal technique gives the verification results in terms of bounded queues, unbounded queues and white boxes mediators. However, synchronization between behaviors and verification of com-

position design were not analyzed.

In (27), the authors used commitments to reason about the correctness of business contracts. Baldoni et al. (8) verified the conformance of agents with respect to a public protocol. However, (27) (8) did not consider CTL in their verification, which make them different from our proposal. Yeung (94) addressed the conformance problem considering only one type of behavior, namely the contract negotiation process. He introduced a formal approach using model checking as an automated means of verifying choreography conformance based on WS-CDL and WS-BPEL.

Pistore et al. (70) proposed a general framework for composition using planning and model checking. The paper follows the line of automated composition and monitoring of BPEL processes. The proposed approach exhibited a distinction with non-deterministic domains, partial observability and extended goals. Similarly, Lazovik et al. (53) presented a planning and monitoring framework to watch the execution against predefined standard business processes. The framework also considered the interactions with the users by releasing an XSRL (XML Service Request Language) request. In the same context, Huang et al. (46) generated automatically test cases for composite scenarios. Unlike (70) (53), they verified the OWL-S process. Saab et al. (15) used a formal semantics for BPEL via process algebra to study the issue of generating an appropriate client given a BPEL-based description of the interaction protocol of a web service. An algorithm deciding whether such a client exists and synthesizes the description of this client as a timed automaton was introduced. Baresi et al. (46) proposed an approach to verify the BPEL workflows at the run time. The required properties are monitored by checking the assertions defined in ALBERT, an assertion language (74). Mongiello et al. (64) used formal methods to model and formalize the correctness properties about the reliability of business process design methods. They built a framework that performs automatic verification of formal models of business processes through the NuSMV model checker. Rossetti (76) verified the composition using TCTL logic to represent specifications that will be verified on the business processes represented by the semantically annotated timed transition systems. Ouyang et al.

(50) proposed an approach emphasizing the reachability analysis and the garbage collection on queued messages by employing the capabilities of Petri nets. All these proposals are different from our work from the perspectives of 1) the used language for specification; and 2) the underlying algorithms and techniques for model checking.

Souter and Pollock (84) proposed a way to realize a structural testing methodology. The provided approach constructs contextual def-use associations, in which a context is provided for each definition and use of an object. Lu et al. (58) studied the effect of the contexts, which are environmental information relevant to an application program, in pervasive context-aware software. They provided a novel family of testing criteria which measures the comprehensiveness of their test sets. The provided approach has been tested on an RFID (Radio Frequency IDentification)-based location sensing software running on top of context-aware middleware. Yan et al. (93) considered the testing of the concurrency aspect of WS-BPEL programs. In this proposal, the WS-BPEL is modeled as a set of concurrent finite state machines, and then reachability analysis to find concurrent paths for the set of FSMs is conducted by a heuristic approach.

Foster et al. (38) proposed a model-based approach to verify web service composition implemented by BPEL. The approach traces the equivalence between the design and implementation. The message sequence charts in UML are used to model the specifications while the implementation is constructed by BPEL. The specifications and the implementation are translated into the FSP (Finite State Process) process algebra and then the LTSA (A Tool for Model-Based Verification of Web Service Compositions and Choreography) analyzes the resulting implementation.

Rouached et al.(77) used theorem proving to verify the web service conversations and their choreography. They mapped BPEL constructs representing web service composition interactions onto the EC algebra. However, theorem proving is computationally more complicated than model checking, particularly for expressive logics and the verification time is usually longer. Bentahar et al (12) used the model checking techniques, specifically, the NuSMV to

verify if composite web services design meets some desirable properties in terms of deadlock freedom, safety, and reachability. The proposed approach models the business and control aspects of web services separately using automata-based techniques, and then operational behavior is checked against properties defined in the control behavior. However, the approach focuses on the verification of sequences of interactions that the operational behavior should follow rather than the actions themselves and how the composite should behave against the invalid actions.

There is an ongoing interest in translating specifications into verification properties. Yonghua (95) proposed an approach to generate automatically verification properties by extending UML as service threat-driven model. The insecure behaviors are specified using the Probabilistic Timed Live Sequence Chart (PTLSC). However, the work focuses on time constraints and probability during service interactions. Unlike our approach, generating the properties automatically still needs a separate complete representation for services' behaviors. Also, the approach considers the undesired behaviors only not all possible behaviors. Minmin (43) formally modeled adaptive web applications via StateCharts and then extracted the proprieties from the model. Rogin (75) automatically generated the properties by describing the abstract design behaviors. Hu (45) generated automatically both verification properties and test cases by defining a standardized structure for access control models. Soeken (82) developed an approach extracting automatically the properties from the protocol specification for the formal verification of bus bridges. Unlike those initiatives, in our approach, the proprieties are automatically extracted from the actual implementation by formalizing some points without the need to model the specifications separately.

1.5 Thesis Organization

After this chapter that discusses the context of the research, motivations, contributions that this research will add to the state of the art and the literature review of relevant related work, the

rest of the thesis is organized as follows: Chapter 2 presents the background, which includes some introductory information about web services, web service compositions, BPEL, formal verification and model checking. Chapter 3 introduces and discusses our approach. First, we present our proposed verification framework and analyze the BPEL process. Second, we describe the marking of the BPEL process and the custom activities. Third, we discuss the compiler and translating rules in details. Thereafter, we describe the generated properties and their expressiveness. Chapter 4 introduces a motivating example with some typical obligations and violations of contract parties, and presents experimental results of our approach on that example. Chapter 5 concludes the thesis and identifies some relevant directions for future work.

CHAPTER 2. BACKGROUND AND LITERATURE REVIEW

2.1 Web Services

2.1.1 Service Oriented Architecture

Service oriented architecture (SOA) is a set of principles governing the design and development process of interoperable services (68). This architecture describes service-based systems as a collection of interoperable and communicating services presented by discrete software components, where each one is responsible of providing a well-defined business function. Those software components communicate together via exchanging messages to complete the desired main functionality of the software system. Web applications invoking web services are concrete examples of service-based software implemented according to the SOA principles.

Web service architecture consists of three main entities responsible of performing specific functionalities using supported protocols, standards and tools. Those entities, illustrated in **Figure 2.1**, are web service provider, web service registry and web service consumer. Web service provider is a software agent that provides the web service to be used by other software components. It has the ability to publish the web service description using web service description language (WSDL) into the web service registry. Web service consumer is a software agent that searches the web service registry to find the desired web service meeting its requirements and invokes it using simple object access protocol (SOAP).

Standardized service contracts that make services adhere to a communication agreement as defined collectively by service description documents is an important principle in SOA. Communication agreements focus on specific aspects of contract design, including the manner in

which services express their functionalities, how data types and data models are defined and how policies are asserted and attached. Although SOA simplifies the testing process by supporting the possibility of testing each service individually, this principle adds more challenges when it comes to test interactions among services. In this case, testing requires both providers and consumers on a continuous basis.

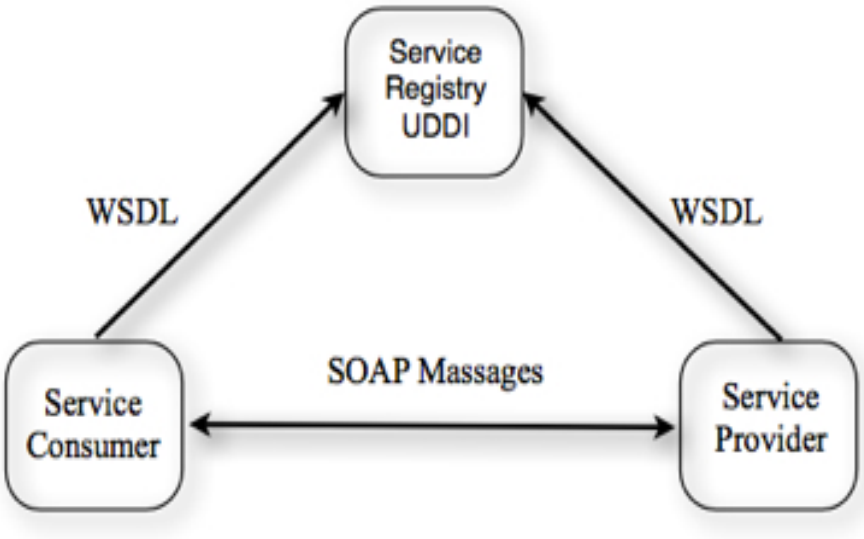


Figure 2.1: Service Oriented Architecture

2.1.2 Web Service Definition

Web service is defined as a self-contained and self-describing software model that resides on the network. It has the ability to provide a complete description about its operations, such as what the service does, the procedure through which it can be invoked and the expected results (68). It provides the necessary I/O requirements for any software application that intends to use it. Web services are built to be independent models that can be executed remotely on the server side without the need for the resources residing on the client side which call those services. Generally, they do not have a GUI as they are not meant to be used directly by the users.

The W3C working group (42) provided the following precise definition to clear the ambiguous semantics of the self-contained notion: “A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.”

In **Figure 2.2**, a purchase order application is shown (68). In this application, different web services collaborate together and finally a purchase is accomplished. First, the customer initiates the process by sending a purchase order to the application. Then, the web services responsible for the tasks of credit checking, stock checking and calculating the bills are executed concurrently. There are also some tasks that are executed sequentially such as, an item will be shipped after the billing is done. This example shows some services interacting with each other to perform a single task through the Internet.

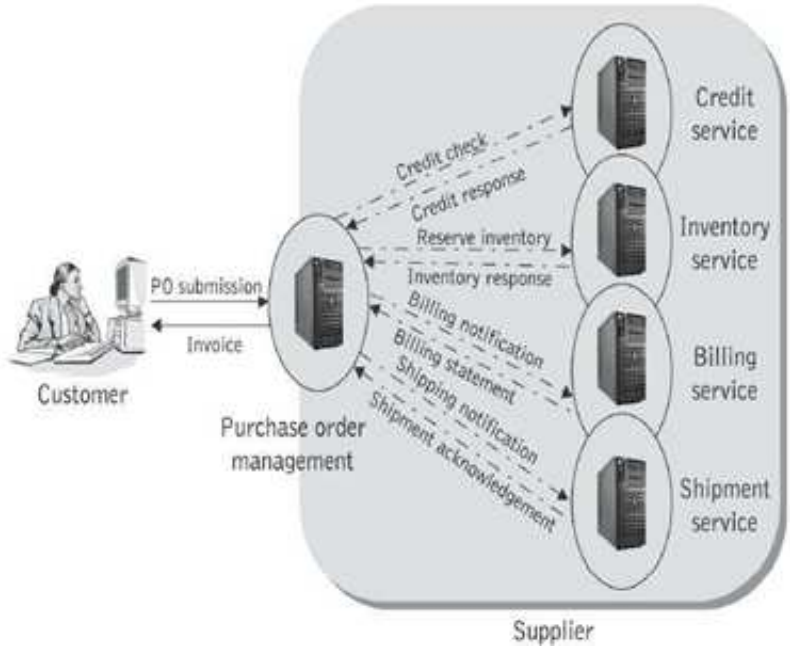


Figure 2.2: Purchase application involving interacting web services

2.1.3 Web Services Stack

Invoking web services within a software system is summarized into four major steps. Each step is carried out by one of the architecture components (68; 42) (see **Figure 2.3**). Each component communicates with its counterpart on a different machine. So, communication protocols that define the message formats, exchanging message patterns and messaging framework to guarantee the communication at each level has been built.

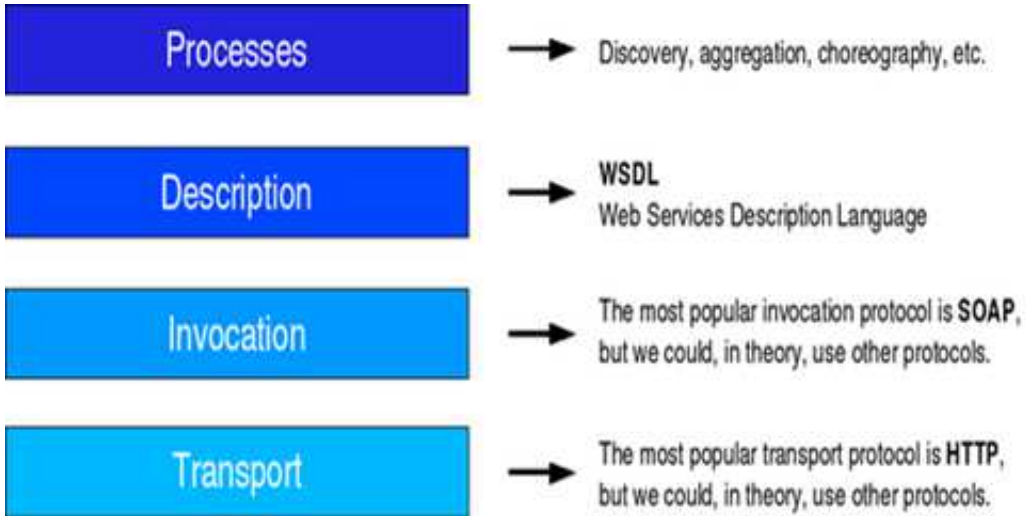


Figure 2.3: Web Service Stack

At the process level, **UDDI** stands for universal description, discovery and integration. It is an XML-based standard for describing, publishing and discovering web services. **UDDI** is the specification of web services distributed registries. Such registries are defined as a set of records or data structures, each record consists of fields in which the web service's data is embedded. Each record is limited to one web service and each field is used for a specific data. The idea behind those registries is to provide an abstract description of web services interfaces based on WSDL.

Once a web service is developed, its author (or a company developing it), starts to fill a registry with required data to publish it and make it accessible on the web. This process is known as the web service publishing. Software systems, such as web applications, requiring

web services to accomplish some tasks explore the registries to find the suitable services. This process is the web service discovery. Manipulating and exploring registries are performed through **APIs** defined in the **UDDI**.

At the description level, fully detailed description of web services is provided. The functionality of each service is clearly described using WSDL, the specialized language for describing the web services. A WSDL file is an XML file that describes the service as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The messages and communications are structurally represented via an XML grammar defined in WSDL.

WSDL is extensible; it allows extending or including customized elements needed to represent a specific technology. The extensibility property has been provided to support possible changes in the underlying protocols used in invocation and transport layers. The extended elements are implemented into name spaces outside WSDL, often in the underlying protocols. The importance of the extensibility property appears in the binding process, where there is a need to define new message formats. This property makes WSDL compatible with a large set of protocols.

At the invocation level, the actual invocation of web services is performed. Exchanging messages between web services at the invocation layer follows SOAP, one of the different technologies through which SOA has been implemented. SOAP, Simple Object Access Protocol, is a transport protocol on which web services rely to exchange their messages. It specifies how a web application formats its requests to the server and how the server should formats its response. SOAP describes the communication between two endpoints at the application level. SOAP messages are part of the actual communication between two software components on the network.

The widespread use of SOAP is a consequence of its adoption of the XML format as many free tools that support message exchange are XML-based. Moreover, many parsers which can understand and convert XML format are available.

The transport layer is responsible of the transmission of SOAP messages. The transmission process is implemented based on well-known standard transport protocols, such as HTTP, SMTP and FTP. HTTP protocol is the transport protocol in the application layer. In general, this layer manages the communication between two application processes, e.g., a web browser communicates with the web application hosted in a server. HTTP protocol has been designed to exchange the hypermedia information in distributed systems. The ultimate destination address of the application message is known via the HTTP header.

2.2 Web Service Composition

Web service composition has been introduced to meet the urgent need of combining web services functionalities in order to deliver a specific business request to the end-customers (1). A composition is a web service implemented by invoking a set of many web services. It does not matter for the client whether the web service is a composite or basic service as far as the request is answered. How the request is managed is rather a matter of implementation choice. Composing web services need to implement a workflow system to manage, organize and coordinate the roles of these services. Web service composition must follow the design standards in this field such as SOA.

2.2.1 Composition Approaches

Approaches to the web service composition problem have been exceptionally diverse and offer different interpretations of what should be addressed in a composition scenario. The approaches are classified into three groups: workflow-based, model-based, and AI planning-based. This thesis aims to verify web service compositions implemented by BPEL, so it focuses on the workflow approach while the others are briefly shown.

These groups are classified based on their degree of dynamism; they are ranged from the static to the full dynamic. In the static approaches, the services are chosen, linked together,

and finally compiled and deployed at design time. The flaw of this approach is the inability to adapt automatically with unpredictable changes. Any updates lead to unavoidable changes on the design and the architecture of the system or at least changes on the service level. In (86), Microsoft Biztalk and Bea WebLogic are introduced as an example on static composition engines. In the dynamic approaches, the services are chosen, linked together, and finally compiled and deployed run time. These approaches may include an implementation for several features such as dynamic service discovery and dynamic conversation selection. They enable the web service composition system to adapt automatically to unpredictable changes with minimal user intervention. Casati, et al.(17) introduced e-flow from HP as an implementation for dynamic composition engine.

1. **Workflow-based approaches**

   For more than twenty years, Workflow organizations have been a major research topic. Drawing mainly from the fact that a composite service is conceptually similar to the workflow, the accumulated knowledge in the workflow field is exploited to build the Web service composition. The workflow technique was one of the initial solutions proposed for web service composition. Initially, it was exploited in static and manual compositions and then, it has been extended to semi-automated and to fully automated composition. Most workflow approaches employ BPEL in one way or another (see for instance (73) for a survey).

   Recent efforts have attempted to realize automated web service composition in this category by proposing frameworks that automatically construct a workflow for the web service composition. Majithia, et al. (59) proposed a workflow generator that attempts to create an abstract workflow (written in BPEL) from a high-level objective taken as input. PAWS (3) provides a semi-automated composition framework that takes SLA (Service Level Agreement) represented in BPEL and then, by performing SLA negotiation, it attempts to find services that have the required interface and do not violate any constraint

at the same time.

2. **Model-based approaches**

   Model-based approaches have been proposed to deal with the increasing complexity of systems by raising a level of abstraction to simplify such systems. They represent web services and service composition using well-established models such as FSM. The general procedure in this category is to provide a representation for the web services and the compositions via one of the designing models, which is then translated automatically to one of the composition executable languages. Berardi et al. (13) modeled the behaviors of services using FSM transformed to DPDL (Deterministic Propositional Dynamic Logic). Skogan et al. (81) proposed an approach modeling service compositions using the UML activity diagrams. The UML diagrams are then used to generate executable BPEL processes.

3. **AI Planning approaches**

   This category employs all research efforts in AI community in order to generate a composition schema. The AI planning techniques generate series of actions starting from the initial state reaching the goal state accomplishing the service requester requirement. A large amount of composition approaches fall into this category, which forced many researchers such as Chan et al. (73) and Ghallab et al. (66) to classify these approaches into sub-categories based on the technique used to generate the composition plans.

2.2.2 Web Service Composition Models

Web services composition needs to coordinate the sequence of service invocations, manage data flow, and manage execution of compositions as transaction units. To accomplish this mission, many models adopted by the previous approaches have been defined. In the following a look at the most composition models has been taken.

1. **Orchestration Model**

   Orchestration model describes the internal behavior for the composition. It governs the workflow of composite web service via main flow control patterns which define the order in which and the conditions under which web services are invoked. Specifically, it describes how services interact at the message level, including the business logic and the sequence of interactions.

   Service orchestrations are typically described and executed using workflow languages. Dumez et al.(28) introduced UML-S activity diagram as process-modeling to design the orchestration model. The most prominent and universally adopted language for describing service orchestration is Business Process Execution Language (BPEL) (2).

2. **Choreography Model**

   Choreography models the composition by making services able to control their internal business processes, which makes it different from orchestration in which the controlling task is empowered to the environment party. Thus, choreography defines rules of interactions and agreements that occur between services or sometimes multiple business processes. The choreography is conceptually associated with the conversation protocols among services. The principal language for defining choreographies is Web Services Choreography Description Language (WS-CDL) (49).

2.3 Business Process Execution Language (BPEL)

The research discussed in this thesis is about verifying web services composition built using BPEL. We start this section by an overview about the business process management and

workflow technologies. And then, we briefly show an overview about BPEL and how it is used.

2.3.1 Business Process Management and Workflow

The standardization of business process management and workflow technology has been discussed for more than ten years (37). Several standardizations have been proposed for different aspects of business process management. WFMC (Workflow Management Coalition) is the first of these standards that separates five different interfaces of workflow systems (37; 54). Each interface addresses a particular aspect in the workflow model such as the process definition. BPMI (Business Process Management Initiative) came in 2000 as a standard for business process management. BPMI was significantly developed twice. In the first version released in 2002, the XML-based language for the specification of executable processes with web service interaction was added to generate BPML (4). The second version, called BPMN, was developed in 2004 where a visual notation for business process was provided (89).

2.3.2 BPEL Overview

Web Services Business Process Execution Language (WS-BPEL) (2) is an orchestration language that coordinates and organizes the web services participating in the composition system. The BPEL process controls and describes the interactions between the participating service by simple communication primitives and control flow constructs corresponding to parallel, sequential, and conditional execution, event and exception handling, and compensation. The interactions adopt the approach of SOA (34) and the web service paradigm (51). Each executable business process can be seen from the technical point of view as a stateful web service presenting itself as an abstract WSDL service.

BPEL supports the description of both abstract and executable service-based processes. An abstract process captures the specifications of the classes composing the service as well as the ordering of messages to be sent or received. This thesis is not interested in the descrip-

tion of abstract processes; it provides an approach for verifying the executable service-based processes. An executable process initially defines the services participating in the process, the messages exchanged, and the events and associated exception handling. Then the executable service-based process defines in order the activities' execution.

The BPEL process definition relates a number of activities. Activities are split into two groups: basic and structured. The Basic activities group contains the atomic actions which describe the communication process between the participating services such as invoke, receive, reply, wait, assign, throw, compensate, exit, and empty. Structured activities impose behavioral and execution constraints on a set of nested activities. These include: sequence, switch, pick, while, if, fault and compensation handlers.

BPEL came to life by IBM and Microsoft after the great success of BPMI.org and the open BPMS. IBM, after launching WSFL (Web Services Flow Language), and Microsoft, after developing Xlang (Web Services for Business Process Design), decided to combine their efforts into a new language, BPEL4WS (2). The OASIS (The Organization for the Advancement of Structured Information Standards) released the BPEL4WS 1.0, the first version of BPEL, as a result of cooperation between BEA Systems, IBM, Microsoft, SAP, and Siebel Systems. In April 2003, the second version of BPEL, BPEL4WS 1.1, was submitted to OASIS. Significant enhancements have been made to BPEL4WS 1.1 to create WS-BPEL 2.0. In the last version of BPEL, BPEL4People, human interactions are implemented.

