Injecting Continuous Time Execution into Service-Oriented Computing

Ning Yu

München 2016

Injecting Continuous Time Execution into Service-Oriented Computing

Ning Yu

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Ning Yu
aus Changchun, China

München, den 23. August 2016

Erstgutachter: Prof. Dr. Martin Wirsing
Zweitgutachter: Prof. Dr. Hubert Baumeister
Tag der mündlichen Prüfung: 08.12.2016

Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12.07.11, § 8, Abs. 2 Pkt. .5.)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

Yu Ning

Name, Vorname

München, 23.08.2016

Ort, Datum	Unterschrift Doktorand/in

Formular 3.2

Contents

List of Figures

List of Tables

Abstract

Service-Oriented Computing is a computing paradigm that utilizes services as fundamental elements to support rapid, low-cost development of distributed applications in heterogeneous environments. In Service-Oriented Computing, a service is defined as an independent and autonomous piece of functionality which can be described, published, discovered and used in a uniform way. SENSORIA Reference Modeling Language is developed in the IST-FET integrated project. It provides a formal abstraction for services at the business level.

Hybrid systems arise in embedded control when components that perform discrete changes are coupled with components that perform continuous processes. Normally, the discrete changes can be modeled by finite-state machines and the continuous processes can be modeled by differential equations. In an abstract point of view, hybrid systems are mixtures of continuous dynamics and discrete events. Hybrid systems are studied in different research areas. In the computer science area, a hybrid system is modeled as a discrete computer program interacting with an analog environment.

In this thesis, we inject continuous time execution into Service-Oriented Computing by giving a formal abstraction for hybrid systems at the business level in a Service-Oriented point of view, and develop a method for formal verifications. In order to achieve the first part of this goal, we make a hybrid extension of Service-Oriented Doubly Labeled Transition Systems, named with Service-Oriented Hybrid Doubly Labeled Transition Systems, make an extension of the SENSORIA Reference Modeling Language and interpret it over Service-Oriented Hybrid Doubly Labeled Transition Systems. To achieve the second part of this goal, we adopt Temporal Dynamic Logic formulas and a set of sequent calculus rules for verifying the formulas, and develop a method for transforming the SENSORIA Reference Modeling Language specification of a certain service module into the respective Temporal Dynamic Logic formulas that could be verified. Moreover, we provide a case study of a simplified small part of the European Train Control System which is specified and verified with the approach introduced above.

We also provide an approach of implementing the case study model with the IBM Websphere Process Server, which is a comprehensive Service-Oriented Architecture integration platform and provides support for the Service Component Architecture programming model. In order to realize this approach, we also provide functions that map models specified with the SENSORIA Reference Modeling Language to Websphere Process Server applications.

Zusammenfassung

„Service-Oriented Computing" (SOC) ist ein Berechnungsparadigma, das Services als elementare Elemente verwendet, um so die schnelle und kostengünstige Entwicklung von verteilten Anwendungen in heterogenen Umgebungen zu unterstützen. Im Kontext von SOC wird ein Service als eine unabhängige, autonome Funktionalität definiert, die auf einheitliche Weise beschrieben, veröffentlicht, dargestellt und verwendet werden kann. Die „SENSORIA Reference Modeling Language" wurde im Rahmen des IST-FET Projekts SENSORIA entwickelt. Sie stellt eine formale Abstraktion für Services auf Geschäftsebene zur Verfügung.

Hybride Systeme entstehen in eingebetteten Steuerungssystemen, wenn Komponenten, die diskrete Änderungen vornehmen, mit Komponenten, die stetig fortlaufende Prozesse ausführen, kombiniert werden. Im Normalfall können diskrete Änderungen durch endliche Zustandsmaschinen und fortlaufende Prozesse durch Differentialgleichungen modelliert werden. Abstrakt betrachtet sind hybride Systeme eine Mischung aus fortlaufenden Prozessen und diskreten Ereignissen. Hybride Systeme werden in unterschiedlichen wissenschaftlichen Bereichen betrachtet. In der Informatik wird ein hybrides System als ein diskretes Computerprogramm, das mit der analogen Umwelt interagiert, modelliert.