2.3.3 BPEL Main Parts

1. **BPEL Process**

   Process element is the root of the BPEL document. It includes the global declaration of elements composing this process such as partnerLinks, partners, variables, correlationSets, faultHandlers, compensationHandlers, eventHandlers. The used elements are abstracted in the process while the actual workflow definition is included under these elements.

2. **Variables**

   The variables element is one of the direct children of the process. It defines the variables needed to be used in the process. This definition contains the types of these variables, and the initial value for each variable. The variables types are WSDL messageType, XML Schema simple type, and XML Schema element. The exchanged messages between the services in the process are variables defined with WSDL messageType and considered as input/output of the partners or as fault variables of the invoke, reply, receive, and throw constructs. The variables' values are changed in the process by the assign or receive activities. In the verification process, in general, the variables capture the state of the interactions during the whole business process.

3. **Basic Activities**

   - Receive activity

     The receive activity defines an external interface for the Web Service to wait for the input messages. In the definition of the receive activity, the incoming input message must match the portType and the operation defined in the interface. Each service in the business process can have maximum one receive activity.

   - Reply activity

     The reply activity defines an output interface for the web service, through which the messages are sent. The reply activity is used in two cases: normal and fault cases. The normal case is when the output message is the actual message sent from the partner; in this case the variable attribute must match the declared response message type for the operation. In the second case, the sent message is a response for the fault. The second case of the reply activity can be distinguished by presenting the faultName attribute and matching the variable attribute for the message type of this fault's variable.

   - Invoke activity

The invoke construct is used to invoke a partner web service. The web service invocation needs to define previously the partner link, port type and operation and optionally an input variable and an output variable. The catch construct has been added to the invoke activity to treat the possible faults raised by the partner in handling the requests. The faults are defined for a particular operation in the partner's WSDL definition, and are treated using the catchAll element. Compensation handler also has been added to the invoke activity. It rolls back the effect of the activity in case of failure.

- Throw activity

  The throw construct is used to specify optionally additional data that will be available to the fault handler.

- Wait activity

  The wait construct is used to represent a time point, delay or deadline, in BPEL, e.g., blocking the process after the timeout.

- Assign activity

  The assign activity aims to initialize or change the values of the variables. Moreover, assign allows performing simple computation by evaluating general expressions.

4. **Structured Activities**

- Sequence activity

  The sequence activity includes in its body a list of activities. These activities are executed in the order in which they are listed.

- Switch activity

  The switch construct is used to choose different behaviors depending on a Boolean condition following the order in which the behaviors are listed. The first branch

whose condition is true will be executed, while the others will be ignored.

- While activity

  The while activity is a loop activity; the activities listed inside its body are executed as long as the Boolean condition is true.

- Pick activity

  The pick activity is composed of multi-branches. Each branch waits for events to be executed. In the execution of the pick activity, the first occurring event will trigger the associated branch and the others will be skipped. The event can be either an arrival message or onAlert.

- Compensate activity

  The Compensation handler is used to recover from a possible fault if it happens. It returns back to the point before the activity started.

2.4 Formal Verification and Model Checking

After discussing web services, web service composition and BPEL, we discuss in this section some model checking techniques and formal verification methods which are needed to understand our proposal about reasoning about and verifying contract-based web service compositions. Moreover, this section briefly describes computation tree logic (CTL), which provides the main basis for the commitment concept used to reason about the interactive behaviors of autonomous services.

2.4.1 Model Checking

Model checking is a model-based verification technique using formal methods. Formal methods have the advantage of dealing with ambiguities and inconsistencies in the system specification in precise manners because of their mathematical and logical foundations. Based

on those methods, model checking aims to verify the system correctness against desired properties. In this way, model checkers (i.e., model checking tools) check whether a given model satisfies a given property or not in a systematic and rigorous way. The key idea is to examine all possible system scenarios by exploring all possible system states (6; 21). **Figure 2.4** depicts the model checking framework. As shown in this figure, model checking uses the system models and properties as inputs. The properties are resulted from formalizing the needed specifications. Model checkers display the results of verifying each encoded property. If the property is not satisfied, a counter example is provided.

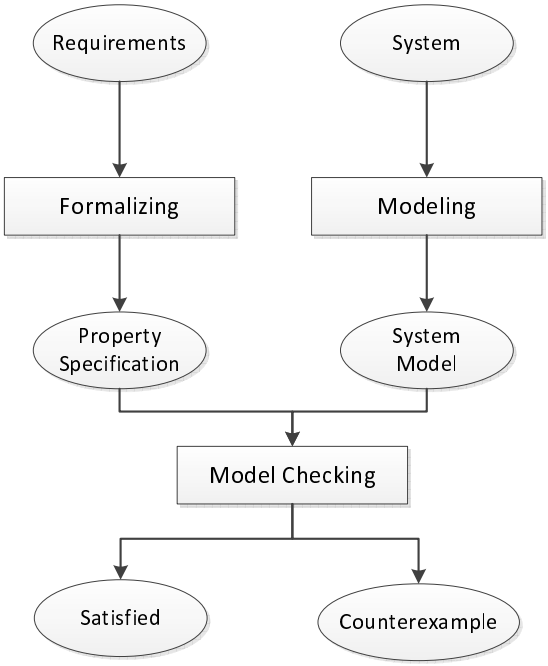


Figure 2.4: Model Checking General Framework

In the model checking approach, specifications are expressed in a temporal logic and systems are modeled as finite-state transition systems. There are two main techniques for model checking: automata-based and symbolic techniques. In the automata-based technique, both the system and specification are modeled as automata, then, the technique makes a comparison between the specification and the system to determine whether the system behavior conforms to that of the specification. This technique suffers from the state explosion problem where the global state space grows exponentially with the number of variables (19; 20). To combat

and alleviate this problem, the symbolic technique uses Ordered Binary Decision Diagrams (OBDD) and Boolean functions to encode the model and specifications (62).

The huge effort performed in the model checking field has led to the creation of new verification algorithms and tools which can verify larger models and deal with a wide variety of extended frameworks (18; 22; 19; 20; 80). Many tools are developed for this purpose such as SPIN (44), NuSMV (18), and MCMAS (55). SPIN is a model checker for LTL (Linear Temporal Logic) (71). The theoretical foundation of SPIN for the verification is based on the automata-theoretic approach. NuSMV Takes as input the finite state machines by means of declaration and instantiation mechanisms for modules and processes, corresponding to synchronous and asynchronous composition and to express a set of requirements in CTL (22) and LTL. Generally, these tools are used in the verification process to serve different purposes. MCMAS differs from the mainstream model checkers such as NuSMV, SPIN, etc., in the fact that it enables the engineer to verify not only temporal properties but also epistemic, commitment and other expressive agent-based logics. The approach proposed in this thesis to reason about and verify contract-regulated services in composition settings uses the logic of commitment, CTLC, supported in a recent extension of MCMAS (9).

Different model checking techniques and model checkers use different temporal logics for modeling the system and specifying the properties. In the following, we briefly review the logic used in our work. And in the next section, we briefly summarize MCMAS and its capabilities along with its specification language.

2.4.2 Computation Tree Logic (CTL)

Computation tree logic (CTL) is a branching temporal logic introduced in (22; 33) to specify and verify software systems. The underlying structure of time in CTL is assumed to have a branching tree-like nature, which refers to the non-deterministic behaviors in software systems where each moment in time may have several possible moments in the future. Each node in the tree has at most one preceding node. The root of the tree is a node with no predecessors

and from which all other nodes are reachable.

Definition 1 (CTL Syntax)**.**

CTL is defined by the following syntactic rules:

- *CTL Elements*
  - $AP\,(\,p\,\in AP)$ *: A set of atomic propositions.*
  - c*: Boolean operators.*
  - $\wedge$, $\rightarrow$, $\leftrightarrow$ *: Additional operators.*
  - X*(next),* U*(until),* G*(always),* F*(eventually) : Temporal Operators.*
  - A, E *: Path quantifiers.*
- *Temporal Operators and Path quantifiers*
  - Xp*: In the next,* p *is true.*
  - Gp*: Always* p*.*
  - Fp*: Eventually* p*.*
  - $p\,U\,q$ *:* p *until* q*.*
  - A *: In all paths.*
  - E *: In some paths.*
- *State formulas are assertions about the atomic propositions in the states and their branching structure.*
- *Path formulas express temporal properties of paths.*
- *Propositional atoms are state formulas, that is to say formulas evaluated on the models states.*
- *if* φ, ψ *are state formula then* $\neg\varphi,\ \varphi \vee \psi$ *are state formulas.*

- *if φ is a path formula then $E\varphi,\ A\varphi$ are state formulas.*
- *if φ, ψ are state formulas then $X\varphi,\ \varphi U \psi$ are path formulas.*
- *A, E are immediately followed by a single one of the temporal operators.*
  - $AFp = A(true\ U\ p)$ *: For all paths, p holds at some point in the future.*
  - $AGp = \neg E(true\ U\ \neg p)$ *: For all paths, p holds globally.*
  - $EFp = E(true\ U\ p)$ *: There exists a path such that p holds at some future point.*
  - $EGp = \neg A(true\ U\ \neg p)$ *: In some paths, p holds globally.*
  - $EFp = E(true\ U\ p)$ *: In some paths, p holds at some point in the future.*
  - $AXp = \neg EX\neg p$ *: For all paths, in the next state p holds.*
  - $AGp = \neg EF\neg p$

Definition 2 (CTL Semantics)**.**

- *s : State*
- *S : States where $s \in S$.*
- *σ : Path, an infinite sequence of states and transitions.*
- *$Label(s) : S \rightarrow 2^{AP}$: An interpretation function, which assigns a set of atomic propositions to each state $s \in S$.*
- *σ_i : A path starting at a given state s_i.*
- *$Path(s)$: The set of all paths emanating from a given state s.*
- $s \models p$ *iff* $p \in label(s)$
- $s \models \neg\varphi$ *iff not* $(s \models \varphi)$
- $s \models \varphi \vee \psi$ *iff* $(s \models \varphi)$ *or* $(s \models \psi)$

- $s \models E\varphi$ *iff* $\sigma \models \varphi$ *for some* $\sigma \in Paths(s)$
- $s \models A\varphi$ *iff* $\sigma \models \varphi$ *for all* $\sigma \in Paths(s)$
- $s \models X\varphi$ *iff* $\sigma_1 \models \varphi$
- $\sigma \models \varphi U \psi$ *iff* $\exists j \geq 0.s.t.(\sigma_j \models \psi$ *and* $\forall k, 0 \leq k < j(\sigma_k \models \varphi))$

2.4.3 Interpreted Systems

The interpreted system model is introduced in (35) as a formal framework to model and reason about multi-agent systems (MASs). This formalism addresses the two main classes of MASs, namely synchronous and asynchronous. It also provides tools to represent and reason about knowledge and temporal properties. The formalism of interpreted systems is defined as follows (36):

- $\mathcal{A} = \{1, \ldots, n\}$: a set of n agents forming the system.
- $L_i = \{l_1, l_2, \ldots, l_m\}$: a set of local states for agent $i \in \mathcal{A}$. Intuitively, each local state of an agent represents the complete information about the system that the agent has his disposal at a given moment.
- $g = \{l_1, l_2, \ldots, l_n\}$: where g is a global state; g is defined by a tuple consisting of a local state from each agent participating in the system.
- $G = \{g_1, g_2, \ldots, g_m\}$: is a set of global states. The set of all global states $G \subseteq L_1 \times L_2 \times \cdots \times L_n$ is a non-empty subset of the Cartesian product of all local states of n agents.
- $l_i(g)$: is a notation to represent the local state of agent i in the global state G.
- $I \subseteq G$: is the set of initial global states for the system
- Act_i : set of local actions associated with agent i.

- $P_i : L \rightarrow 2^{Act_i}$: where P_i is local protocol which gives the set of enabled actions that may be performed by i in a given local state.

- $\tau : G \times ACT \rightarrow G$: where τ is global evolution function, $ACT = Act_1 \times \cdots \times Act_n$ and each component $a \in ACT$ is a *joint action*,which is a tuple of actions (one for each agent).

- $\tau_i : L_i \times Act_i \rightarrow L_i$: where τ_i is an evolution function that determines the transitions for an individual agent i between his local states.

Bentahar et al. (9) extended the formalism of interpreted systems to explicitly capture the communication between agents that occurs during the execution of MASs. This extension is as follows:

- Var_i: a set of n local Boolean variables which represent the communication channels through which messages are sent and received (each agent has one communication channel with each other agent).

- $l_i^x(g)$: a notation which denotes the value of a variable x in the set Var_i at local state $l_i(g)$.

- $l_i(g) = l_i(g')$ iff $l_i^x(g) = l_i^x(g')$ for all $x \in Var_i$.

- Agents i and j can communicate if the following conditions are satisfied:

  - $|Var_i \cap Var_j| = 1$.
  - $l_i^x(g) = l_j^x(g')$.

In this extended version of interpreted systems, the concept of model is defined as follows:

Definition 3. *(Model). A model* $\mathcal{M} = (S, I, R_t, \{\sim_{i \rightarrow j} \ | (i,j) \in \mathcal{A}^2\}, \mathcal{V})$ *that belongs to the set of all models* $\mathbb{M}$ *is a tuple, where:*

- $S \subseteq L_1 \times \cdots \times L_n$ *is the 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 joint action* $(a_1, \ldots, a_n) \in ACT$ *such that* $\tau(s, a_1, \ldots, a_n) = s'$.
- *For each pair* $(i, j) \in \mathcal{A}^2$, $\sim_{i \to j} \subseteq S \times S$ *is the social accessibility relation defined by* $s \sim_{i \to j} s'$ *iff the following conditions are satisfied*
  - $l_i(s) = l_i(s')$.
  - $Var_i \cap Var_j \neq \emptyset$ *such that* $\forall x \in Var_i \cap Var_j$ *we have* $l_i^x(s) = l_j^x(s')$.
  - $\forall y \in Var_j - Var_i$ *we have* $l_j^y(s) = l_j^y(s')$.
- $\mathcal{V} : S \to 2^{\Phi_p}$ *is a valuation function where* Φ_p *is a set of atomic propositions.*

The intuition of the social accessibility relation $\sim_{i \to j}$ from one global state s to another global state s' ($s \sim_{i \to j} s'$) is that there is a communication channel (shared variable) between i and j. Agent i fills the channel in s, and agent j receives the information (i.e., the channel's contents) in s'. After receiving the information, all the shared variables between i and j will have the same values (i.e., $l_i^x(s) = l_j^x(s')\ \forall x \in Var_i \cap Var_j$) and the values of the unshared variables for agent j remain the same (i.e., $l_j^y(s) = l_j^y(s')$) (9; 32). **Figure 2.5** illustrates an example of this relation where two agents i and j are communicating and their shared and unshared variables are as follows. Agent i: $Var_i = \{x_1, x_2\}$, Agent j: $Var_j = \{x_1, x_3\}$ where x_1 represents the shared variable and x_2 and x_3 represent the unshared variables. After establishing the communication channel, the value of the shared variable for agent j at state s' becomes equal the value of x_1 for agent i. However, the value of the unshared variable x_3 is not changed.

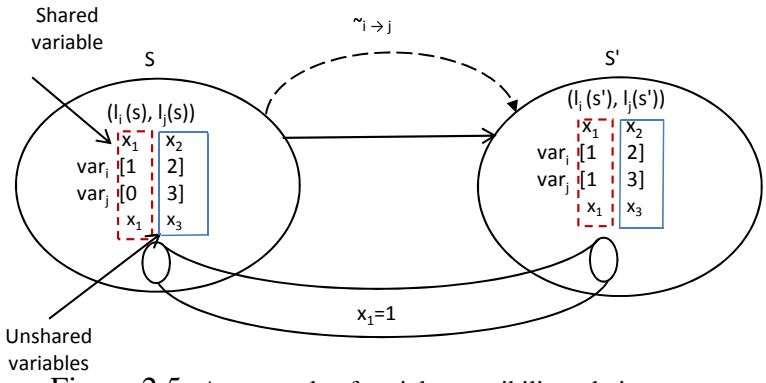


Figure 2.5: An example of social accessibility relation $\sim_{i \to j}$

2.4.4 CTLC

As mentioned earlier in this thesis, we use the expressive CTLC logic, supported in an extended version of MCMAS (9), to express the properties to be verified for the contract-driven compositions of web services. The syntax of CTLC, which is a combination of branching time CTL (20) with social commitments, is defined as follows (9):

Definition 4 (Syntax of CTLC)**.**

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

where:

- $p \in \Phi_p$ *is an atomic proposition.*
- φ, ψ *are formulas.*
- *The boolean connectives* $\neg$ *and* $\vee$ *are defined in the usual way.*
- E *is the existential quantifier on paths.*
- X, U, *and* G *are CTL path modal connectives standing for "next", "until", and "globally" respectively.*
- *The modal connective* $C_{i \to j}$ *stands for "commitment from* i *to* j*".*
- *The modal connective* Fu *stands for "fulfillment".*

The modal connectives $C_{i\to j}\varphi$ and $Fu(C_{i\to j}\varphi)$ stand for commitment and fulfillment of commitment, respectively. $C_{i\to j}\varphi$ is read as agent i commits towards agent j that φ, or equivalently from communication perspective as i is conveying information φ to j, or simply as φ is committed to when i and j are understood from the context. $Fu(C_{i\to j}\varphi)$ is read as the commitment $C_{i\to j}\varphi$ is fulfilled. Other temporal modalities, e.g., F (future), and the universal path quantifier A can be defined in terms of the above as usual (see for example (20)).

Definition 5 (Semantics of CTLC from (9))**.**

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

In this semantics, the state formula $C_{i\to j}\varphi$ is satisfied in the model $\mathcal{M}$ at s iff the content φ holds in every accessible state s' obtained by the social accessibility relation $\sim_{i\to j}$. The state formula $Fu(C_{i\to j}\varphi)$ is satisfied in the model $\mathcal{M}$ at s iff there exists a state s' satisfying the commitment and s is accessible from s' by the social accessibility relation $\sim_{i\to j}$.

2.5 MCMAS: a Model Checker for Multi-Agent Systems

MCMAS is a fully symbolic model checker for Multi-Agent Systems (MAS) that uses Ordered Binary Decision Diagrams (OBDD) (55). As a model checker, MCMAS takes two inputs: a model description for the system to be verified and a set of specifications against which the system is to be checked. MCMAS can verify a variety of properties specified by different logics such as CTL, CTLC (9; 31), and CTLK, the extension of CTL with the knowledge operator (36). The inputs of MCMAS are formatted by the ISPL language which is used to describe the MAS to be checked and encode the desired specifications. The ISPL is a dedicated programming language for interpreted systems that formalize MASs (35). MCMAS automatically evaluates the truth value of the encoded specifications and produces counterexamples which can be analyzed graphically for false specifications. MCMAS can also provide witness executions for the satisfied specifications and graphical interactive simulations.

2.5.1 ISPL General Structure

An ISPL code has the following format:

```
Agent Environment
Obsvars:
...
end Obsvars
Vars:
...
end Vars
RedStates:
...
end RedStates
Actions = ...;
```

```
Protocol:
...
end Protocol
Evolution:
...
end Evolution
end Agent
Agent TestAgent
Lobsvars = ...;
Vars:
...
end Vars
RedStates:
...
end RedStates
Actions = ...;
Protocol:
...
end Protocol
Evolution:
...
end Evolution
end Agent
Evaluation
...
end Evaluation
InitStates
```

```
...
end InitStates
Fairness
...
end Fairness
Formulae
...
end Formulae
```

The following elements form the ISPL code:

- Set of agents: One of these agents is a special agent representing the environment in which the agents interact together. The environment agent have a role in coordinating and organizing the agents interactions and it sometimes intervenes and govern the business itself. The rest are normal agents.

- Each agent has a set of local variables defined in the variables block, `Vars`. The local variables cannot be observed by the other agents, but this is a little different for the environment agent where some of these variables can be observed by the other agents. The observed variables block, `Vars`, is defined only in environment agent. The local states of each agent composed of a valuation of its local variables.

- The set of red states is used to check the correct behavior properties. It can be formed by copying the noncompliant states into red states.

- The navigation between the states for each agent can be simulated by the actions, protocol function and evolution function.

- For the model, the initial state is described by determining the initial state for each agent. Actually, the initial state for the agent is expressing the initial values of its local variables.

- The properties which will be checked are specified in ISPL by the initial states, propositions, groups and formulas.

- All the strings in the above structure are reserved keywords except TestAgent

2.5.2 ISPL Syntax

- ***Definition of variables***

  As all programming languages, the variables are defined in ISPL by giving a name and determining its data type. Three data types are allowed in ISPL; BOOLEAN, ENUMERATION and INTEGER. The following example introduces three variables, one from each type.

```
x :  boolean;
y :  a, b, c;
z :  1 ..  4;
```

  The allowable arithmetic and logical operations can be performed over the variable based on its type. The following table explains the allowed operations for each type.

DATA TYPE	Allowed operations
ENUMERATION	Equality comparison only
INTEGER	Arithmetic operations: =, !=, <, <=, >, >=,
BOOLEAN	Bit operations: ∼, |, & ,

- ***Definition of local observable variables***

  In the following, the syntax of local observable variables for an agent in the Lobservars section is provided.

```
Lobsvars = {x, y, z};
```

Where `x` , `y` and `z` are standard variables of the environment.

- ***Definition of red states***

  Actually the red states are represented by Boolean formulas over the agents local variables. Any state that satisfies one of these formulas is consider as a red state. The following is an example of a Boolean formula defined in the red states section.

```
x = true and (!(Environment.y = 'a') or z >3).
```

  Where `x` and `z` are local variables while `y` is a local variable belongs to the Environment.

- ***Actions***

  All agents' actions are included in the action section. For example:

```
Actions = {a1, b2, c3};
```

- ***Definition of protocol function***

  The protocol function returns the allowed actions to be performed by the agent at a particular state. This can be represented by two parts: the condition and list of actions. The condition is a Boolean formula used to represent a state or a group of states. The following is an example of protocol function:

```
x = true and Environment.z < 2 :  {a1, a2};
```

  Where `a1` and `a2` are defined actions.

  For the local states in which there are no expected actions to be performed, ISPL reserves Other as a keyword to represent a null action. The following is an example for using this keyword:

```
Other :  { a1, a2, a3};
```

Where `a1, a2` and `a3` are defined actions.

- ***Definition of the evolution function:***

  This section describes how the agent can navigate between its states. A line in an evolution function consists of two parts: the left hand side (LHS) and the right hand side (RHS). The LHS contains a set of assignments over local variables for the agent. The RHS is known by the enabling condition which is represented by a Boolean formula over local variables, observable values of the environment and actions of all agents. The following is an example of a line in the evolution function:

```
x=true and z=Environment.z+1
if y=b and TestAgent.Action=a1;
```

- ***Definition of the initial states***

  The initial states are defined by a Boolean formula which contains only assignments over the agent's variables. The following example shows how to define an initial state:

```
Environment.x = false
and TestAgent.x = true
and Environment.y = a
and TestAgent.z = 1;
```

- ***Definition of fairness formulae***

  The fairness formula is a type of formula to be checked in the model. It means an existence of fair scheduling of the execution of processes. Fairness is concerned with a fair resolution. The fairness formula, like the other formulas, is a Boolean formula over atomic proposition. The fairness formula can be distinguished from other formulas by the " $\rightarrow$ " operator.

- ***Definition of formulae to be checked***

  A formula to be verified is defined over atomic propositions. It can have one of the following forms:

```
 Formula ::= Formula
| Formula and Formula
| Formula or Formula
| !  Formula
| AG (Formula)
| EG (Formula)
| AX (Formula)
| EX (Formula)
| AF (Formula)
| EF (Formula)
| A (Formula U Formula)
| E (Formula U Formula)
| AtomicProposition
| C(i,j,commitment content)
| Fu(i,j,commitment content)
```

CHAPTER 3. PROPOSED APPROACH

3.1 General Overview

This section presents the framework, illustrated in **Figure 3.1**, proposed for the verification of contract-regulated MASs implementing web services. It takes a BPEL process representing the composition as an input to be verified. The BPEL process controls and coordinates the participating web services in the composition according to the business process composed of service contracts, which are usually expressed in natural languages. The BPEL process acts as the environment which implements the business process.

From the contract that specifies desired and undesired behaviors, our framework provides the option to mark the BPEL process manually using BPEL custom activities, called *labels*. This process is proposed as an untraditional mean to highlight the contract clauses in BPEL instead of representing those clauses separately. It accomplishes the following functionalities in the proposed approach:

- Distinguishing between the types of services behaviors.
- Highlighting the contracts details.
- Making the automatic process of generating the verification properties possible.

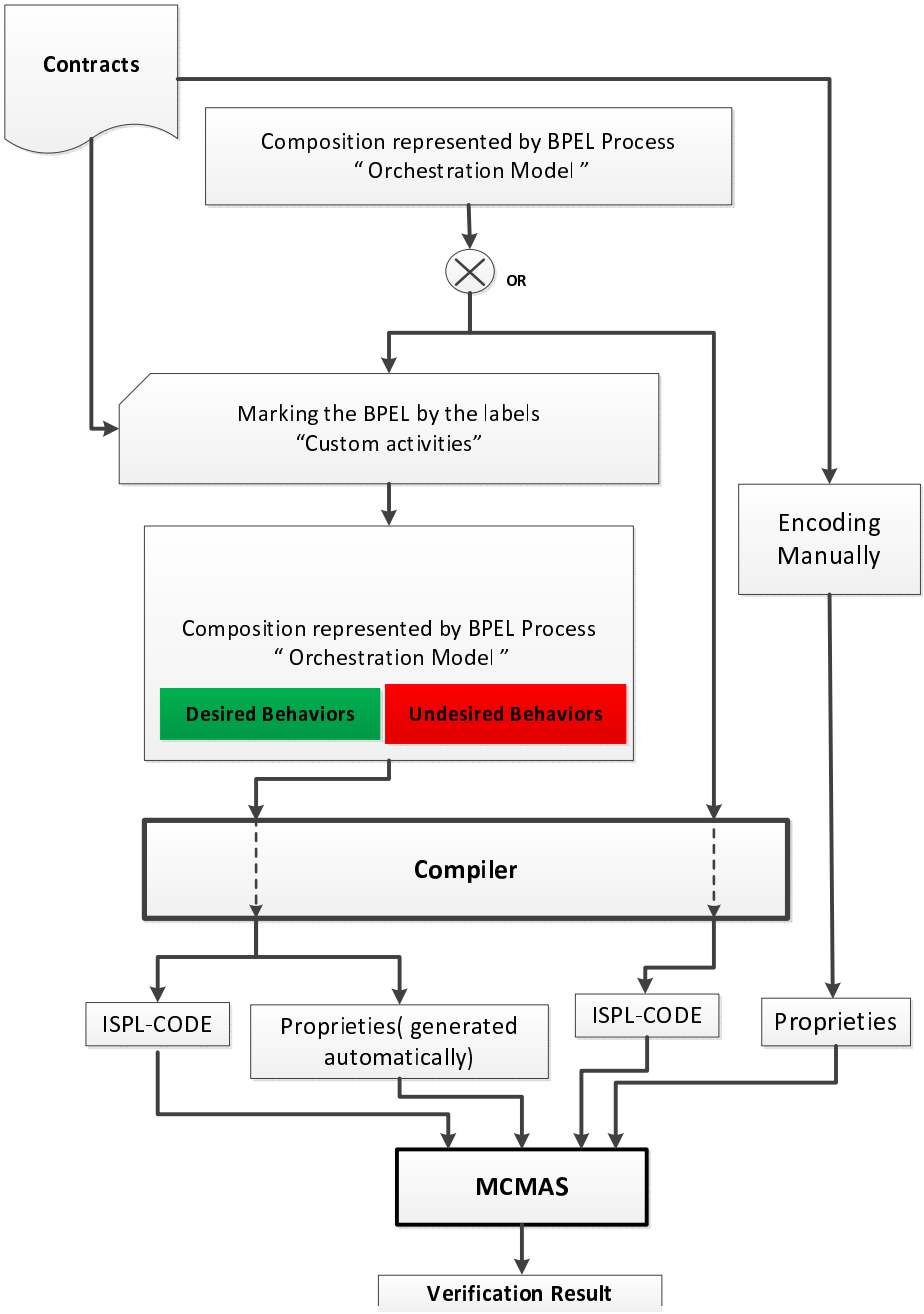


Figure 3.1: Verification Architecture

The compiler, the key ingredient in the proposed framework, takes the marked BPEL process as an input and produces an ISPL program to be used in the MCMAS model checker. Considering the CTLC logic used in expressing the verification properties, well-defined trans-

lation rules from BPEL to ISPL are implemented into the compiler. In addition to the ISPL code that encodes the system, the compiler generates verification properties.

The main objective of the proposed approach is verifying the robustness and correctness of the composition. Specifically, the aim of the generated verification properties is to analyze the following:

- The extent to which the overall system meets the desired outcomes of the composition by verifying the contract's details. In case of failure, the purpose is to determine services causing the violations.
- The ability of the system to understand or evaluate the services' behaviors whether they are in compliance or in violation with their contracts.
- The violations which the system can treat, the extent to which the system can prevent the consequences of these violations, and how the violations can be recovered from.

3.2 Analyzing BPEL Process in terms of Contracts and Service Behaviors

In our approach, the composition system is implemented using the orchestration model where the role of services is limited to receiving and sending messages and the business process is implemented as part of the environment. In the orchestration model, web services interact in the composition as black boxes where the main focus is on the control and coordination of services participating in the composition system. Using the orchestration model in our approach for implementing the composition has the following advantages. First, the Cartesian product of services states needed for the composition is reduced by implementing the business process into a centralized unit, which forms the environment. Second, unlike the approaches modeling the composition by explicitly representing services as peers in addition to the interaction protocol using the choreography model, the orchestration focuses only on the

composition protocol, removing thus any replication from the representation of the composition. Moreover, web developers usually aim to verify a particular part in the system and show the involved states of services in that part. The orchestration model represents the system as one agent moving from one state to another, and therefore the part to be verified can be easily determined by selecting two points only, namely starting and ending points in the BPEL process representing the environment. In contrast, in the chorography model, there is a need to select a starting and ending points for each single service to determine this part, and there is no guarantee that these points are correctly selected in the sense that they are relevant to that particular part.

An orchestrated business process is an activity or a set of activities that will accomplish a specific organizational goal. Many useful concepts were proposed to describe business processes; contract is one of them coming from human societies. The contract regulates the interactions between services to reach the desired goals of the business process. Providing a comprehensive analysis of composition design and verification outcomes needs a deep understanding of the key components of the contract and how the contract describes the system. Clarity of the contract depends on the power of its representation. The proposed approach uses an expressive logic, CTLC, to specify the contract in terms of commitments, a natural choice to describe contract clauses. Each commitment is representing one clause in the contract between two services where one of those services is committed towards the other to send a particular message with a particular content. If a commitment is violated, the contract provides a description how the system should react to recover from this situation.

The *contract details* are described as follows:

1. **Debtor**: the service committing to another service and acting as sender of messages.

2. **Creditor**: the service to which the debtor is committed and acts as receiver of messages.

3. **Commitment content**: the content of the message sent by debtor and its content.

4. **Condition of the commitment**: the action which causes the commitment to be activated (i.e., launched).

5. **Violation conditions**: all the cases that are considered as breaking of a commitment whether it is the sender did not send the message, unwanted message, unwanted content of the message, or breaking the environment conditions related to this commitment.

6. **Recovery**: the reaction once the commitment is broken and the service responsible to execute it.

Services show different types of behaviors during the composition with respect to their contracts. We classify these behaviors into the following three categories:

1. **Compliances**: Compliant behaviors are the set of all agreed behaviors in the contract that must be followed by the services in the system. Services behave in conformance with their contract if they follow and satisfy their commitments.

2. **Violations**: Violations are the set of arbitrary behaviors that the system can in principle engage in and they are considered as infringement of what is agreed on in the contract. For instance, a service in the system may fail to respect one of the contract clauses, or perform actions in violation with certain clauses in the contract.

3. **Recoveries**: Recoveries are the set of reactions arising once any of the contract clauses has been violated. The recoveries in general try to make the system on the track again, or protect the services' rights. Who executes the recovery and what is the action to be taken are two key attributes to describe the recovery. The self-recovery which is executed by the service breaking its contract as a try to fix its mistake, and the penalty-recovery which is executed by the environment that governs the interaction between the services to punish the service breaking its contract are two typical examples of recoveries.

Compliances and recoveries are classified as desired behaviors, while violations are undesired. Based on that, the system can engage into two types of states: undesired states that arise

when any service violates the contract; and desired states that arise when all services act without deviating from their contracts or the system recovers from the violations. In general, most of the proposals in this field focus on compliance behaviors. These are often very well defined and described with a range of formal mechanisms. Indeed, the adoption of compliances alone in the verification process of web service composition may have the ability to check whether services act without deviating from their commitments or not, and verify whether the details of commitments are correctly implemented or not, but it cannot verify the withstand of the composition against the violations and its reactions to recover from them. Verifying this significant aspect needs a deep reasoning about the violations and a knowledge on how they can be represented in the compositions.

In fact, as argued in (48), system engineers have often problems in considering undesired behaviors, which are poorly analyzed and documented. The proposed approach embeds a mechanism to consider this type of behavior. The first step is to define more accurately the undesired behaviors. Based on this definition, types of undesired behaviors are extracted and classified. The second step is to determine the expected possible undesired behaviors that the system may engage in. The final step is to observe how the undesired behaviors are represented in the BPEL process.

As mentioned earlier, the role of services into the composition is limited to exchange messages. The CTLC logic we use in our approach specifies the contract in terms of commitments conveyed as exchanged messages between interacting services. Based on that, ruling a behavior for a particular service as desired (compliance) or undesired (violation) is given by the message sent by this service. The desired behavior are the set of all agreed messages in the commitments with their agreed values that must be sent by the services in the system. Therefore, undesired behaviors can take place by two possible actions:

1. **Not sending the message**:

   This type of action occurs because of the external influences that may hinder or completely prevent the services to fulfill their commitments. For instance, failure in network

can lead to delay or not deliver the message. Although services have no control on such actions, they are unjustly inset under undesired behaviors of those services. Also, it can be added under this action the case in which a service sends a wrong message to invoke another service.

2. **Sending the message with wrong contents**:

   The possible contents of messages are a representation of the possible behaviors of services toward their commitments. Ruling a behavior for a particular service as desired or undesired is given by the contents of the messages sent by this service. Thus, our approach classifies the possible values for each defined message into two main subcategories. The first one includes all possible values that lead to fulfilling the commitment, and the second one contains all possible values that lead to violate the commitments.

Generally, web services communicate into the composition by the following two scenarios.

1. **Explicit behaviors**:

   The debtor's possible behaviors with respect to a particular commitment are represented using different types of messages. This describes the case in which it is allowed for a service to choose between different types of operations, deployed on different port types. Some of these operations are considered as fulfillment, while the others are considered as violations. The judgment on the behavior is done via the type of message not its content. Because of the non-deterministic debtor's behaviors at this moment, the pick activity is used to represent the possible messages. In such a representation, the service possible behaviors, i.e., violations and compliances, are directly extracted from the BPEL process.

   Figure 3.2 illustrates two services communicating in the composition. Delivering the software to the client triggers the commitment of payment from the client to the software provider. The client may want to change the software instead of payment,

this is considered as a violation and it should be punished by paying more as penalty. Note that the fulfillment is satisfied by invoking the payment operation via pay-decision message while the violation happens by invoking another operation via change-software message.

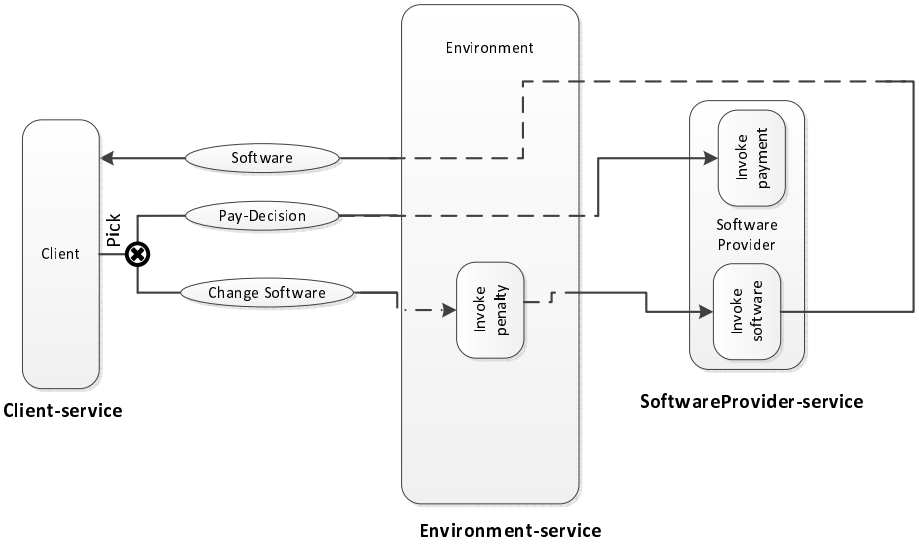


Figure 3.2: Example for an explicit behavior during the composition

2. **Implicit behaviors**

When the commitment is launched, the debtor-service sends the related message for this commitment to the environment-service. The environment-service picks up this message and transfers its content into a container variable. The environment analyzes the value held by the container, if the value is a fulfillment value, the environment delivers the message to the creditor-service, otherwise the system should enter into a recovery stage.

In the implicit representation, the debtor's possible behaviors with respect to a particular commitment are represented using multiple possible values for one message deployed on one port type. The possible service behaviors in such a representation are indirectly extracted from the BPEL process via the control operations inside the composition. It is worth noticing that most of the interactions during the composition are represented in

BPEL as implicit behaviors.

Figure 3.3 illustrates two services communicating in the composition. Delivering the software to the client triggers the commitment of payment from the client to the software provider. The client may pay or reject. The client responses are recognized in the environment by analyzing the value of pay-decision message. In case of payment, i.e., fulfillment, the environment sends the amount to the software provider. In case of rejection, i.e., violation, the environment stops the execution of the BPEL process.

The composition deals with the violations in different ways; some cases treat the violations together as one set by the same recovery, while in other cases each violation is separately handled by a different recovery. This distinction is important to extract the undesired behaviors from the recoveries.

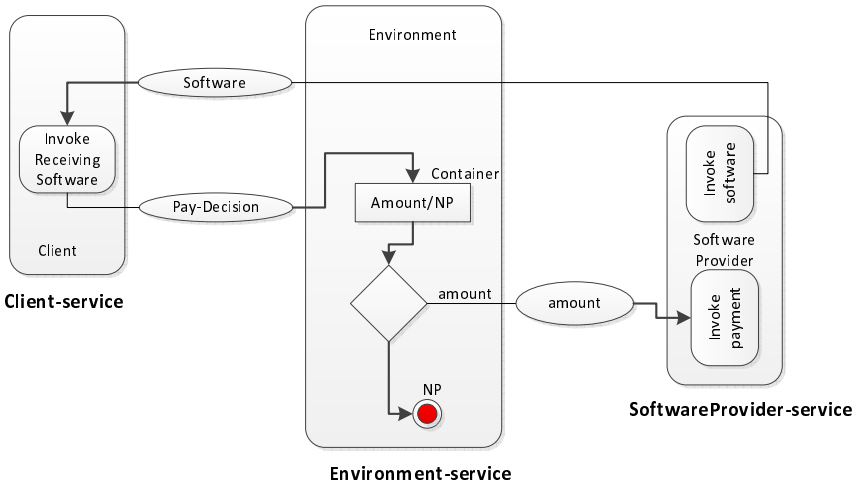


Figure 3.3: Example for an implicit behavior during the composition

3.3 Marking the BPEL Process

Specifying and generating verification properties for web service composition is still a challenging issue. As mentioned earlier, one of the contributions of this thesis is generating the verification properties automatically. As illustrated in **Figure 3.4**, most of the verification techniques use a specification language to generate the model, and then the desired verification

properties are expressed by considering this model. Undeniably, the manual specification of such properties is time-consuming and error-prone.

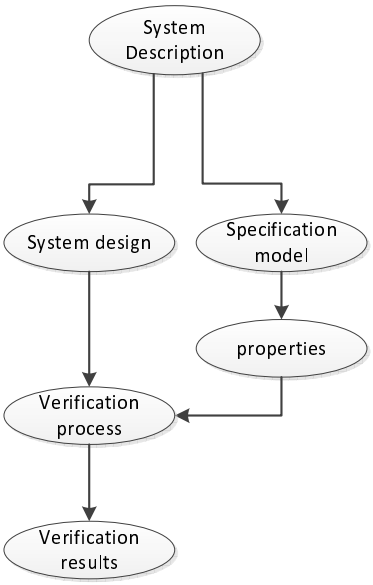


Figure 3.4: General schema for system verification

The proposed approach verifies the web service composition designed using BPEL. In our point of view, the BPEL language can be used for both specification and design of the system. As explained in **Figure 3.5**, if the specifications are modeled by BPEL, we obtain a specifications model that is included in the composition model. Note that the specifications are the desired behaviors.

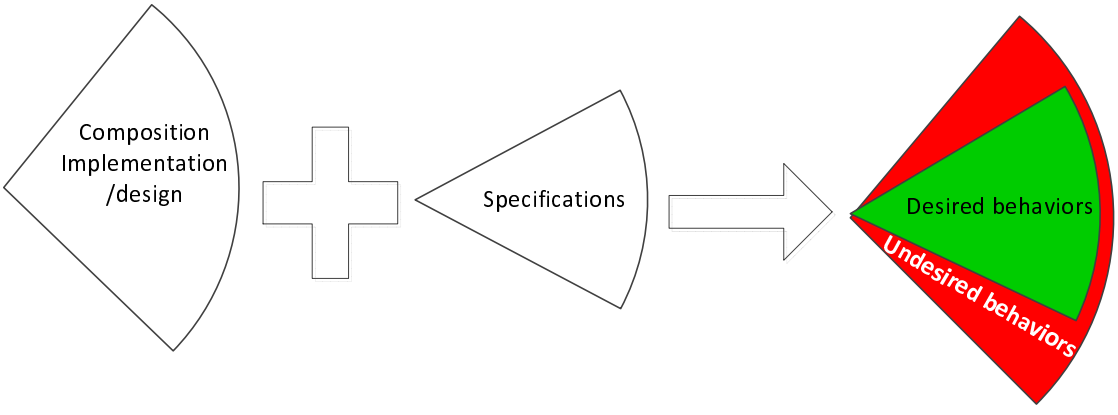


Figure 3.5: Implementation and specifications relationship

The proposed approach uses the composition design that includes all specifications to rep-

resent those instead of modeling separately. The idea is to highlight (mark) the specifications as points in the BPEL process representing the composition design. Specifically, the marking process is based on choosing manually the beginning of the contract clauses, beginning of the fulfillment and beginning of the recovers and then the compiler automatically extracts how the contract clauses has been implemented in the composition. The points are not selected arbitrarily. However, arbitrary points can easily be recognized by the incorrect syntax of generated properties. These points are marked by custom activities called labels, which have no effect on the design or execution of the process. This no-effect of the labels holds also at the automata level. If a comparison is made between two generated automata for the same system, one generated from marked BPEL and the other from unmarked one, they will have the same number of states and transitions. Thus, labeling does not exacerbate the state explosion problem. **Figure 3.6** illustrates the marking process. As shown in the figure, the contract consists of clauses. Each clause describes the commitment, violation conditions and recoveries. Based on the contracts clauses, the labels are inserted in the composition process to highlight the specifications.