In dieser Arbeit wird kontinuierliche Ausführung in SOC eingebunden. Dazu wird von einem Service-orientierten Standpunkt aus eine formale Abstraktion für hybride Systeme auf Geschäftsebene gegeben und eine Methode für formale Verifikationen entwickelt. Um den ersten Teil dieses Ziels zu erreichen, erstellen wir eine hybride Erweiterung von „Service-Oriented Doubly Labeled Transition Systems", die wir „Service-Oriented Hybrid Doubly Labeled Transition Systems" nennen, und eine Erweiterung der „SENSORIA Reference Modeling Language" und interpretieren diese durch „Service-Oriented Hybrid Doubly Labeled Transition Systems". Um den zweiten Teil unseres Ziels zu erreichen, verwenden wir „Temporal Dynamic Logic" Formeln und eine Menge von Regeln im Stile eines Sequenzenkalküls zur Verifikation der Formeln und wir entwickeln eine Methode zur Überführung der „SENSORIA Reference Modeling Language" Spezifikation eines bestimmten Service-Moduls in die Formeln der „Temporal Dynamic Logic", welche verifiziert werden können. Wir illustrieren den Ansatz anhand einer Fallstudie des „European Train Control System", die mit dem in dieser Arbeit entwickelten Ansatz spezifiziert und verifiziert wird.

Außerdem implementieren wir das Modell der Fallstudie mit dem IBM Websphere Process Server, einer integrierten Plattform für Service-orientierte Architekturen, die das Programmiermodell der „Service Component Architecture" unterstützt. Zur Realisierung dieses Ansatzes stellen wir Funktionen bereit, die Modelle, die mit der „SENSORIA Reference Modeling Language" spezifiziert wurden, auf Websphere Process Server Applikationen abbilden.

Acknowledgements

First of all, I would like to express my sincere appreciation to my supervisor, Prof. Martin Wirsing, who helped me greatly during the whole period of my PhD study. In the academic aspect, he supervised me a lot by offering me uncountable times of discussions, from which I learned much knowledge and experience in my research area. As a foreign student, in the administrative aspect I received much support from Prof. Wirsing on the necessary documentations and facility, which enable me to concentrate on my thesis but not to be disturbed too much by other affairs. Moreover, many valuable personalities of Prof. Wirsing, such as rigorousness, the dedication to work, open-mindedness, patience and so on, give me deep impression and I think that I will always benefit from them.

Secondly, I greatly appreciate the support provided by Prof. Rolf Hennicker. He did not only give me good suggestions for my work, but also helped me to get used to the life in Munich by introducing to me a lot of interesting things and places around from time to time. His kindness and humor made me feel at home.

Moreover, I wish to express many thanks to Prof. Hubert Baumeister, the second reviewer of my PhD thesis. He gave me many valuable feedbacks to my thesis, and also traveled from Denmark to Munich for my thesis disputation. He really helped much in the finishing of my thesis.

Additionally, I profusely thank Ms. Marianne Diem and Mr. Anton Fasching. They provided me strong support on my personal affairs in the PST group, from making appointments to arranging a computer for work. They handled these affairs always in time such that I was never delayed.

Also, I am grateful to the crew of the PST group, such as Nora Koch and Joschka Rinke, who shared the same office with me and were always willing to give me a hand; Marianne Bush, Annabelle Klarl and Lenz Belzner, who helped me in the group activities and shared with me the experience in doing a PhD work.

At last but not least, I thank my parents deeply. For so long time, they never stopped encouraging me and bringing me confidence. Without their support, I would never be able to finish my thesis.

Chapter 1

Introduction

In computer science, the term "service" refers to a software functionality or a set of software functionalities that can be reused by different clients for different purposes, together with the policies that should control its usage. Service-Oriented Computing aims at solving the problems arose in the construction and application of services. As services with various functionalities are required, different types of execution need to be injected into Service-Oriented Computing. Continuous time execution is one of the most important and needed types since it appears in various systems that call for services to accomplish certain tasks.

To inject continuous time execution into Service-Oriented Computing, we choose hybrid systems which could yield continuous time execution and study their features in a Service-Oriented context. Thus in this chapter, we introduce Service-Oriented Computing and hybrid systems respectively. Moreover, we give an overview of this thesis.