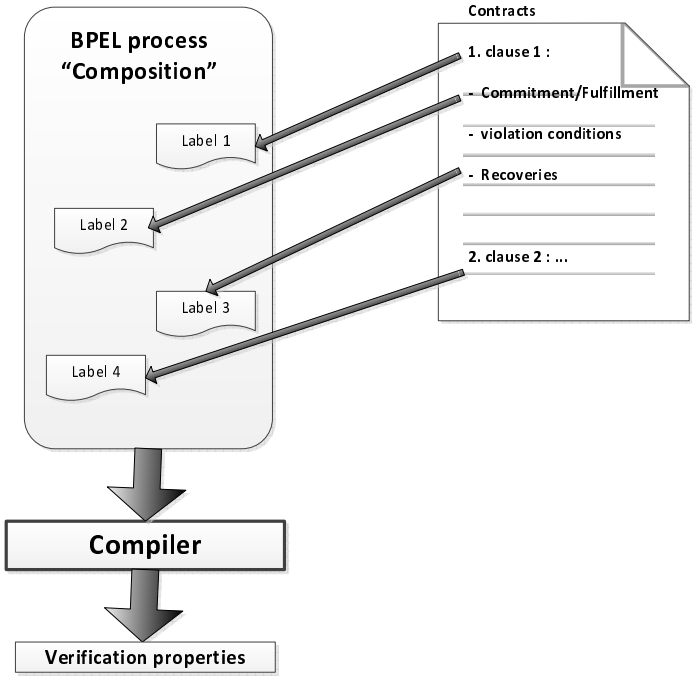


Figure 3.6: Marking the BPEL process

In fact, marking the BPEL process is not only a mechanism to represent the specifications, but it also enables the web developers to verify what the design exhibits at particular points. However, incorrect marking can negatively affect the correctness of the verification outcomes. For instance, stating incorrectly that the system recovers from a violation because of marking some undesired behaviors as desired behaviors. The following is a description of the labels and their functionalities.

1. Trace-label:

   This label is used to distinguish between the desired and undesired behaviors in the BPEL process. It has two attributes: service-name and behavior-type. The behavior-type attribute is a Boolean variable; 0 for the undesired behaviors and 1 for the desired ones. The service-name attribute catches the service that performed this behavior.

```
<bpel:extensionActivity>
    <ext:Trace-label
name="UndesiredBehaviour"
 ServiceName ="Service1"
   BehaviourType = 0 >
    </ext:Trace-label>
</bpel:extensionActivity >
```

   The number of the trace-labels equals the number of the services participating in the system as each service is associated to one label. It is worth mentioning that we do not need to label each behavior in the system to denote its type. The trace-label's value is reassigned each time the service behavior is changed from undesired to desired behavior or vies versa.

2. Str-label / end-label:

   These labels are provided to verify the system partially. Str-label is inserted at the beginning of the part to be verified and end-label is inserted at the end of this part. If these custom activities are inserted, only this part will be encoded into the ISPL code; otherwise the whole system is encoded.

```
<bpel:extensionActivity>
    <ext:str-label
   name="Strpoint" >
    </ext:str-label>
</bpel:extensionActivity>
```

```
<bpel:extensionActivity>
    <ext:end-label
   name="Endpoint" >
    </ext:end-label>
</bpel:extensionActivity>
```

3. Commitment-label:

   This label is used to mark services commitments inside the BPEL. It is not mandatory to mark all the commitments; only those related to the contract clauses the web developer wants to verify are marked. The label is inserted at the beginning of the commitment, the point at which it is thought that the commitment becomes active. Often the condition activating the commitment is a message received by the debtor-service, so the commitment-label is often inserted before the invoke-activities.

   The label consists of three attributes; debtor, creditor and the commitment-ID. Commitment-ID attribute is used as a primary key to distinguish between the selected commitments. The compiler, as explained later, uses this label to determine the parties involved in the commitment, content of the commitment, and the condition under which the commitment can be launched.

```
<bpel:extensionActivity>
  <ext:commitment-label
      name="Comm1"
    commitment-ID = 1
   debtor = "Service1"
 creditor = "Service2">
 </ext:commitment-label>
</bpel:extensionActivity>
```

4. Fulfillment-label:

This label is used to mark the points at which it is thought that the commitments are successfully fulfilled. The fulfillment takes place when the creditor-service receives the message sent by the debtor-service, so it is often inserted before the invoke-activities. The compiler uses the fulfillment-label to extract the message contents which are consider in compliance with the contract as well as the contents that are considered in violation. The label has one attribute, commitment-ID as a foreign key to interrelate it with the commitment-label.

```
<bpel:extensionActivity>
 <ext:fulfillment-label
       name="Fu1"
   commitment-ID = 1>
</ext:fulfillment-label>
</bpel:extensionActivity>
```

5. Recovery-label:

This label is used to mark the points at which it is thought that the recoveries start. The label has one attribute, commitment-ID as a foreign key.

```
<bpel:extensionActivity>
   <ext:recovery-label
       name="Rec1"
    commitment-ID = 1>
  </ext:recovery-label>
</bpel:extensionActivity>
```

Figure 3.7 illustrates an example of marking a BPEL process, in which the client-service is committed to pay for the SoftwareProvider-service if he gets the software. The client may violate his commitment by sending change-software request instead of paying. As recovery, the system forces the client to pay more as a penalty for his violation.

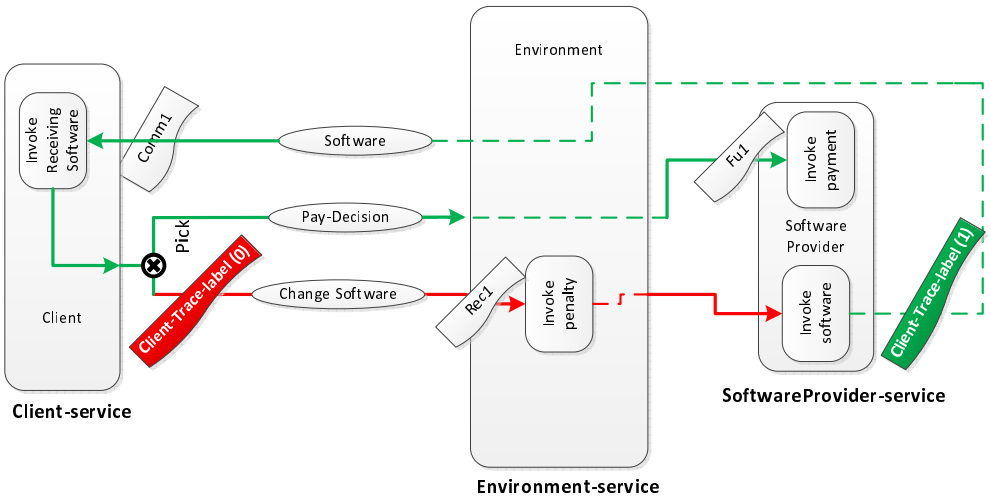


Figure 3.7: Example of marking a BPEL process

3.4 Automatic Compilation from BPEL to ISPL

The compiler is the core component in the proposed framework which automatically translates the BPEL process to ISPL-code including the CTLC verification properties. It extracts the behaviors for each participating service from the overall behavior of the composition. In the cases in which services act as black boxes or their behaviors are unknown until run time, the compiler extracts their behaviors from the control operations implemented in the composition. It generates the verification properties if the marking step has been performed. **Figure 3.8** illustrates the abstract compiler's outputs.

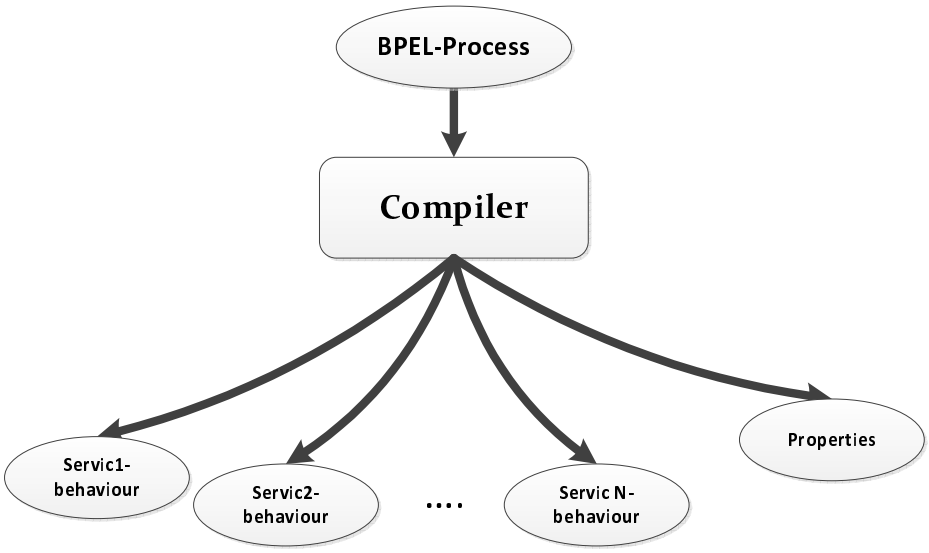


Figure 3.8: Abstract compiler's outputs

3.4.1 General Overview of the Internal Design of the Compiler

The internal architecture of the compiler is illustrated in **Figure 3.9** showing a five-step construction process to generate the corresponding ISPL code from the BPEL process. This section shows the abstract steps of this process while the technical details are given in Sections 3.4.3 and 3.4.4. In a nutshell, this process is performed as follows.

1. The BPEL process is translated into one automaton representing the composition and then it is flooded into a database.

2. Systematically, an automaton is extracted for each service from the composition automaton.

3. The generated automata are colored. Specifically, the trace-labels' values are used to color each state. The violation states are colored red while the others green.

4. The labels' positions are analyzed and then verification properties are generated. Specifically, the actions happening at the commitment-labels and the recovery-labels are formalized. Then, a set of verification properties showing the elements that the designers may wish to verify on the contract's clauses referred by these labels are generated.

5. The colored automata in addition to the proprieties are encoded in ISPL, which forms the output of our compiler.

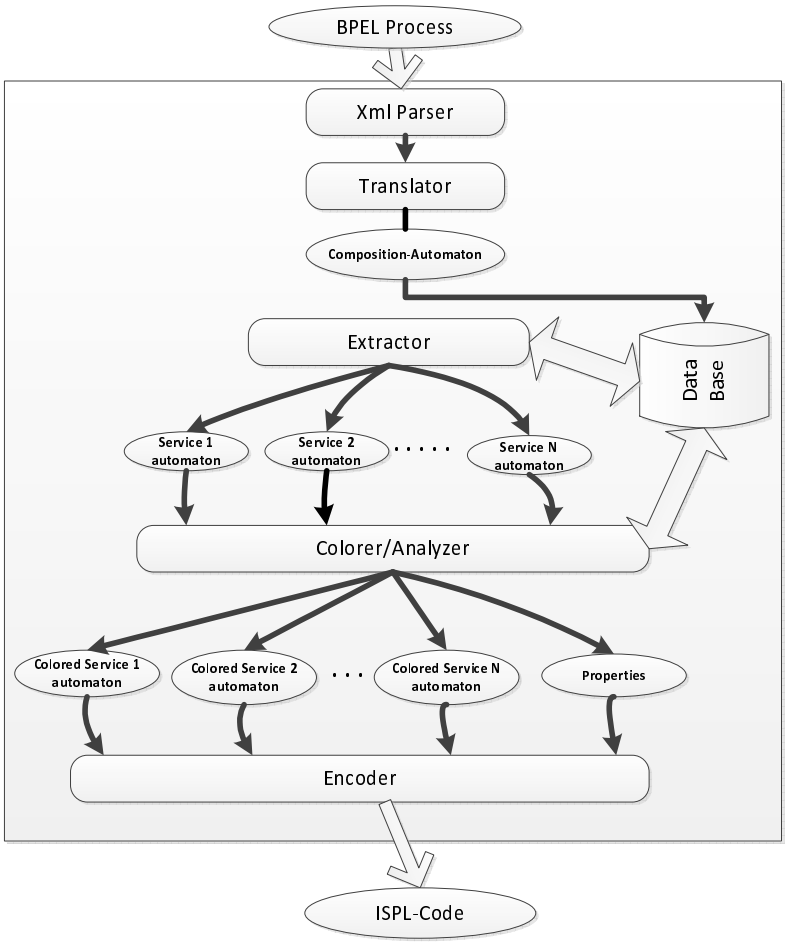


Figure 3.9: Internal design of the compiler

3.4.2 Encoding the Communication Architecture in ISPL

In the verification process, the system model must be compatible with the used logic. The CTLC logic we use in our approach captures the intuition that commitments are conveyed through communication between interacting agents. Thus, the BPEL process is translated as a set of services interacting through our communication architecture to exchange commitments. This architecture, illustrated in **Figure 3.10**, allows us to capture the intuitive semantics of social commitments as defined in CTLC by Bentahar et al. (9). Associated with each service is a set of local variables used to represent the communication buffers (each service has one com-

munication buffer for each other service it is interacting with) through which the messages are exchanged. The buffers are an implementation for the shared variables used in the semantics of social commitments (see Section 2.4.4).

The environment is implemented using a set of local variables which represent the communication channels. These channels hold the communications between the participating services. They capture the intuitive functionality of the orchestration model in controlling and coordinating the interactions between services. The number of the communication channels equals the maximum number of concurrent communications in the composition, which depends on the flow activity used to represent the concurrent communications. To hold a communication between two services, channel 1 is firstly checked, if it is busy by another communication, then channel 2 is checked and so on.

Let us consider the example of two services i and j. Service i commits to service j by establishing a connection using its shared buffer with service j, Buffer $i - j$, and the first available communication channel in the environment. The communication process between i and j is performed in two steps:

1. **First step**: service i fills its shared buffer with service j. This can be observed immediately by the environment service. Consequently, a communication channel is also filled out with the sent message in this step.

2. **Second step**: the held message is sent to service j and inserted into its shared buffer with service i. At the end of this step, the shared buffers in both services become equal.

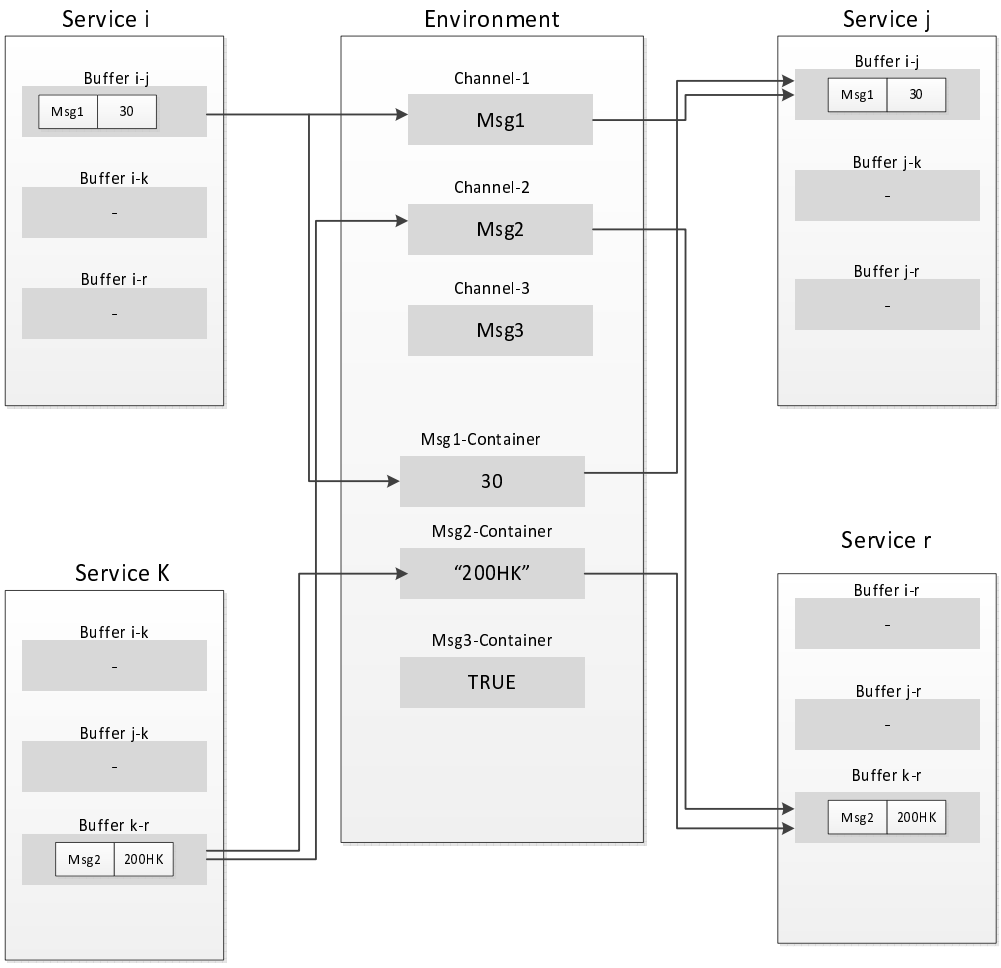


Figure 3.10: The proposed communication architecture

The social accessibility relation is successfully represented by encoding these steps as two states in the generated automata. However, modeling the communication process alone is not enough to represent the commitment; still there is a need to reason about the commitment content. As explained earlier, the commitment content is determined either by the sent message in the explicit behaviors or the message values in the implicit behaviors. Thus, it is held by the shared buffers in the explicit behaviors and by the containers in the implicit behaviors.

The containers are a set of variables catching the contents of the exchanged messages during the composition. Their values are assigned in the first step of the communication process and evaluated later to know whether they are fulfillments or violations. Each container can catch the values for one message. Because of the implicit services behaviors, the real values of

the containers can only be observed at runtime. At design time, only approximate values can be obtained from the control operations in which those containers are involved. The containers are then translated into automata with those approximate values. The technical details on how this translation is performed are given in Section 3.4.3.

The commitment content is accounted for by reasoning about the shared buffers values and the container values. As argued by Bentahar et al. in (9), the key idea is that the communication process is established to deliver the commitment content. However, unlike (9)'s proposal where shared variables are only used to indicate the existence of a communication channel between two services, in our approach, shared variables are effectively used to convey the commitment content.

3.4.3 Translation Rules

As intermediate representation between BPEL and MCMAS, we use an automata-based model in our approach. The BPEL process is translated into automata, close to the ones presented in (39). However, our approach differs from (39) by constructing the composition built over a communication infrastructure. More complicated translation rules are declared to model the communication processes especially in the case of implicit services behaviors. Furthermore, translation rules for the pick, switch and the labels are added.

In fact, the XML-parser and translator, the parts in the compiler responsible for translating the BPEL process into automata, have many implementation details such as parsing the XML text as states and loading them into the database. To shorten those details, we settle to clarify the internal design of these parts by simple BPEL examples and simple description of the algorithms implementing the translation rules. Those rules are described as follows:

- **Main Parties**

  As a first step, the participating services are extracted from the BPEL process via the partnerlinks tag. A set of variables are directly defined for each service to represent

its shared buffers as explained earlier in the communication architecture. In addition, a Boolean variable is added for each service to capture the trace-label values.

The environment includes three types of variables: communication channels, containers and operational variables. The containers and operational variables are extracted from the variables tag defined in the BPEL process. The containers differ from the operational variables in the fact that their values are assigned from service messages. In addition, a set of variables representing the communication channels are defined. The number of communication channels is extracted from the flow activities defined in the BPEL process. It equals the maximum number of branches defined in those flow activities.

- **Assign-Activities**

  Each assign-activity is composed by a list of assignments. Its translation depends on the types of variables involved in the list. Thus, it results in different transitions in automata. The followings are the proposed rules to translate the assign-activity:

  - **Message variables**

    In BPEL, the communication process between two services is represented by the assignment and invoke activities respectively. The assignment activity assigns the output message of the sender service to the input message of the receiver service and then the receiver service is invoked by the invoke activity. So, if the assignment activity assigns a variable message to another variable message such as $var1 = var2$, an initiation for a communication process between two services is considered. Such assignments represent the first step of the communication process, while the invoke activities represent the second step (see Section 3.4.2).

    The following BPEL code represents an example for a communication process between service i (the sender) and service j (the receiver). The communication process code starts from line 9 while the first eight lines are definitions of the mes-

sages. As explained from line 9 to line 14, the assignment activity assigns $var2$, the output message of service i, to the input message of service j, $var1$. From line 15 to line 17, the actual transmission is performed by invoking service j.

```
Line1:  <bpel:variable name="var1"
Line2:  messageType="ns:servicej_input ">
Line3:  </bpel:variable>
Line4:  <bpel:variable name="var2"
Line5:  messageType="ns0:servicei_output ">
Line6:  </bpel:variable>
Line7:  <bpel:invoke name="Invoke_servicei"
partnerLink="servicei"
Line8:  outputVariable="var2"></bpel:invoke>
Line9:  <bpel:assign validate="no" name="assign var1">
Line10:  <bpel:copy>
Line11:  <bpel:from part="parameters" variable="var2">
Line12:  </bpel:from>
Line13:  <bpel:to part="parameters" variable="var1">
Line14:  </bpel:to> </bpel:copy> </bpel:assign>
Line15:  <bpel:invoke name="Invoke_servicej"
Line16:  partnerLink="servicej"
Line17:  InputVariable="var1"></bpel:invoke>
```

The translation of such assignments in (lines 9-14) depicts the scenario of the first step of the communication process when service i prepares itself to commit toward service j. Thus, the shared buffer in service i, buffer $i - j$, and a communication channel in the environment are filled by the sent message, $var2$ (see Section 3.4.2).

Figure 3.11.A illustrates the resulting translation of the above BPEL code. First, the source state S_0 results from the translation of the BPEL activity preceding the assign activity. It most probably holds the condition which triggers service i to send its message. The condition oftentimes is a message received at service i. According to the above code, S_0 results from the translation of Invoke_service i in lines 7-8 (translation of the invoke activity is also explained in Section 3.4.3). Second, the transition labeled by an action performed by the service i, **servicei.Action = Send-Var2**, and the target state S result from the translation of the assignment activity in lines 9-14. In state S, the shared buffer $i - j$ and channel1 in the environment are filled by the sent message, $var2$. In terms of commitments, S_0 contains the commitment condition while S_1 contains the commitment content.

In a nutshell, this rule is technically implemented in the compiler as following:

* **Determining the assignment type**: The compiler recognizes the assignment type by the variables involved in the assignment activity. If these variables are messages, the compiler considers that a communication process will take place. From the `variables attributes` in the `from-tag` and the `to-tag` (lines 11 & 13) the variables are extracted, $var1$ and $var2$. The variables are defined as messages in the BPEL process by assigning the `MessageType attribute` . Lines 1-6 are definitions for $var1$ and $var2$. Lines 2 and 5 denote that $var1$ and $var2$ are messages respectively.
* **Determining the participating services**: The participating services are also determined by the variables involved in the assignment activity. The services to which the variables belong are extracted from the `inputvariable` and `outputvariable` attributes in the invoke activities. Lines 16 and 17 denote that $var1$ is an input message for service j while lines 7 and 8 denote that $var2$ is an output message for service i. Based on this, the compiler concludes the correct buffer, the sender and the receiver.

* **Translating into automata**: After determining the sender, the receiver and the message content, the compiler creates a transition and a target state. The transition is labeled by an action performed by the sender. In the target state, the shared buffer between the sender and receiver in the sender side is filled by the sent message. Also, one of the communication channels in the environment is filled by the sent message. The translation of such assignments affects the local states for both services: the sender and environment. As explained in **Figure 3.11.A**, state S_0 and S are global sates. The local states of service i and the environment are updated in the global target state S. Regarding service i, it receives a message from service k in the source state S_0. This triggers service i to prepare for sending its message, $var2$, to service j in the target state S. In terms of commitment, the local state of service i in S_0 contains the commitment condition, while its local state in S contains the commitment content (see Section 3.4.2).

– **Containers**

The containers refer to the variables defined in the BPEL process and their values are assigned from the services messages. Specifically, the containers are the variables which are used in the control operations and based on their values the implicit services behaviors are evaluated to enable the system to react properly. In contrast, services messages are exchanged without containers in the explicit services behaviors (see Section 3.2).

In BPEL, the implicit service behavior is represented by the assignment, one of the control flow constructs, and invoke activities respectively. The assignment activity assigns the output message of the sender service to a container in the environment. Then, the environment analyzes the content held by the container to choose whether entering in recovery stage or invoking the receiver. So, if the assignment activity assigns a variable message to a container such as $var1 = var2$,

this means the message values will be checked by one of the workflow constructs. Such an assignment depicts the first step of the communication process (see Section 3.4.2).

The following BPEL code represents an example for an implicit communication process between service i (sender) and service j (receiver). The code, lines 1-7, defines three variables; two messages, $var1$ and $var2$, and an integer variable, $con1$. The communication process code starts from line 9 to line 16. As explained, the assignment activity assigns $var2$, the output message of service i, to an environment container, $con1$. In lines 17-18, the environment checks $con1$ whether it is less than 5000 or not. In case it is less, service j is invoked to deliver $con1$, lines 19-29.

```
Line1:  <bpel:variable name="var1"
Line2:  messageType="ns:servicej_input ">
Line3:  </bpel:variable>
Line4:  <bpel:variable name="var2"
Line5:  messageType="ns0:servicei_output ">
Line6:  </bpel:variable>
Line7:  <bpel:variable name="con1" type="ns3:integer">
Line8:  <bpel:invoke name="Invoke_servicei"
partnerLink="servicei"
Line9:  outputVariable="var2"></bpel:invoke>
Line10:  <bpel:assign validate="no" name="assign con1">
Line11:  <bpel:copy>
Line12:  <bpel:from part="parameters" variable="var2">
Line13:  </bpel:from>
Line14:  <bpel:to part="parameters" variable="con1">
```

```
Line15:  </bpel:to> </bpel:copy> </bpel:assign>
Line16:  <bpel:if name="checkcon1">
Line17:  <bpel:condition><![CDATA[$ con1 < 5000]]>
Line18:  </bpel:condition>
Line19:  <bpel:sequence name="Sequence1">
Line20:  <bpel:assign validate="no" name="assign var1">
Line21:  <bpel:copy>
Line22:  <bpel:from part="parameters" variable="con1">
Line23:  </bpel:from>
Line24:  <bpel:to part="parameters" variable="var1">
Line25:  </bpel:to> </bpel:copy> </bpel:assign>
Line26:  <bpel:invoke name="Invoke_servicej"
Line27:  partnerLink="servicej"
Line28:  InputVariable="var1"></bpel:invoke>
Line29:  </bpel:sequence>
Line30:  <bpel:else>
Line31:  <bpel:sequence name="Sequence1">
Line32:  ...
Line33:  </bpel:sequence>
Line34:  </bpel:else>
Line35:  </bpel:if>
```

Similar to the translation of the message variables assignments, the resulting translation from the containers assignments simulates the first step of the communication process. Thus, the shared buffer in service i, buffer $i - j$, and a communication channel in the environment are filled by the sent message, $var2$ (see Section 3.4.2). The values of the containers are not assigned until the runtime.

As explained in Section 3.2, the possible container values reflect different possible service behaviors toward a particular contract clause. Thus, the possible container contents are considered in the translation process.

Figure 3.11.C illustrates the resulting translation of the above BPEL code. First, the source state S_0 results from the translation of the invoke activity in lines 8-9. Second, the assignment in lines 10-15 is translated into two transitions and two target states, S_1 and S_2. The number of transitions equals the number of possible contents that the container may hold. The resulting transitions are labeled by actions performed by service i, **servicei.Action = Send3000** and **servicei.Action = Send7000**. The target states represent the first step of the communication process. Thus, the shared buffer between the sender and receiver at the sender side, a communication channel in the environment and the container are filled by one of the possible container contents. As explained in **Figure 3.11.C** , in S_1, the shared buffer $i - j$ in service i and channel1 in the environment are filled by the first possible content for $con1$, 3000, while in S_2, they are filled by the second possible content, 7000. The possible container contents are approximate values, they are extracted from the control operation in which the container is involved. In lines 16-18, If-activity checks the container $con1$ whether it is less than 5000 or not, so the compiler generates two random contents: one content is less than 5000, 3000, and another content is greater than 5000, 7000.

In a nutshell this rule is technically implemented in the compiler as following:

* **Determining the assignment type**: As mentioned before, the compiler recognizes the assignment type by the variables involved in the assignment activity. If the assignment assigns a message variable to a container, the compiler considers an implicit communication process will be triggered. the variables, $var2$ and $con1$, are extracted from the `variables attributes` in the `from-tag` and the `to-tag` (lines 12 & 13). Line 4 defines $var2$ as a mes-

sage and line 7 defines $con1$ as a regular integer variable. The compiler considers $con1$ as a container because of assigning $con1$ by a value from $var2$, the message variable (see Section 3.2).

* **Determining the participating services**: The participating services are also determined by the variables involved in the assignment activity. The sender is extracted by matching the message variable with the `outputVariable` attribute in the invoke activities. Line 9 denotes that $var2$ is an output message for service i. Then, the compiler traces the container during the code to catch the receiver. Delivering the container to the receiver is done by another assignment in which the container content is assigned to the input message of the receiver. The assignment-activity in lines 20-25 assigns $con1$ to $var1$. $var1$ is defined in lines 1-3 as a message. The receiver is determined by determining the service to which the input message belongs. Lines 26-28 denote that $var1$ is an input message for service j. Based on this, the compiler chooses the correct buffer.

* **Determining the container content**: After determining the sender, the receiver and the correct shared buffer, The remaining part is to determine the possible container contents by which the shared buffer is filled. The compiler catches the first control operation in which the container is involved, which comes after assigning that container. From the condition in the control-flow construct, the compiler extracts the possible contents. The compiler generates one example on each case covered by this condition. For example, suppose x is a container and the condition is $3 < x < 5$, the compiler will generate $\{2, 4, 6\}$ as possible contents. In the switch-activity, the compiler complies with the values mentioned in the cases.

* **Translating into automata**: After determining the sender, the receiver, the correct shared buffers and the message content, the compiler creates the tran-

sitions and target states. The transitions are labeled by actions performed by the sender. In the target states, the shared buffer between the sender and receiver in the sender side is filled by one of the possible container contents. Also, one of the communication channels in the environment is filled. The translation of such assignments impacts the local states of both services: the sender and environment. As explained in **Figure 3.11.C**, States S_0, S_1 and S_2 are global sates. The local states of service i and the environment are updated in the global target states S_1 and S_2. Service i receives a message from service k in the source state S_0. This triggers filling the shared buffer $i - j$ in service i by $var2$. In terms of commitment, the local state of service i in S_0 contains the commitment condition while its local states in S_1 and S_2 contain the commitment content (see Section 3.4.2).

– **Process variables**

The process variables refer to the variables defined in the BPEL process and their values are not assigned from services messages. Such assignments are translated into transitions with true as their guards where the new values of the variables held in the target state (see **Figure 3.11.D**)

- **Invoke-Activities**

As explained before, the assignment-activity precedes the invoke-activity to assign the input message of the receiver to the output message of the sender. The actual transmission of the message is done by the invoke-activity. The assignment simulates the first step of the communication process, which is committing, while the invoke simulates the second step of the communication process, which is fulfilling the commitment (see Section 3.4.2). Thus the translation of the invoke-activity simulates what happens in the receiver side.

Let us consider the example of two services i (the sender) and j (the receiver). **Figure**

3.11.B illustrates the resulting translation when service j is invoked. First, the source state S_1 results from the translation of the BPEL activity preceding the invoke activity. It is not necessary for S_1 to be the same state resulting from translation of the assignment activity which assigns the output message of service i to the input message of service j because the invoke activity may not come directly after the assignment in the BPEL process. Second, the transition is labeled by an action performed by service i, **servicei.Action = transmissionvar2**. The resulting transition is labeled by an action performed by the sender in the explicit behaviors and by the environment in the implicit behaviors. In the target sate S_2, the shared buffer $i - j$ in service j is filled by the sent message, $var2$. **Figure 3.11.A** combined with **Figure 3.11.B** show the whole communication process established in the explicit behavior. By means of the invoke-activity translation, the shared buffers values in services i and j become equal. The target states resulting from the translation of the assignments and invoke activities compose the the accessibility relation.

- **If-Activities**

  The if-activity permits to define two different behaviors of the process depending on a Boolean condition. Then-branch is executed when the condition is true while the else-branch when it is false. Thus, to translate the if-activity into automata, each branch is separately translated. Assume that $S1$, $S'1$ are the beginning and ending states respectively for then-branch, and, similarly, $S2$, $S'2$ for else-branch. The activity is translated into two transitions: one ends at $S1$ with the Boolean condition as a guard, while the other ends at $S2$ with the negative Boolean condition as a guard. As before, the source state is the state resulting from the translation of the preceding activity of the if-activity (see **Figure 3.12.A**).

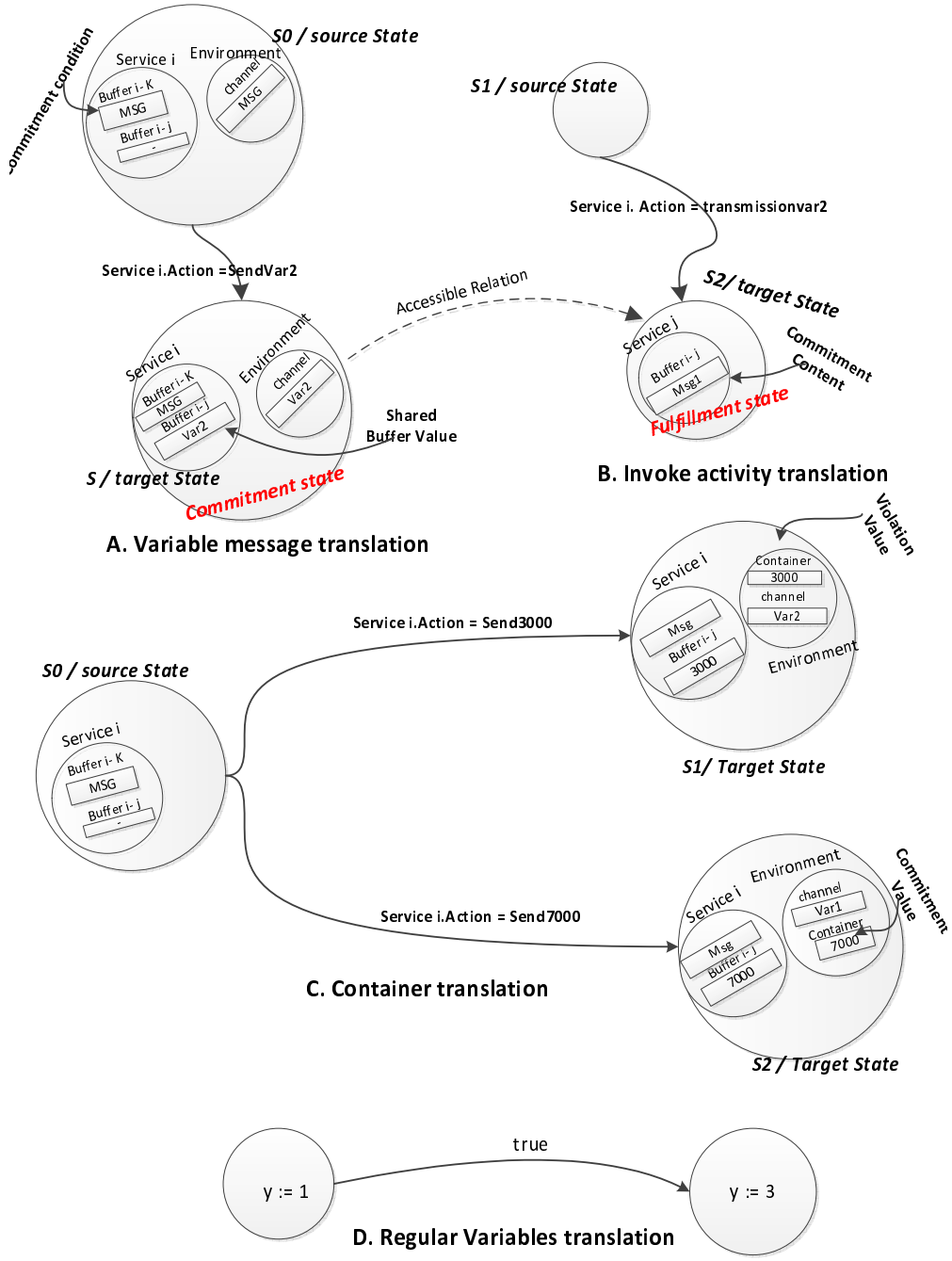


Figure 3.11: BPEL translating - Assign and Invoke Activities

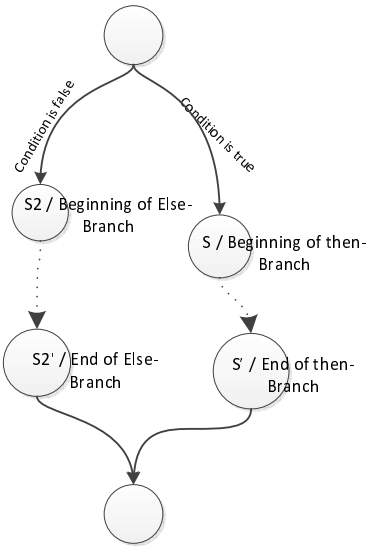
- **While-Activities**

The while-activity is composed of a Boolean condition and a loop body which is executed as long as the Boolean condition is true. To translate the while-activity into automata, the loop body is firstly translated. Assume that S, S' are beginning and ending

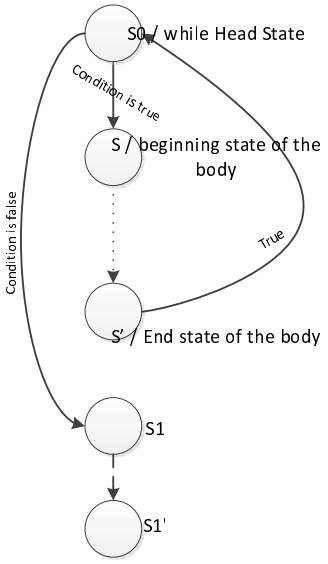
states respectively, assume also that $S1,\ S'1$ are beginning and ending states for a BPEL process coming after the while-activity block. The while-activity is translated into an empty state $S0$ preceding S and three transitions. **Figure 3.12.B** illustrates the guard for each transition.

- **Pick-Activities**

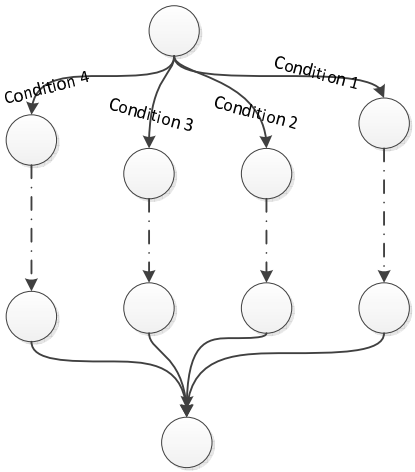
  The pick-activity is composed of multi branches, each branch is associated with an event triggering the branch execution. There are basically two types of events. The first type, onMessage, is the arrival of a new message. The second kind of event, onAletre, is alarm, which a condition for something to happen. These can be seen as multi-branches if-activity. Thus, the pick-activities are encoded by translating each branch as an if-activity (see **Figure 3.12.C**).



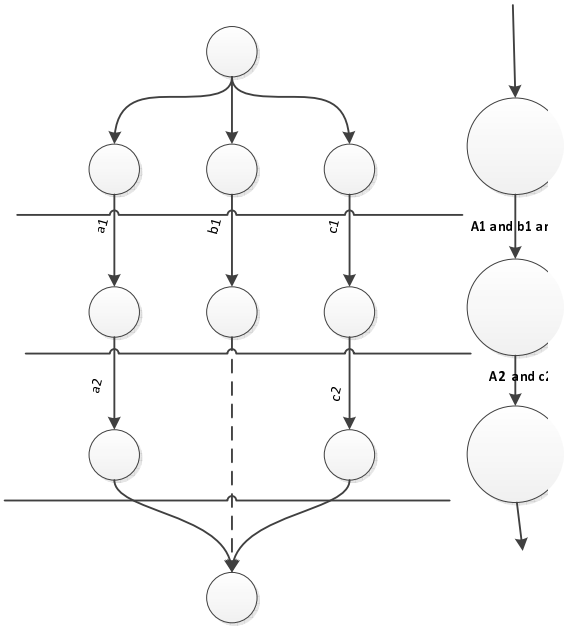
A. Translation of IF-activity



B. Translation of while-Activity



C. Translation of Pick-Activity



D.Translation of Flow-Activity

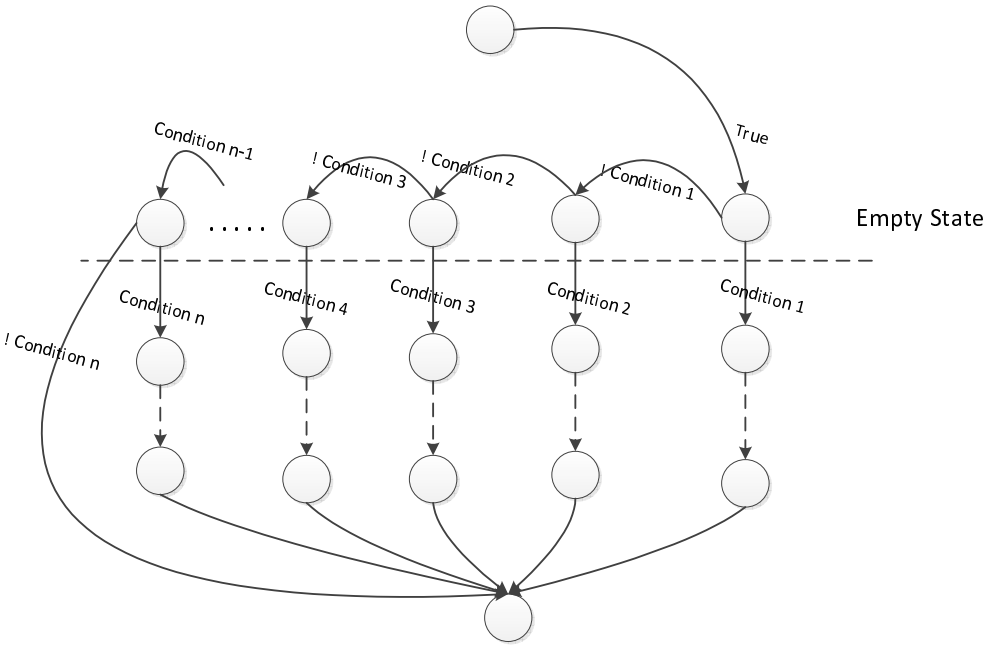


Figure 3.12: Translating BPEL - Control Activities

- **Flow-Activities**

  The flow-activity is composed of multi-branches which are executed concurrently. The links defined in the flow scope enable us to enforce precedence between these activities, i.e., they permit synchronization. To encode the Flow-activity, each branch is separately translated into automata, and then they are combined into a main automaton by synchronizing the corresponding transitions (see **Figure 3.12.D**).

- **Switch-Activities**

  The Switch-activity is composed of multi-branches, execution of each one is associated with a Boolean condition. The first branch whose condition is true is executed while the others are ignored. To translate the switch-activity, the branches are separately translated into automata. An empty state giving off two transitions is added at the beginning of each automaton. The first transition ends at the beginning state with the branch Boolean condition as a guard. The second transition ends at the empty state belonging to

the next branch with a negation for the branch Boolean condition as a guard (see **Figure 3.12.E**).

- **Custom-Activities**

  The custom-activities are provided to mark the generated automata and select some particular points to verify the system at. As explained earlier, the custom-activity has a set of assigned attributes with no internal operations. In fact, they are dealt with as a special type of assignment-activities where the attributes are considered as assigned variables. Thus, the compiler defines first the attributes as variables, then the custom-activities translation follows the process variables translation. Because of their non-impact on the BPEL-process, the translation results are merged. **Figure 3.13** shows an example where a custom-activity is translated into a transition $e0$ from S to $S0$. $e0$ is neglected and $S0$ is merged with the next state.

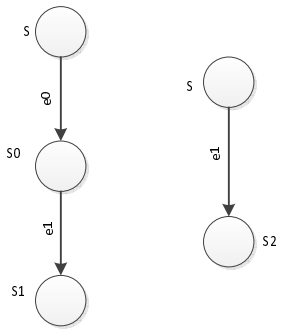


Figure 3.13: Translating BPEL - Custom Activities

This section provided a proof of soundness of the translation process. The completeness is straightforward as all the BPEL constructs are considered.

3.4.4 Encoding Automata into ISPL

The whole BPEL process representing the composition is translated into an automaton and then the services behaviors are extracted. In this section, we present the extraction and encoding processes. The compiler starts the encoding process at the point marked by the str-label and

ends at the point marked by the end-label. A database has been implemented in the compiler to store the resulting translation of the BPEL process. States_table, the key table inside the database, consists of the following tuple $<$ `StateID, StateName, SourceStateID, ActionName, EffectingService, EffectedService, VaraiableName, VaraiableVal ActivityType` $>$. EffectingService and EffectedService denote the sender and receiver respectively. Each row in the table represents a local state. The action name is the name of the action that causes the transition to this state, not the action allowed in this state. The EffectingService is the sender which performs this action while the EffectedService is the receiver. ActivityType denotes the BPEL-activity which is translated in this row. This tuple describes each transition in the resulting automaton. The extractor and encoder, the main parts in the compiler responsible to build the ISPL code, consist of SQL statements and stored procedures which are executed to retriever all the required information to compose the ISPL code. The encoding rules are described as follows:

1. **Local states generation**:

   As mentioned before, each service has two types of variables; the shared buffers and the Boolean variable to capture the trace-label values. The extractor initially defines the shared buffers for each service (each service has one communication buffer for each other service is interacting with). The service local states are simply generated by evaluating its shared buffers and its Boolean variable. Regarding to the environment, it has three types of variables: communication channels, containers and operational variables. The environment local states are generated by evaluating its communication channels as well as all the variables defined in the process.

   The following SQL-statement retrieves all possible values for each variable belonging to service i. It takes the service name as a parameter.

   ```
   Select distinct EffectedService, VariableName, VariableValue
   from States_table
   ```

```
where EffectedService = 'Servicei'
order by VaraiableName
```

Then, the encoder formats the SQL-statement output in ISPL as shown in the following example:

```
Vars:
Buffer_i_j:  { Msg1, Msg2, ...};
Buffer_i_K: { Msg11, Msg12, ...};
i_trace :  boolean;
end Vars
```

2. **Local actions generation**:

   As explained in the translation rules earlier, each transition is labeled by action performed by a particular service. The service actions are simply generated by retrieving all actions performed by this service. The following SQL-statement retrieves all possible actions belonging to service i. It takes the service name as a parameter.

```
Select distinct EffectingService, ActionName
from States_table
where EffectingService = 'Servicei'
```

   Then, the encoder formats the SQL-statement output in ISPL:

```
Actions = { SendMsg1, SendMsg2, ...  };
```

3. **Protocol generation**:

   The protocol defines at which state an action becomes allowed. Thus, for any service, its protocol is extracted by determining the source states for its actions. The service actions are fired based on the observed values of the communication channels defined in the environment and not only on the values of its internal buffers. This captures the fact that the services autonomy is affected by the environment, where they are temporally disabled until the environment reaches particular states.

The following SQL-statement retrieves the allowed actions for each state belonging to service i:

```
Select distinct SS.StateID RefStateID, SS.StateName RefStateName,
S.StateID State_ID, S.StateName State_Name,
S.EffectingService Effecting_Service, S.ActionName
from States_table S inner join States_table SS
on S.SourceStateID = SS.StateID
where S.EffectingService = servicei
order by SS.state_id
```

The extractor executes different stored procedures to get the values of the shared buffers and the communication channels at each state. Then, the encoder formats the SQL-statement output in ISPL, such as the following example:

```
Protocol:
Environment.Ch1 = Msg2 and Buffer_i_j = Msg2:  { SendMsg6,SendMsg8
};
Environment.Ch1 = Msg15 and Buffer_i_k = Msg3 :{ SendMsg30 };
Other :{ none };
end Protocol
```

4. **Evolution function generation**:

For a service, the evolution function describes the evolution of its local states in the system. Specifically, it returns the next local states given a source state and a transition. The local states can be affected by actions from other services. Consequently, it is possible that the transition is not a local transition.

The following ISPL code is an example on the evolution function for service i. As explained in lines 1-4, service i goes to new state, new value for `buffer_i_k` (line 1), if service k performs its action, `K_SendMsg_i` (line 4), and the values of the shared

`buffer_i_k` in service i and `Ch1` in the environment are `Msg1` (lines 3 & 4). In line 5, service i goes to another new state, new value for `buffer_i_j` , if it performs its action, `i_PrepareSendMsg_j` , and the values of the shared `buffer_i_k` in service i and `Ch1` in the environment are `Msg2` (lines 7 & 8).

```
line1:  Buffer_i_k = Msg2
line2:  if ( Environment.Ch1 = Msg1
line3:  and Buffer_i_k = Msg1
line4:  and ServiceK.Action = K_SendMsg_i) ;
line5:  Buffer_i_j = Msg3
line6:  if ( Environment.Ch1 = Msg2
line7:  and Buffer_i_k = Msg2
line8:  and Action = i_PrepareSendMsg_j);
```

To generate the evolution function for service i, We need to retrieve all attributes of the States_table tuple. In addition, we need to correlate each state in the table with its source state. The following SQL-statement retrieves all transitions for service i. It executes an inner join on the same table to retrieve the source states for each state. In fact, the extractor executes more complicated procedures to resolve hard issues such as a state has multiple source states, synchronized actions from multiple services (as in the translation of flow activity) and a transition with conditions.

```
Select distinct SS.StateID RefStateID, SS.StateName RefStateName,
S.*
from States_table S inner join States_table SS
on S.SourceStateID = SS.StateID
where S.EffectingService = servicei
order by SS.state_id
```

5. **Generating the initial state**:

   The system initial state consists of services initial states. It is expressed by assignments to services local variables. The system evolves from the initial state according to the protocols and evolution functions, and this process is used to compute the truth value of formulae specified by the user. The communication channels and shared buffers are assigned by empty values to denote that there is no communication that has been established yet among the services. The variables defined in the BPEL process are initialized using the same initialization values defined in the BPEL process.

6. **Generating evaluation and formula sections**:

   Evaluation section contains the propositions and basic formulae forming the verification properties. This section is built by formalizing the automata states marked by the translated labels. Specifically, the encoder uses the trace-labels to formalize the types of the services behaviors, desired or undesired behaviors, and it uses the commitment, fulfillment and recovery labels to formalize the messages received at the states marked by those labels.

   As mentioned earlier, the trace-label is a Boolean variable. The service behavior is undesired if the value of this variable is 0. The encoder retrieves the Boolean variable for each service and then specifies two atomic propositions, RedState and GreenState, for each variable. the following ISPL code is an example on these atomic propositions to represent the types of service i behaviors.

```
  iRedstate if (i_BehaviorType =0);
iGreenstate if (i_BehaviorType =1);
```

As explained in Section 3.4.3, the custom-activities are translated into automata by defining their attributes as variables, then the translation follows the translation rule of the assignment of the process variables. Thereafter, the resulting states are merged with the other states. In fact, the commitment-ID (defined in Section 3.3) is the key attribute in the commitment, fulfillment

and recovery labels. The compiler defines commitment-ID as an integer variable at the begging of the translation process. The commitment-ID value is 0 in all states, except the marked states where it has an integer number greater than 0. The encoder retrieves all marked states, and then generates for each state an atomic proposition formalizing the message it receives (as explained in Section 3.3, the commitments labels family (commitment, fulfillment and recovery labels) are inserted before the invoke activities).

The following SQL statement retrieves all marked states in the system. The compiler inserts global states into States_table, and each row in the table represents a local state inside a global state. Thus, there are many rows with the same StateID and StateName. Merging the commitment-labels state with the invoke activity state results to insert two rows in the table with the same StateID and StateName, one for assigning commitment-ID variable while the other for filling a buffer in the receiver. The SQL statement retrieves the message and the buffer on which the receiver gets the message, then the encoder formats the atomic proposition into ISPL-code as shown below.

SQL statement:

```
Select distinct EffectedService, VaraiableName, VaraiableValue
from States_table
where StateID in (Select StateID
from States_table where VaraiableName = 'commitment-ID'
and VaraiableValue <> 0 ) and VaraiableName <> 'commitment-ID'
```

ISPL:

```
Msg1 if (Buffer i-j = Msg1 and Environment.Ch1 = Msg1) ;
```

3.5 Generated Properties and their Expressiveness

A set of properties is automatically produced to verify the system at particular points. Specifically, the compiler formalizes the actions performed at the marked points and generates the consequent properties that should hold in the system. These properties reflect key issues that web developers generally aim to investigate in the composition, such as the commitments details and the parties' possible behaviors, implicit and explicit behaviors, evaluation of services behaviors in terms of fulfillment and violations, violations consequences and recoveries. The automatic properties can be used as a foundation to encode more complex properties. The compiler generates the properties automatically with static format, but the atomic propositions used in these properties are changed dynamically based on the marked points. This means that the compiler generates the same type of properties each time is executed, but the propositions are dynamic.

In a nutshell, generating the verification properties is implemented in the compiler as follows:

- **Generating the atomic propositions**:

  For each point marked by one of the commitment labels family (commitment, fulfillment and recovery labels), the encoder generates an atomic proposition formalizing the received message. This step is explained in details in Section 3.4.4.

- **Determining labels types and properties components**

  The encoder depends on the `ActivityName` attribute in the States_table to extract the label types. Based on the label type and its attributes, the encoder builds the properties. The points marked by the commitment-label capture the beginnings of the contract clauses. Mainly, the encoder investigates three key items for any point marked by the commitment-label: commitment condition, commitment content, and what behaviors respect to this commitment the committed service can engage in. Property (1) is generated automatically as a result of compiling the commitment-label as follows.

$$A(\neg\varphi U(\varphi \wedge EX(C_{i\rightarrow j}\psi) \wedge i_{Greenstate})) \quad (1)$$

- The encoder considers the atomic proposition, φ, generated from the point marked by the commitment-label as the commitment condition.
- Based on the `debtor` and `creditor` attributes in the commitment-label, the encoder determines the contracting parties in this clause, and then encodes the commitment operator, $C_{debtor\rightarrow creditor}$. As expressed in Property (1), service i is the debtor while service j is the creditor.
- For each path emerging from the state marked by the commitment-label, the encoder explores it and fetches the first message sent by the debtor toward the creditor. The encoder formalizes the fetched message as an atomic proposition as explained in Section 3.4.4. This message is considered as the commitment content. As shown in Property (1), ψ is the sent message, i.e., commitment content, from service i to service j.
- Based on the value of the debtor trace-label in the explored path, the encoder determines whether the fetched message is an desired or undesired behavior.
- The encoder generates a copy from Property (1) for each explored path to show what behaviors, with respect to this commitment, the debtor can engage in. The copies have different atomic propositions, ψ, representing the sent messages with different atomic proposition ($i_{Greenstate}$, $i_{Redstate}$) representing the behavior types (desired and undesired).

The encoder generates Properties (6-14) by defining a conjunction between the atomic proposition resulting of compiling the commitment-label and the atomic propositions resulting of compiling the fulfillment and recovery labels which have the same value of the commitment-ID in the commitment-label.

Generally, the point marked by the commitment-label catches the firing moment at which the commitment condition holds. After this moment, service i behaves either toward fulfilling

or violating its commitment. Property (1) means that there exists a path where service i will not commit to service j about ψ, commitment content, until it receives message φ, commitment condition. This property expresses reachability, which is a particular situation that can be reached from the initial state via some computation sequences.

The proposed approach uses the CTLC ability in describing the exchanging message to express the violations.

The commitment operator abstracts the operation of conveying messages, while the atomic proposition $i_{Redstate}$ is used to indicate violation. Suppose that Properties (2-5) are generated with respect to a particular commitment between service i (the debtor) and service j (the creditor). Property (3) means that service i will convey Msg_2 to service j and this is considered as a violation, $i_{Redstate}$. In the case of explicit behaviors, Msg_1, Msg_2, Msg_3 and $NullMsg$ represent the service i's behaviors toward its commitment. In the case of implicit behaviors, Msg_1, Msg_2, Msg_3 and $NullMsg$ represent what the environment expects from service i toward its commitment.

$$A(\neg Msg_0\ U(\varphi \wedge EX(C_{i\rightarrow j}Msg_1) \wedge i_{Greenstate})) \quad (2)$$

$$A(\neg Msg_0\ U(\varphi \wedge EX(C_{i\rightarrow j}Msg_2) \wedge i_{Redstate})) \quad (3)$$

$$A(\neg Msg_0\ U(\varphi \wedge EX(C_{i\rightarrow j}Msg_3) \wedge i_{Redstate})) \quad (4)$$

$$A(\neg Msg_0\ U(\varphi \wedge EX(C_{i\rightarrow j}NullMsg) \wedge i_{Redstate})) \quad (5)$$

Web developers can catch any missing behaviors in the design with respect to a particular contract clause just by reviewing these copies independently of their satisfiability in the system. Property (5) is generated when one of the explored paths does not observe any behavior from service i toward its commitment where $NullMsg$ is null. Property (5) indicates that some clauses are not implemented in the system. This property will be engendered, for instance, if a service assigns a container and then this container is unchecked by any of the control operations in the environment. Entire absence of Properties (3),(4) and (5) refers to an absolute compliance of service i toward its commitment.

More faults in the system can be discovered by scanning the satisfiability of the generated copies. For instance, an incorrect or incomplete commitment implementation is captured when Property (2) is false. This refers to the absence of accessibility relation, which means that service j does not receive Msg_1. However, unsatisfiability of Property (3) does not necessarily mean an existence of faults; because in the violation cases, most of the compositions stop the communication process.

In fact, the previous properties reflect immediate reactions of service i toward its commitment. However, services do not always show immediate reactions, so the operator X is replaced by the operator F in cases of the non-immediate reactions. The system engineers may still wish to guarantee that the commitment held at the marked point will eventually happen in their design as well as it will not happen without its condition. This is classified under the notions of liveness and safety properties which state that a particular thing will eventually happen and something bad never happens respectively. Property (6) is generated to achieve this goal. It means that in all paths eventually the condition will happen and service i will commit to service j. The importance of this property is checking whether the system recovers from service i's violations by forcing it to commit again to service j or not.

$$AF(\varphi \wedge (C_{i \to j}\psi)) \quad (6)$$

Still, these properties do not verify the commitments completely; precisely, commitments fulfillment is missing. The key issue related to the fulfillment is checking whether the fulfillment takes place in all commitment paths or not. In the following, two properties are automatically proposed to serve this goal (Properties (7) and (8)). Property (7) means in all paths globally if service i commits to service j, then in all future computations, the fulfillment will happen. If Property (7) is false, it can be stated by Property (8) whether the fulfillment may not take place in any path or may take place in some paths and not in the remains. Property (8) means in all paths globally, if service i commits to service j, then in some future computations, the fulfillment will happen. The two properties together determine exactly the case.

When they are true, it is inferred that the commitment will be correctly realized and delivered to the creditor.

$$AG((C_{i\to j}\psi) \to AF(Fu(C_{i\to j}\psi))) \quad (7)$$

$$AG((C_{i\to j}\psi) \to EF(Fu(C_{i\to j}\psi))) \quad (8)$$

One may want to check the systems resilience in dealing with the violation cases. Particularly, whether the system can recover or not and what reactions the system provides. As mentioned earlier, red states denote the violations. So, the recoveries are observed when the system switches from the red states to the green states. Initially, the compiler generates Properties (9) and (10) which examine generally the system paths in terms of totally compliant and unrecovered violations respectively.

$$EG(i_{Greenstate}) \quad (9)$$

$$EF(G(i_{Redstate})) \quad (10)$$

Property (11) is automatically generated to know whether the system can recover from a particular violation or not. It means that in some paths eventually the commitment will not take place and there is no way for service i to recover from its situation. If Property (11) is true, this reports unrecovered violations associated with this commitment. Property (12) is generated to determine unrecovered violations.

$$EF((\neg C_{i\to j}\psi) \wedge EG(i_{Redstate})) \quad (11)$$

$$AG((C_{i\to j}\psi_1) \to AF((i_{Redstate}) \wedge (i_{Redstate}\, U\, i_{Greenstate}))) \quad (12)$$

Indeed, these properties are not able to show recoveries details. The recoveries may be applicable in the form of penalties, additional rights to some party, and, possibly, compensations paid to the affected services. Properties (13) and (14) formalize for each violation its recovery via the recovery-labels. Precisely, the compiler formalizes the action ω captured at the point

where the recovery-label is inserted. The captured events may be applicable in form of commitments or fulfillments. Property (13) means that in all paths globally if service i violates its commit toward service j, then the recovery represented by ω will be executed in all future computations. One may want to check if a recovery may result to force service i to recommit to service j about the same violated commitment. Property (14) is generated to do that.

$$AG(C_{i\to j}\psi_1 \wedge i_{Redstate} \rightarrow AF(i_{Redstate} \wedge (i_{Redstate}\, U\, (i_{Greenstate} \wedge \omega)))) \quad (13)$$

$$AG(\omega \rightarrow AF(C_{i\to j}\psi_1) \wedge AF(Fu(C_{i\to j}\psi_1))) \quad (14)$$

CHAPTER 4. DETAILED CASE STYDY AND EXPERIMENTAL ANALYSIS

In this section, we apply our approach on a case study, which was presented in (56) to verify service composition with MCMAS. We selected this case study, so that we can compare our results with those obtained from (56). The case study is summarized into subcontracts where each one is represented here as a commitment from one of the participating services toward another one. In the scenario, the participating contract parties include a principal software provider (PSP), a software provider (SP), a software client (C), a testing agency (T), a hardware supplier (H), and a technical expert (E). Table 1 illustrates these commitments, the conditions under which some local violations may occur and the recoveries if any.

Table 4.1: Case study: contract clauses

Clause number	Contract's content	Creditor Service	Message Name	Debtor service	Violation condition	Recovery
1	Client C would like to have a piece of software developed and deployed on hardware. To do that, the client C initiates this scenario by sending a request to the PSP asking for the needed software.	C	Software_request	PSP		
2	PSP sends this request to SP	PSP	component_request	SP		

Continued on next page

Table 4.1 – *Continued from previous page*

Clause number	Contract's content	Creditor Service	Message Name	Debtor service	Violation condition	Recovery
3	SP replies to the PSP's request by sending the needed components. And then the PSP integrates these components.	SP	components	PSP		
4	Through the integration process, the PSP has to update the client C twice about the progress of the software development.	PSP	Update_process	C	If PSP does not send his updates as per schedule	charging a penalty
5	After the first update from the PSP, C can update the software specification by sending a change request. This at no cost before the second round of updates.	C	Change_request	PSP	Any change required by the client after the second update	The client may be charged a penalty at PSP's discretion. If the penalty is levied, the client can recover from this violation by paying the penalty or by withdrawing the request for changes.
6	After the PSP has integrated the components, he sends the integrated software to T for testing.	PSP	Integrated_Components	T		
7	If the integration test fails, the components are revised and tested again. Components can be revised twice.	T	Revised_component_req	PSP	The third test fails.	C may cancel the contract with PSP
8	Through the testing process, PSP has to update SP and C concurrently after each time he gets the result from T.	PSP	Testing_result	SP	PSP does not send this update as per schedule	charged a penalty
9	If the testing succeeds, PSP delivers the integrated components to C.	PSP	Software	C		

Continued on next page

Table 4.1 – *Continued from previous page*

Clause number	Contract's content	Creditor Service	Message Name	Debtor service	Violation condition	Recovery
10	After getting the integrated components, C then has to pay PSP as a first payment.	C	Payment	PSP	C does not pay to PSP	PSP does not send the code and then the process is stopped
11	After getting the integrated components, C then invokes H to order the hardware.	C	Hardware_Req	H		
12	H then replays by sending the needed hardware	H	Hardware	C	H does not deliver the Hardware	The process is stopped
13	Finally C invokes E to get the software deployed.	C	Deployment_Req	E		
14	If the software cannot be deployed, then the hardware and the components have to be re-evaluated. Software components can be revised twice at no penalty.	E	Revised_Request	H	third test fails	C cancels the contract with PSP and H.
15	If the software and hardware are deployed correctly, then C has to pay the second payment to PSP.	C	Second_Payment	PSP	C does not pay for PSP	process is stopped
16	After the payment, E has to send the deployed component to the client	E	Deployed_Software	C		
17	PSP has to pay SP	PSP	payment_portion	SP	PSP does not pay SP	PSP has to be charged directly.
18	After the payment, the process will be successfully finished					

Various properties of contract details for the motivating case study are formalized. Some of these properties, as explained earlier, are generated automatically based on the custom ac-

tivities, while the others are generated manually. An illustrative example, suppose that the BPEL-process representing the case study is marked to cover the contract clauses 4 and 8 described in Table 4.1. We only explain these two clauses as they cover the key cases addressed in the proposed approach, specifically, the recovery process.

As mentioned in clause 4 (Table 4.1), PSP may show two behaviors. First, he commits to the client to send an updating message when he receives the components. Second, he does not send the updating message. Formulas 16 and 17 specify these behaviors respectively.

$$A(\neg components\, U\, (components \wedge EX(C_{PSP\rightarrow C} update\_process))) \quad (16)$$

$$A(\neg components\, U\, (components \wedge EX(C_{PSP\rightarrow C} null \wedge PSP_{Redstate}))) \quad (17)$$

Unlike Formula 17 that returns false, Formula 16 holds in the system. In fact, Formula 17 represents a violation in which the debtor-service does not send any message to the creditor service, which means unsatisfiability of the social accessibility relation $\sim_{PSP\rightarrow C}$. As expressed in Formula 18, if PSP commits to send the update message, then in all paths eventually C will receive it.

$$AG(C_{PSP\rightarrow C} update\_process \rightarrow AF(Fu(C_{PSP\rightarrow C} update\_process))) \quad (18)$$

In case PSP does not send the update message, it is considered as a violation, which can be recovered from by charging PSP a penalty. As expressed in Formula 19, charging the PSP brings him back to the green state.

$$AG(C_{PSP\rightarrow C} null \rightarrow$$
$$AF(PSP_{Redstate} \wedge (PSP_{Redstate}\, U\, (PSP_{GreenState} \wedge Fu(C_{PSP\rightarrow C} Charging))))) \quad (19)$$

Clause (8) (Table 4.1) is similar to Clause (4); PSP has to update C about the testing-results after each time he gets the result from T. Thus, the following properties, similar to the above, are generated.

$$A(\neg Result from T\, U\, (Result from T \wedge EX(C_{PSP \to C} testing\_result))) \quad (20)$$

$$A(\neg Result from T\, U\, (Result from T \wedge EX((C_{PSP \to C} null) \wedge PSP_{Redstate}))) \quad (21)$$

$$AG(C_{PSP \to C} testing\_result \rightarrow AF(Fu(C_{PSP \to C} testing\_result))) \quad (22)$$

$$AG(C_{PSP \to C} null \rightarrow$$
$$AF(PSP_{Redstate} \wedge (PSP_{Redstate}\, U\, (PSP_{GreenState} \wedge Fu(C_{PSP \to C} Charging))))) \quad (23)$$

The first violation for C is spotted when he gets the software, which means that C is always in compliance until getting the software where he becomes committed to PSP about the first payment. Properties 24 and 25 describe the behaviors of C towards this commitment. Property 25 encodes a possibility for C to not fulfill his commitment by sending a cancel message. It is desired to make sure that the payment is successfully delivered to PSP in case C has paid. Property 26 states that if C commits to PSP about the first payment, then in all paths eventually PSP will get the amount; the fulfillment will be held in all paths. As expressed by properties 27 and 28, the violation case is handled in the system by stopping the process where C is blocked. The system cannot recover from this situation so the client becomes always in red state. Properties 24, 25, 26 hold while 27 and 28 are false.

$$A(\neg software\, U\, (software \wedge EX(C_{C \to PSP} first\_payment))) \quad (24)$$

$$A(\neg software\, U\, (software \wedge EX((C_{C \to PSP} Cancel) \wedge C_{Redstate}))) \quad (25)$$

$$AG(C_{C \to PSP} first\_payment \rightarrow AF(Fu(C_{C \to PSP} first\_payment))) \quad (26)$$

$$AG(C_{C \to PSP} Cancel \rightarrow$$
$$AF(C_{Redstate} \wedge (C_{Redstate}\, U\, (C_{GreenState} \wedge BlockingProcess)))) \quad (27)$$

$$AG(BlockingProcess \rightarrow (AF(C_{C \to PSP} null) \wedge AF(Fu(C_{C \to PSP} null)))) \quad (28)$$

The second violation case for C takes place when he would not pay the final payment for PSP after getting the deployed software. This violation is handled in the same manner in which the first payment is handled. So, the following properties are similar to those through which the

C's second commitment is verified. The only difference is the commitment condition which, in our case, is getting the deployed software.

$$A(\neg Deplsoftware\, U\, (Deplsoftware \wedge EX(C_{C\rightarrow PSP} second\_payment))) \quad (29)$$

$$A(\neg Deplsoftware\, U\, (Deplsoftware \wedge EX((C_{C\rightarrow PSP} Cancel) \wedge C_{Redstate}))) \quad (30)$$

$$AG(C_{C\rightarrow PSP} second\_payment \rightarrow AF(Fu(C_{C\rightarrow PSP} second\_payment))) \quad (31)$$

$$AG(C_{C\rightarrow PSP} Cancel \rightarrow AF(C_{Redstate} \wedge (C_{Redstate}\, U\, (C_{GreenState} \wedge BlockingProcess)))) \quad (32)$$

$$AG(BlockingProcess \rightarrow (AF(C_{C\rightarrow PSP} null) \wedge AF(Fu(C_{C\rightarrow PSP} null)))) \quad (33)$$

In all paths, PSP must pay to SP his portion once C has paid. The system will not allow PSP to escape from his commitment, so as recovery in the violation case, PSP pays SP inevitably. As expressed by the following property, the recovery forces eventually PSP to commit towards SP to send him his portion and satisfy his commitment.

$$AG(Fu(C_{C\rightarrow PSP}\, second\_payment) \rightarrow AF((C_{PSP\rightarrow SP}\, portion\_payment) \wedge AF(Fu(C_{PSP\rightarrow SP}\, portion\_payment)))) \quad (34)$$

There is a trace in which the software will be eventually delivered once C has requested it from PSP. Property 35 describes this particular situation.

$$AG(\neg software\_request\, U\, (software\_request \wedge EF(Fu(C_{PSP\rightarrow C} software)))) \quad (35)$$

Always in all paths, if PSP commits to C about the software, then there is a path in which the software will be eventually delivered to the client provided that the software has not undergone more than two times the testing process. This is expressed in property 36 which is satisfied in the system.

$$AG(C_{PSP\rightarrow C} software \rightarrow EF(Fu(C_{PSP\rightarrow C} software) \wedge TestingLess\_3)) \quad (36)$$

To make sure that the software will be never delivered to C if it does not pass the testing process from the first two times, the property 37 is used. This property verifies if there is a possible computation in which eventually the software undergoes the testing process more than three times and then it is delivered for the client. This property is false in our case study.

$$EF(TestingLess\_3 \wedge Fu(C_{PSP \rightarrow C} software)) \quad (37)$$

Failing to pass the testing process in the first two times is considered as a violation and the system must handle the case. As mentioned in the example, the system cancels the contract with PSP if the integrated software does not pass the testing process into two rounds as a maximum. The cancelation cannot recover the system, so PSP becomes always in red state. Property 38, which holds in the example, formalizes this issue.

$$AG((EF(Testingmore\_3) \wedge C_{C \rightarrow PSP} software) \rightarrow A(Cancel \wedge AG(PSP_{Redstate}))) \quad (38)$$

The system allows C to update the software specifications with no cost after the first update-round. Property 39 checks the existence of a path in which C can eventually request a change in the software specifications. As a result of this request, PSP must respond by revising the components of the software. Property 40 is provided to make sure that in all paths PSP eventually revises the software's components once the changing request is sent by C. these properties are both holding in the case study.

$$AG(FirstRoundUpdate \rightarrow EF(Fu(C_{C \rightarrow PSP} ChangeRequest1))) \quad (39)$$

$$AG(Fu(C_{C \rightarrow PSP} ChangeRequest1) \rightarrow AF(Fu(C_{PSP \rightarrow C} revisedcomponent))) \quad (40)$$

It is possible for C to make update request after the second round, but it is considered as a type of violation. However, the scenario here is similar to the previous one except that C, in addition, must be charged penalty. Property 41 states that always in all computations if the second update round comes, then there is a trace in which C can request an update for the software specifications. This property is true. As expressed in property 42, once the update

request after the second round is sent, in all paths PSP must revise the software components as C desires. The system gives C the chance to withdraw his request but if he insists to update the specifications, then the system recovers by charging him. Property 43 is provided to serve this goal. It verifies if in all paths C is charged, PSP revises the software's components and C's reaches a green state again after his request for an update in the specifications. Because of the withdraw option, this property is false. However, property 44, which instead of considering all the paths as in property 43, it only considers the possibility of finding a path, holds in the system.

$$AG(SecondRoundUpdate \rightarrow EF(Fu(C_{C\rightarrow PSP}changeRequest2))) \quad (41)$$

$$AG(Fu(C_{C\rightarrow PSP}changeRequest2) \rightarrow AF(Fu(C_{PSP\rightarrow C}revisedcomponent))) \quad (42)$$

$$AG(C_{C\rightarrow PSP}changeRequest2 \rightarrow AF(Fu(C_{C\rightarrow PSP}Charging) \wedge AF(Fu(C_{PSP\rightarrow C}revisedcomponent) \wedge EG(C_{Greenstate})))) \quad (43)$$

$$AG(C_{C\rightarrow PSP}changeRequest2 \rightarrow EF(Fu(C_{C\rightarrow PSP}Charging) \wedge AF(Fu(C_{PSP\rightarrow C}revisedcomponent) \wedge EG(C_{Greenstate})))) \quad (44)$$

To guarantee that C does not completely get the software until he pays the first payment to PSP, property 45 is satisfiable in the example.

$$A(\neg GettingSoftwareCode\, U\, Fu(C_{PSP\rightarrow C}FirstPayment)) \quad (45)$$

In the same way, property 46 is provided to guarantee that C does not get the hardware until he pays the second payment to PSP. This property also holds in the example.

$$A(\neg DeployedSoftware\, U\, Fu(C_{PSP\rightarrow C}Secondpayment)) \quad (46)$$

In some paths, the software can be eventually integrated, tested and deployed if PSP and H fulfill their commitments about software and hardware respectively. This property (47) is true in the example.

$$AG(Fu(C_{PSP\rightarrow C}software) \wedge EF(Fu(C_{H\rightarrow C}Hardware)) \rightarrow$$
$$EF(Fu(C_{E\rightarrow C}Deployedsoftware))) \quad (47)$$

For more clarification and better understanding of the proposed approach, we provide details about clause 10 (see Table 4.1) in Appendices A and B. **Figure 1** shows section of the BPEL process file which implements clause 10. **Figures 2**, **3**, and **4** show section of the generated ISPL code for the Environment, C, and PSP respectively. Finally, **Figure 5** presents the generated properties.

We are not aware of any work supporting commitment properties in contract-oriented settings to compare with in terms of execution statistics of the verification process (e.g., execution time, memory, number of states). However, the closest work to ours is the framework proposed in (57), which is based on epistemic logic. Table 4.2 compares those execution statistics in the two proposals using the same case study On a machine running Windows 7 Enterprise on an Intel(R) Core (TM) i7 2.6GHz with 9GB memory. It is worth mentioning that the techniques used in the two approaches are different and also the models generated are different (see Section 1.4 for the technical comparison). The main point in this empirical comparison is that the size of the model in our approach is smaller than the one from (57). This is mainly because in our approach the service composition is explicitly modeled as the environment agent, which allows us to better control the possible actions that services can perform. Consequently, the number of reachable states in our approach is smaller. This justifies why our approach shows better performance in terms of execution time and memory usage.

Table 4.2: Comparison with Lomuscio's Approach (57)

	Proposed Approach	Lomuscio's Approach (57)
Number of BDD variables	150	134
Number of properties	28	22
Execution time	1.132 sec	9 sec
Memory consumption	8 MB	16 MB
Number of Global States	642	13799

CHAPTER 5. CONCLUSION AND FUTURE WORK

5.1 Summary of Contributions

In this thesis, we proposed an automatic novel approach to verify the correctness and robustness of web services compositions designed by BPEL. The approach starts by marking the BPEL process to represent the specifications. Custom activities, called labels, are created to mark the BPEL process. In addition to represent the specifications, the labels are used to determine the points at which the system engineers desire to verify the composition. We implemented a tool translating automatically the marked BPEL process into ISPL code and generating the verification properties. The CTLC logic, expressive enough to represent details of the contracts regulating MASs, is used to verify the web service composition. Translation rules are provided for transforming BPEL process into ISPL implementing a communication infrastructure to apply the CTLC logic. The tool has been implemented to provide a fully automatic verification process. Thereafter, the resulting code together with the required properties against which the system will be checked are used as inputs to the MCMAS model checker that supports CTLC. MCMAS shows then whether the properties are satisfied or not in the system. We have discussed our approach using a case study and the simulation results shows that the whole approach is promising in terms of performance.

The approach has many salient features distinguishing it from the literature. First, we optimize the verification process by marking the specifications instead of fully modeling the whole specification set separately; which also decreases the change of having errors in representing the specifications and time consumption. Second, the verification properties are generated au-

tomatically; they are extracted from the composition implementation. The approach extracts what properties the system exhibits rather than checking some properties randomly on the system. Some faults are discovered just by observing the existence and absence of some types of generated properties without verifying them in the system. Third, our approach allows verifying the composition partially and at particular points. Fourth, we brought a rich and expressive range of specifications coming directly from multi-agent systems theories and applied them into web services environments. Fifth, a powerful representation of the contracts regulating web services is provided. Services' behaviors are described in terms of contract-compliant, violations, and recoveries. We can reason about multiple violations as required, respect to a particular contract clause. Sixth, our approach is realistic and rational. It 1) considers web services as black boxes, which hide their internal implementations; and 2) realizes the challenges, difficulties and sometimes the inability to model the specifications, i.e. commitments mentioned in the contracts, separately or to compile them from their informal representations. Seventh, the approach is automatic and paired with a state-of-the-art model checker for multi-agent systems, thereby enabling the possibility of verifying very large state spaces such as those arising from real scenarios.

5.2 Future Work

One of the limitations of this work is the common problem of model checking, which is state explosion problem. The other limitation is related to the MCMAS model checker itself, which cannot define concurrent behaviors of the same service. On the other hand, we are investigating the extension of the present framework to generate more complicated properties formalizing different types of actions and commitments, such as, the actions that go beyond the exchanging messages, or the commitments which have interventions from the environment or other services. This approach limited the services' roles into exchanging messages because of unawareness of their internals. As future work we aim to extend the framework to verify

the composition if there is possibility to get the actual code of each participating service.

BIBLIOGRAPHY

[1] ALONSO, G., CASATI, F., KUNO, H., AND MACHIRAJU, V. *Web Services: Concepts, Architectures and Applications*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[2] ALVES, A., ARKIN, A., ASKARY, S., AND BARRETO, C. Web Services Business Process Execution Language Version 2.0 (OASIS Standard). WS-BPEL TC OASIS, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, 2007.

[3] ARDAGNA, D., COMUZZI, M., MUSSI, E., PERNICI, B., AND PLEBANI, P. Paws: A framework for executing adaptive web-service processes. *Software, IEEE 24*, 6 (2007), 39–46.

[4] ARKIN, A. Business Process Modeling Language (BPML). Tech. rep., Business Process Management Initiative (BPMI), 2002.

[5] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. A view of cloud computing. *Communications of the ACM 53*, 4 (2010), 50–58.

[6] BAIER, C., AND KATOEN, J. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[7] BALDONI, M., BAROGLIO, C., AND MARENGO, E. Behavior-oriented commitment-based protocols. In *Proceedings of the 2010 Conference on ECAI 2010: 19th European*

Conference on Artificial Intelligence (Amsterdam, The Netherlands, The Netherlands, 2010), IOS Press, pp. 137–142.

[8] BALDONI, M., BAROGLIO, C., MARTELLI, A., AND PATTI, V. Verification of protocol conformance and agent interoperability. In *Computational Logic in Multi-Agent Systems* (London, UK, 2005), Springer, pp. 265–283.

[9] BENTAHAR, J., EL-MENSHAWY, M., QU, H., AND DSSOULI, R. Communicative commitments: Model checking and complexity analysis. *Knowledge-Based Systems 35* (2012), 21–34.

[10] BENTAHAR, J., MOULIN, B., MEYER, J., AND CHAIB-DRAA, B. A logical model for commitment and argument network for agent communication. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 792–799.

[11] BENTAHAR, J., MOULIN, B., MEYER, J., AND LESPÉRANCE, Y. A new logical semantics for agent communication. In *Proceedings of the 7th International Conference on Computational Logic in Multi-agent Systems* (Berlin, Heidelberg, 2007), Springer-Verlag, pp. 151–170.

[12] BENTAHAR, J., YAHYAOUI, H., KOVA, M., AND MAAMAR, Z. Symbolic model checking composite web services using operational and control behaviors. *Expert Systems with Applications 40*, 2 (2013), 508–522.

[13] BERARDI, D., CALVANESE, D., DE GIACOMO, G., LENZERINI, M., AND MECELLA, M. Automatic service composition based on behavioral descriptions. *International Journal of Cooperative Information Systems 14*, 4 (2005), 333–376.

[14] BLAKE, M. B., CUMMINGS, D. J., BANSAL, A., AND BANSAL, S. K. Workflow composition of service level agreements for web services. *Decision Support Systems 53*, 1 (2012), 234–244.

[15] BOUTROUS-SAAB, C., COULIBALY, D., HADDAD, S., MELLITI, T., MOREAUX, P., AND RAMPACEK, S. An integrated framework for web services orchestration. *International Journal of Web Services Research 6*, 4 (2009), 1–29.

[16] BOZKURT, M., HARMAN, M., AND HASSOUN, Y. Testing verification in service-oriented architecture: A survey. *Software Testing, Verificaton and Reliability 23*, 4 (2012), 261 – 313.

[17] CASATI, F., AND SHAN, M. Dynamic and adaptive composition of e-services. *Information Systems 26*, 3 (2001), 143–163.

[18] CIMATTI, A., CLARKE, E. M., GIUNCHIGLIA, E., GIUNCHIGLIA, F., PISTORE, M., ROVERI, M., SEBASTIANI, R., AND TACCHELLA, A. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification* (London, UK, UK, 2002), Springer-Verlag, pp. 359–364.

[19] CLARKE, E., EMERSON, E. A., AND SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8*, 2 (1986), 244–263.

[20] CLARKE, E., GRUMBERG, O., AND PELED, D. *Model Checking*. The MIT Press, Cambridge, 1999.

[21] CLARKE, E., GRUMBERG, O., AND PELED, D. *Model Checking*. MIT Press, 2000.

[22] CLARKE, E. M., AND EMERSON, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop* (London, UK, UK, 1982), Springer-Verlag, pp. 52–71.

[23] CRAVEN, R., AND SERGOT, M. Agent strands in the action language. *Journal of Applied Logic 6*, 2 (2008), 172–191.

[24] DASTANI, M., HINDRIKS, K. V., AND MEYER, J. *Specification and Verification of Multi-agent Systems*. Springer Publishing Company, Incorporated, 2010.

[25] DESAI, N., CHENG, Z., CHOPRA, A. K., AND SINGH, M. P. Toward verification of commitment protocols and their compositions. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems* (New York, NY, USA, 2007), ACM, pp. 144–146.

[26] DESAI, N., CHOPRA, A. K., AND SINGH, M. P. Amoeba: A methodology for modeling and evolving cross-organizational business processes. *ACM Transactions on Software Engineering and Methodology 19*, 2 (2009), 1–45.

[27] DESAI, N., NARENDRA, N. C., AND SINGH, M. P. Checking correctness of business contracts via commitments. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems* (Richland, SC, 2008), International Foundation for Autonomous Agents and Multiagent Systems, pp. 787–794.

[28] DUMEZ, C., SIDI MOH, A. N., GABER, J., AND WACK, M. Modeling and specification of web services composition using UML-S. *Next Generation Web Services Practices, International Conference on* (2008), 15–20.

[29] DUSTDAR, S., AND SCHREINER, W. A survey on web services composition. *International Journal of Web and Grid Services 1*, 1 (2005), 1–30.

[30] EL-MENSHAWY, M., BENTAHAR, J., AND DSSOULI, R. *Verifiable Semantic Model for Agent Interactions Using Social Commitments*, vol. 6039 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, ch. 8, pp. 128–152.

[31] EL-MENSHAWY, M., BENTAHAR, J., EL KHOLY, W., AND DSSOULI, R. Verifying conformance of multi-agent commitment-based protocols. *Expert Systems with Applications 40*, 1 (2013), 122–138.

[32] EL-MENSHAWY, M., BENTAHAR, J., KHOLY, W. E., AND DSSOULI, R. Reducing model checking commitments for agent communication to model checking ARCTL and GCTL*. *Autonomous Agents and Multi-Agent Systems 27*, 3 (2013), 375–418.

[33] EMERSON, E. A., AND HALPERN, J. Y. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1982), ACM, pp. 169–180.

[34] ENDREI, M., ANG, J., ARSANJANI, A., CHUA, S., COMTE, P., KROGDAHL, P., LUO, M., AND NEWLING, T. *Patterns: Service-Oriented Architecture and Web Services*. IBM Redbooks, 2004.

[35] FAGIN, R., AND HALPERN, J. Y. Reasoning about knowledge and probability. *Journal of the ACM 41*, 2 (1994), 340–367.

[36] FAGIN, R., HALPERN, J. Y., MOSES, Y., AND VARDI, M. Y. *Reasoning about Knowledge*. The MIT Press, Cambridge, 1995.

[37] FAN, S., ZHAO, J. L., DOU, W., AND LIU, M. A framework for transformation from conceptual to logical workflow models. *Decision Support Systems 54*, 1 (2012), 781–794.

[38] FOSTER, H., UCHITEL, S., MAGEE, J., AND KRAMER, J. Ltsa-ws: A tool for model-based verification of web service compositions and choreography. In *Proceedings of the 28th International Conference on Software Engineering* (New York, NY, USA, 2006), ACM, pp. 771–774.

[39] FU, X., BULTAN, T., AND SU, J. Analysis of interacting bpel web services. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA, 2004), ACM, pp. 621–630.

[40] GIUNCHIGLIA, E., LEE, J., LIFSCHITZ, V., MCCAIN, N., AND TURNER, H. Nonmonotonic causal theories. *Artificial Intelligence 153*, 1-2 (2004), 49–104.

[41] GOTNES, T., VAN DER HOEK, W., RODRGUEZ-AGUILAR, J. A., SIERRA, C., AND WOOLDRIDGE, M. On the logic of normative systems. In *IJCAI* (2007), M. M. Veloso, Ed., pp. 1175–1180.

[42] GROUP, W3C W. Web Services Glossary. http://www.w3.org/TR/ws-gloss/, 2007.

[43] HAN, M., AND HOFMEISTER, C. Modeling and verification of adaptive navigation in web applications. In *Proceedings of the 6th International Conference on Web Engineering* (New York, NY, USA, 2006), ACM, pp. 329–336.

[44] HOLZMANN, G. J. The model checker spin. *IEEE Transactions on Software Engineering 23*, 5 (1997), 279–295.

[45] HU, V., KUHN, D., AND XIE, T. Property verification for generic access control models. In *Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference on* (2008), IEEE Computer Society, pp. 243–250.

[46] HUANG, H., TSAI, W., PAUL, R., AND CHEN, Y. Automated model checking and testing for composite web services. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)* (Los Alamitos, CA, USA, 2005), IEEE Computer Society, pp. 300–307.

[47] HULL, R., BENEDIKT, M., CHRISTOPHIDES, V., AND SU, J. E-services: A look behind the curtain. In *Proceedings of the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 2003), ACM, pp. 1–14.

[48] JAMSHIDI, M. *Systems of Systems Engineering: Principles and Applications*, 1 ed. CRC Press, 2008.

[49] KAVANTZAS, N., BURDETT, D., RITZINGER, G., FLETCHER, T., LAFON, Y., AND BARRETO, C. Web services choreography description language version 1.0. http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109, 2007.

[50] KAZHAMIAKIN, R., PISTORE, M., AND SANTUARI, L. Analysis of communication models in web service compositions. In *Proceedings of the 15th International Conference on World Wide Web* (New York, NY, USA, 2006), ACM, pp. 267–276.

[51] KREGER, H. Web services conceptual architecture (wsca 1.0). Tech. rep., IBM Software Group, May 2001.

[52] LAMANNA, D. D., SKENE, J., AND EMMERICH, W. Slang: A language for defining service level agreements. In *Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 100–107.

[53] LAZOVIK, A., AIELLO, M., AND PAPAZOGLOU, M. Associating assertions with business processes and monitoring their execution. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing* (New York, NY, USA, 2004), ACM Press, pp. 94–104.

[54] LAZOVIK, A., AIELLO, M., AND PAPAZOGLOU, M. Planning and monitoring the execution of web service requests. *International Journal on Digital Libraries 6*, 3 (2006), 235–246.

[55] LOMUSCIO, A., QU, H., AND RAIMONDI, F. Mcmas: A model checker for the verification of multi-agent systems. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 682–688.

[56] LOMUSCIO, A., QU, H., AND SOLANKI, M. Towards verifying compliance in agent-based web service compositions. In *AAMAS (1)* (2008), IFAAMAS, pp. 265–272.

[57] LOMUSCIO, A., QU, H., AND SOLANKI, M. Towards verifying contract regulated service composition. *Autonomous Agents and Multi-Agent Systems 24*, 3 (2012), 345–373.

[58] LU, H., CHAN, W. K., AND TSE, T. H. Testing context-aware middleware-centric programs: A data flow approach and an rfid-based experimentation. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2006), ACM, pp. 242–252.

[59] MAJITHIA, S., WALKER, D., AND GRAY, W. A framework for automated service composition in service-oriented architectures. In *The Semantic Web: Research and Applications*, C. Bussler, J. Davies, D. Fensel, and R. Studer, Eds., vol. 3053 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 269–283.

[60] MALLYA, A. U., AND HUHNS, M. N. Commitments among agents. *Internet Computing, IEEE 7*, 4 (2003), 90–93.

[61] MALLYA, A. U., YOLUM, P., AND SINGH, M. P. Resolving commitments among autonomous agents. In *Advances in Agent Communication, International Workshop on Agent Communication Languages* (Melbourne, Australia, 2003), pp. 166–182.

[62] MCMILLAN, K. L. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[63] MICHAEL, P. P., PAOLO, T., SCHAHRAM, D., AND FRANK, L. Service-oriented computing: State of the art and research challenges. *Computer 40*, 11 (2007), 38–45.

[64] MONGIELLO, M., AND CASTELLUCCIA, D. Modelling and verification of BPEL business processes. In *MDB/MOMPES* (2006), pp. 144–148.

[65] MORGAN, G., PARKIN, S. E., MOLINA-JIMNEZ, C., AND SKENE, J. Monitoring middleware for service level agreements in heterogeneous environments. In *I3E* (2005), M. Funabashi and A. Grzech, Eds., Springer, pp. 79–93.

[66] NAU, D., GHALLAB, M., AND TRAVERSO, P. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., 2004.

[67] PANAGIOTIDI, S., VÁZQUEZ-SALCEDA, J., ALVAREZ-NAPAGAO, S., ORTEGA-MARTORELL, S., WILLMOTT, S., CONFALONIERI, R., AND STORMS, P. Intelligent contracting agents language. In *In Proceedings of the Symposium on Behaviour Regulation in Multi-Agent Systems (BRMAS) at AISB* (2008), pp. 49–55.

[68] PAPAZOGLOU, M. P. *Web Services: Principles and Technology*. Pearson, Prentice Hall, 2008.

[69] PHAM, D. Q., AND HARLAND, J. Temporal linear logic as a basis for flexible agent interactions. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems* (New York, NY, USA, 2007), ACM, pp. 28:1–28:8.

[70] PISTORE, M., BARBON, F., BERTOLI, P., SHAPARAU, D., AND TRAVERSO, P. Planning and monitoring web service composition. In *AIMSA* (2004), pp. 106–115.

[71] PNUELI, A. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1977), IEEE Computer Society, pp. 46–57.

[72] PRISACARIU, C., AND SCHNEIDER, G. A formal language for electronic contracts. In *FMOODS* (2007), pp. 174–189.

[73] RAO, J., AND SU, X. A survey of automated web service composition methods. In *SWSWPC* (2004), pp. 43–54.

[74] ROBBY, DWYER, M. B., AND HATCLIFF, J. Bogor: A flexible framework for creating software model checkers. In *TAIC PART* (2006), P. McMinn, Ed., IEEE Computer Society, pp. 3–22.

[75] ROGIN, F., KLOTZ, T., FEY, G., DRECHSLER, R., AND RLKE, S. Advanced verification by automatic property generation. *IET Computers and Digital Techniques 3*, 4 (2009), 338–353.

[76] ROSSETTI, A. Model checking business processes. *Doctoral Thesis, Universit Politecnica delle Marche.* (2009).

[77] ROUACHED, M., FDHILA, W., AND GODART, C. Web services compositions modeling and choreographies analysis. *International Journal of Web Services Research 2*, 7 (2010), 87–110.

[78] SAVITHA SRINIVASAN, I. B. M., AND VLADIMIR GETOV, U. O. W. Navigating the cloud computing landscape - technologies, services, and adopters. *Computer 44*, 3 (2011), 22–23.

[79] SINGH, M. P. A social semantics for agent communication languages. In *Issues in Agent Communication* (2000), pp. 31–45.

[80] SISTLA, A. P., AND CLARKE, E. The complexity of propositional linear temporal logics. *Journal of ACM 32*, 3 (1985), 733–749.

[81] SKOGAN, D., GROENMO, R., AND SOLHEIM, I. Web service composition in uml. In *Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International* (Sept 2004), pp. 47–57.

[82] SOEKEN, M., KHNE, U., FREIBOTHE, M., FEY, G., AND DRECHSLER, R. Automatic property generation for the formal verification of bus bridges. In *DDECS* (2011), R. Kraemer, A. Pawlak, A. Steininger, M. Schlzel, J. Raik, and H. T. Vierhaus, Eds., IEEE, pp. 417–422.

[83] SOLAIMAN, E., MOLINA-JIMENEZ, C., AND SHRIVASTAV, S. Model checking correctness properties of electronic contracts. In *Service-Oriented Computing - ICSOC 2003*, M. Orlowska, S. Weerawarana, M. Papazoglou, and J. Yang, Eds., vol. 2910 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 303–318.

[84] SOUTER, A. L., AND POLLOCK, L. L. The construction of contextual def-use associations for object-oriented systems. *IEEE Transactions on Software Engineering 29*, 11 (2003), 1005–1018.

[85] SU, J. Web service interactions: Analysis and design. In *Computer and Information Technology, 2005. CIT 2005. The Fifth International Conference on* (Sept 2005), pp. 3–3.

[86] SUN, H., WANG, X., ZHOU, B., AND ZOU, P. Research and implementation of dynamic web services composition. In *Advanced Parallel Processing Technologies*, X. Zhou, M. Xu, S. Jhnichen, and J. Cao, Eds., vol. 2834 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 457–466.

[87] VIEIRA, M., LARANJEIRO, N., AND MADEIRA, H. Benchmarking the robustness of web services. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on* (Dec 2007), pp. 322–329.

[88] WALTON, C. D. Model checking agent dialogues. In *DALT* (2004), pp. 132–147.

[89] WHITE, S. A. Process modeling notations and workflow patterns. http://www.bpmn.org, 2004.

[90] WOOLDRIDGE, M. *An Introduction to MultiAgent Systems*, 2nd ed. Wiley Publishing, 2009.

[91] XING, J., AND SINGH, M. P. Formalization of commitment-based agent interaction. In *Proceedings of the 2001 ACM Symposium on Applied Computing* (New York, NY, USA, 2001), ACM, pp. 115–120.

[92] XING, J., AND SINGH, M. P. Engineering commitment-based multiagent systems: a temporal logic approach. In *AAMAS* (2003), ACM, pp. 891–898.

[93] YAN, J., LI, Z. J., YUAN, W., AND SUN, J. Z. BPEL4WS unit testing: Test case generation using a concurrent path analysis approach. In *ISSRE* (2006), IEEE Computer Society, pp. 75–84.

[94] YEUNG, W. L. A formal and visual modeling approach to choreography based web services composition and conformance verification. *Expert Systems with Applications 38*, 10 (2011), 12772–12785.

[95] YONGHUA, Z., AND HONGHAO, G. A novel approach to generate the property for web service verification from threat-driven model. *Applied Mathematics Information Sciences 8*, 2 (2014), 657–664.

[96] YU, Q., LIU, X., BOUGUETTAYA, A., AND MEDJAHED, B. Deploying and managing web services: issues, solutions, and directions. *The VLDB Journal 17*, 3 (2008), 537–572.

[97] ZHANG, Q., CHENG, L., AND BOUTABA, R. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications 1*, 1 (2010), 7–18.

Appendices

BPEL Process File

```
<bpel:extensionActivity>
    <ext:CommLabel
         name="Comm1" CommID = 1 Debtor = "ws_client_pl" creditor = "ws_PSP_PL">
    </ext:CommLabel>
</bpel:extensionActivity>

<bpel:invoke name="Invoke_C_passing_Software"
      partnerLink="ws_client_pl"  operation="paying" portType="ns2:paying_function"
      inputVariable="ws_Client_agentRequest" outputVariable="ws_Client_agentResponse">
</bpel:invoke>

<bpel:assign validate="no" name="Passing_payment_PSP">
<bpel:copy>
        <bpel:from part="parameters" variable="ws_Client_agentResponse">
            <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
            <![CDATA[ns0:payingReturn]]></bpel:query>
        </bpel:from>
        <bpel:to variable="Client_pay"></bpel:to>
</bpel:copy>

<bpel:copy>
        <bpel:from part="parameters" variable="ws_Client_agentResponse">
            <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
                <![CDATA[ns2:payingReturn]]>
            </bpel:query>
        </bpel:from>
        <bpel:to part="parameters" variable="ws_PSP_PLRequest2">
            <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
                <![CDATA[ns0:result]]>
            </bpel:query>
        </bpel:to>
</bpel:copy>

</bpel:assign>

<bpel:if name="Check_Client_payment">
    <bpel:condition><![CDATA[$Client_pay = "cancel"]]>
    </bpel:condition>

        <bpel:extensionActivity>
            <ext:Tracelabel
                name="Client_Behaviour"
                ServiceName ="ws_client_pl" BehaviourType = 0 >
            </ext:Tracelabel>
        </bpel:extensionActivity>

<bpel:invoke name="Invoke_PSP_to_get_thepayment"
       partnerLink="ws_PSP_PL"  operation="update_with_result"
       portType="ns0:components_request"
       inputVariable="ws_PSP_PLRequest2"
       outputVariable="ws_PSP_PLResponse2">
</bpel:invoke>

<bpel:extensionActivity>
<ext:RecLabel
        name="Rec1" CommID = 1>
        </ext:RecLabel>
</bpel:extensionActivity>

<bpel:exit name="Exit"></bpel:exit>

<bpel:else>

  <bpel:invoke name="Invoke_PSP_to_get_thepayment"
       partnerLink="ws_PSP_PL"  operation="update_with_result"
       portType="ns0:components_request"
```

Figure 1: BPEL Process

ISPL code

```
Agent Environment
Vars:
Main_Buffer: {
                    S13_passing_software , S14_passing_payment_psp,
                    S15_exiting, s16_passing_H_name
              };
T_S_result : boolean;
Output_process : boolean;
Client_pay : {S0_initial,cancel,aaaa};
ws_client_pl_Behaviour : boolean;
end Vars
Actions = { none,  Env_SendMsg_Env_S15};
Protocol:
Main_Buffer = S14_passing_payment_psp
and Client_pay = cancel :{Env_SendMsg_Env_S15};
Other:{none};
end Protocol
Evolution:
Main_Buffer = S13_passing_software
if
( Main_Buffer= S12_Endwhileloop and T_S_result = true
and Action = PSP_SendMsg_ClientS13  );
Main_Buffer = S14_passing_payment_psp and Client_pay = aaaa
if
( Main_Buffer= S13_passing_software and
 ws_client_pl.Action = client_SendMsg_PSP_S14
);
Main_Buffer = S14_passing_payment_psp
and Client_pay = cancel and ws_client_pl_Behaviour = false
if
( Main_Buffer= S13_passing_software
and ws_client_pl.Action = client_SendMsg_PSP_S14_false);
Main_Buffer = S15_exiting and Output_process = false
if
( Main_Buffer = S14_passing_payment_psp and Client_pay = cancel);

Main_Buffer = s16_passing_H_name
if
( Main_Buffer = S14_passing_payment_psp and Client_pay = aaaa
  and ws_client_pl.Action = client_SendMsg_H_S15 );
end Evolution
end Agent
```

Figure 2: Environment ISPL-code

```
Agent ws_client_pl
Lobsvars = { Main_Buffer,Client_pay};
Vars:
Buffer_C_PSP :{ S13_passing_software , S14_passing_payment_psp};
end Vars
Actions = { client_SendMsg_PSP_S14,client_SendMsg_PSP_S14_false,none };
Protocol:
Environment.Main_Buffer = S13_passing_software : {client_SendMsg_PSP_S14
,client_SendMsg_PSP_S14_false};
Environment.Main_Buffer = S14_passing_pay_decision
and Environment.Client_pay = cancel  : {client_SendMsg_PSP_S14_false};
Environment.Main_Buffer = S14_passing_pay_decision
and Environment.Client_pay = aaaa  : {client_SendMsg_PSP_S14};
Other :{none};
end Protocol
Evolution:
Buffer_C_PSP = S13_passing_software
if
( Environment.Main_Buffer= S12_Endwhileloop
and Environment.T_S_result = true
and Environment.Action = PSP_SendMsg_ClientS13
);
Buffer_C_PSP = S14_passing_payment_psp
if
( (Environment.Main_Buffer = S13_passing_software
and Action = client_SendMsg_PSP_S14)
or Environment.Main_Buffer = S13_passing_software
and Action = client_SendMsg_PSP_S14_false);
end Evolution
end Agent
```

Figure 3: Client ISPL-code

```
-----------------------------------------------------------------
Agent ws_PSP_PL
Lobsvars = { Main_Buffer,Client_pay};
Vars:
Buffer_C_PSP :{ S13_passing_software , S14_passing_payment_psp};
end Vars
Actions = { PSP_SendMsg_ClientS13,none };
Protocol:
Environment.Main_Buffer= S12_Endwhileloop
and Environment.T_S_result = true
: {PSP_SendMsg_ClientS13};
Other :{none};
end Protocol
Evolution:

Buffer_C_PSP = S14_passing_pay_decision
if
( (Environment.Main_Buffer = S14_passing_payment_psp
and ws_client_pl.Action = client_SendMsg_PSP_S14)
or Environment.Main_Buffer = S14_passing_payment_psp
and ws_client_pl.Action = client_SendMsg_PSP_S14_false);
end Evolution
end Agent
```

Figure 4: PSP ISPL-code

```
Evaluation
software if Environment.Main_Buffer = S13_passing_software
and ws_client_pl.Buffer_C_PSP = S13_passing_software;
first_payment if Environment.Main_Buffer = S14_passing_payment_psp
and Environment.Client_pay = aaaa ;
cancel if Environment.Main_Buffer = S14_passing_payment_psp
and Environment.Client_pay = cancel ;
C_RED if Environment.ws_client_pl_Behaviour = false;
Blocking_Process if Environment.Main_Buffer = S15_exiting;
end Evaluation

Formulae
E( !software U ( software and (EX (C(ws_client_pl,ws_PSP_PL,first_payment) ))));
E( !software U ( software and (EX (C(ws_client_pl,ws_PSP_PL,cancel) and C_RED))));
AG(C( ws_client_pl,ws_PSP_PL,first_payment) ->
     AF(Fu(ws_client_pl,ws_PSP_PL,first_payment)));
AG( C( ws_client_pl,ws_PSP_PL,cancel) ->
        A( C_RED U(C_RED and Blocking_Process)));
AG (Blocking_Process ->
        EF C(ws_client_pl,ws_PSP_PL,first_payment) );
end Formulae
```

Figure 5: Evaluation ISPL-code

Execution of ISPL code

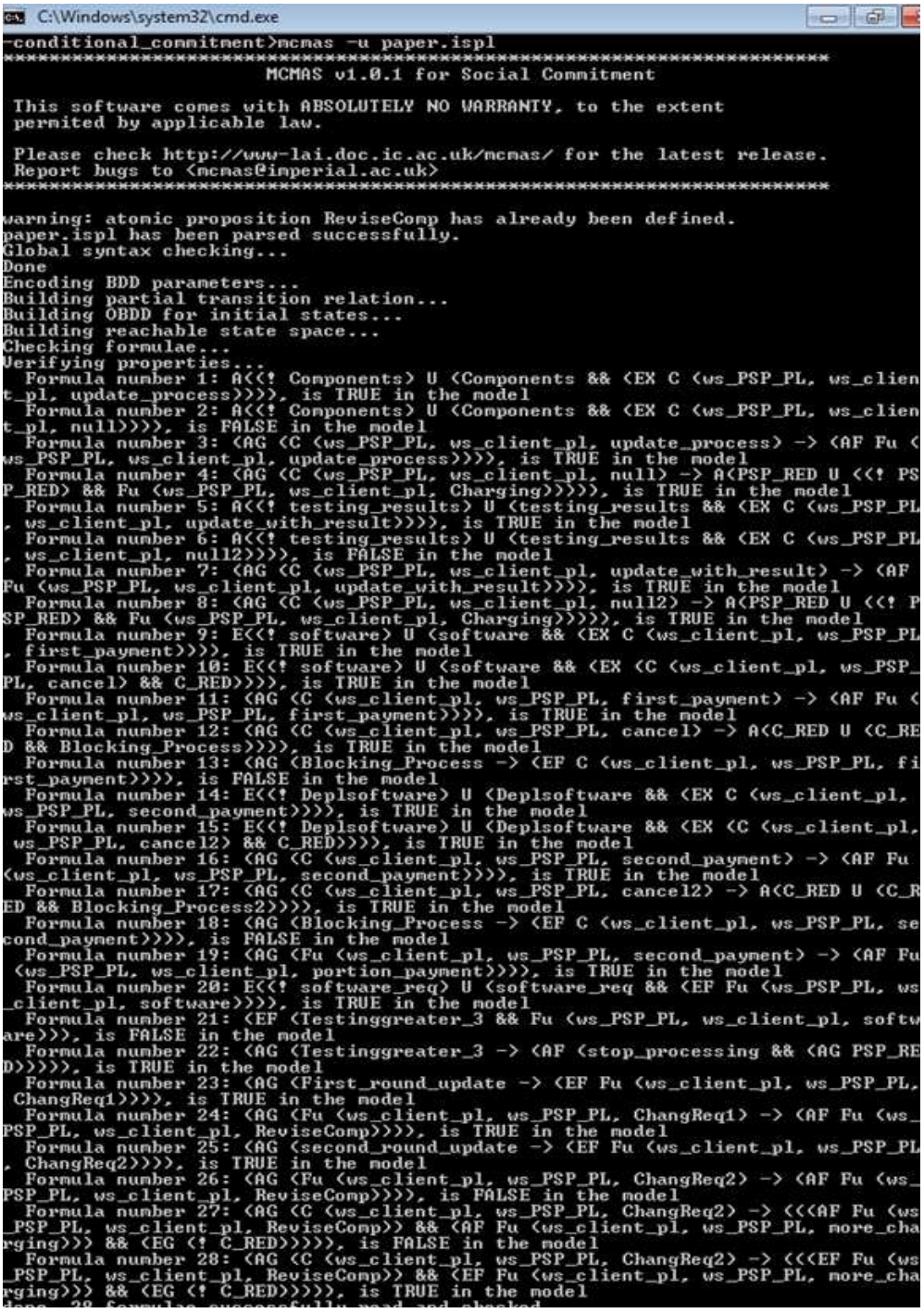


Figure 6: Verification Results

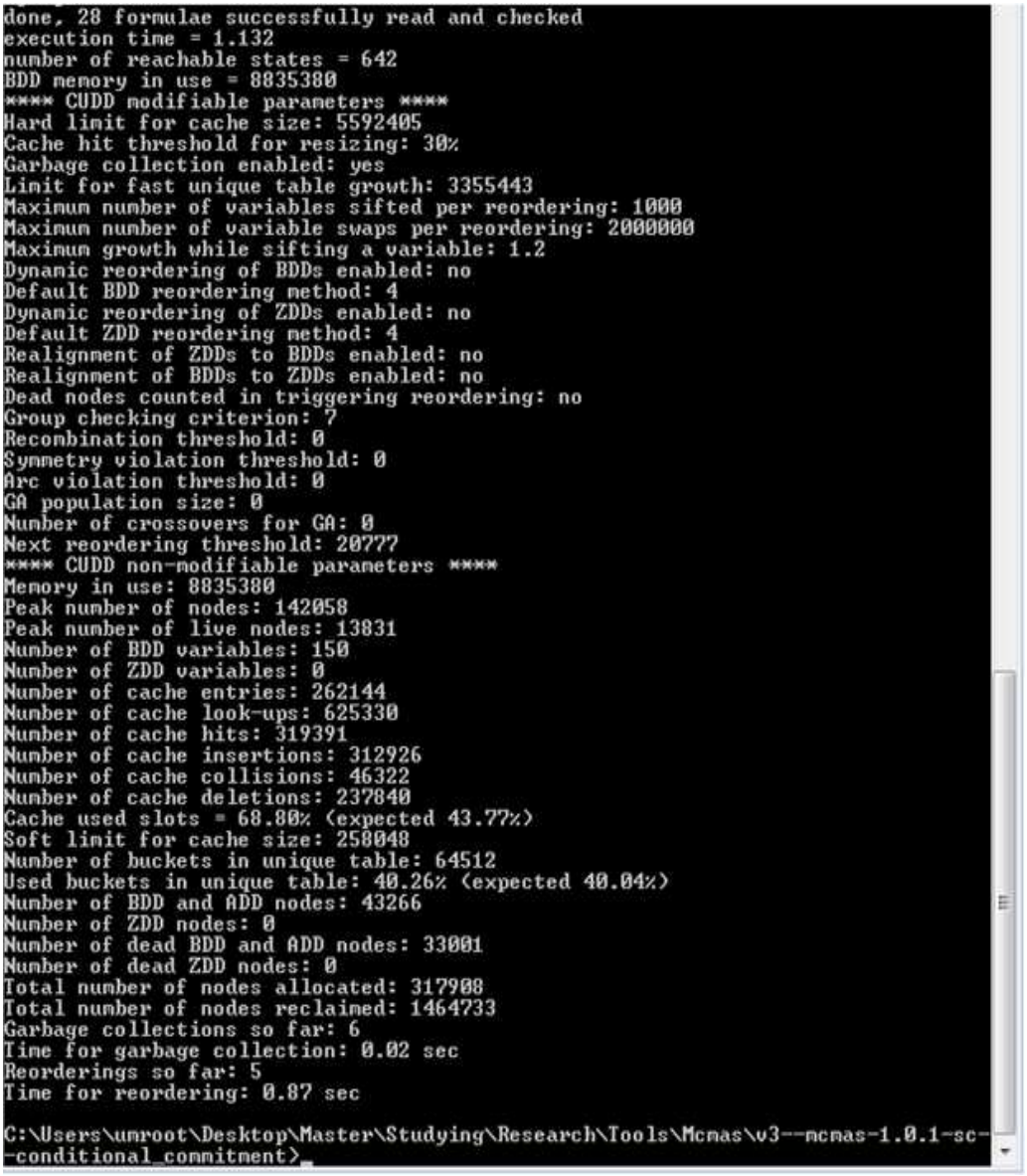


Figure 7: Statistics Results