1.1 Service-Oriented Computing and SENSORIA Reference Modeling Language

Service-Oriented Computing (SOC) is a computing paradigm that utilizes services as fundamental elements to support rapid, low-cost development of distributed applications in heterogeneous environments [1]. In SOC, a service is defined as an independent and autonomous piece of functionality which can be described, published, discovered and used in a uniform way. Services upon a SOC-based infrastructure are loosely coupled and can flexibly crete dynamic business processes and agile applications that may span organizations and computing platforms. These business processes and agile applications, or so called orchestrations, rebuild the services to form new, higher-level functionality. Thus a SOC-based infrastructure can adapt quickly and autonomously to changing mission requirements and environment.

Realizing SOC requires the development of Service-Oriented Architectures (SOAs)[2], which is an architectural paradigm describing the fundamental design of service-oriented software. SOA prescribes that all functions of a SOA-based application are provided as services and is designed independent of any specific technology. In particular, SOA requires services to be self-contained, platform-independent and dynamically discoverable, invocable and composable. The services in a SOA-based application can be new functions, or functions that already exist and are wrapped by the service implementation. Therefore, SOA permits composing new business functions and processes by creating, deploying and integrating multiple and heterogeneous implemented services. SOAs can be implemented using different technologies such as Web Services [3], Grid Computing [4], the OSGi framework [5], the Microsoft Distributed Component Object Model (DCOM) [6] and the OMG Common Object Request Broker Architecture (CORBA) [7]. Among these technologies, Web Services are the most preferred implementation, since they support distributed and loosely coupled applications on existing and ubiquitous infrastructure such as HTTP, SOAP and XML. In Web Services, requested operations are implemented using one or more Web Service components. Web Service components [8] are normally hosted within a Web Services container [9] which serves as an interface between business services and low-level infrastructure services.

To implement SOA, there yields solutions in which different patterns of organization of service roles are designed. We introduce three such patterns by explaining how the respective service roles are organized as follows:

1. The service requester-provider pattern
   Service requester and service provider communicate via service request, which are messages formatted according to the Simple Object Access Protocol (SOAP) [10]. SOAP entails a light-weight protocol allowing RPC-like calls over the Internet using a variety of transport protocols including HTTP, HTTP/S and SMTP. Web Services is a solution which adopts the service requester-provider pattern. In a Web Service, SOAP request is received by a runtime service provider that accepts the SOAP message, extracts the XML message body, transforms the XML message into a HTTP protocol that is native to the requested service and delegates the request to the actual function or business process within an enterprise.

2. The service aggregator pattern
   When interactions between service requesters and service providers involve discovering/publishing, negotiating, reserving and utilizing services from potentially different service providers, they can be complex. The service aggregator pattern is brought forward to reduce such complexity. A service aggregator [11] is a role in which the service provider and requester functionality is combined. A service aggreator acts both as an application service provider which offers a complete solution by creating composite, higher-level services using specialized composition languages such as BPEL [12] and BPML [13], and as a service requester by requesting and reserving services from other service providers.

3. The service broker pattern
   The service broker pattern is brought forward when a service requester needs to select a specific application service provider. Service brokers [14] are trusted parties that force service providers to adhere to information practices that comply with privacy laws and regulations or industry best practices. Within such a pattern, service providers are registered to a service broker. When a service requester needs to call services with specific functionalities, it looks up the registry information of service providers in the service broker and finds the most

suitable one, then sets up communication with the selected service provider. UDDI [15] and SAML [16] are different solutions of service brokers.

SENSORIA is a IST-FET integrated project which develops methodologies and tools for dealing with SOC. Its main goal is to produce new knowledge for systematic and scientifically well-founded methods of service-oriented software development. The SENSORIA Reference Modeling Language (SRML [17]) has been developed in SENSORIA, and it is a prototype domain-specific language for modeling service-oriented systems at the business level [18] abstraction. SRML adopts a set of primitives which are used specifically for modeling the business conversations that occur in SOC, and services in SRML are characterized by the conversations that they support and the properties of those conversations. These properties of conversations, also known as conversation protocols, do not need to be modeled explicitly — they are assumed to be provided by the underlying SOA. In this thesis, we consider the following aspects of SRML:

- *The semantic domain:* In SRML, the execution of services are modeled with transition systems in which the transitions represent the exchange and processing of events of each component involved in the delivery of that service. Thus, the semantic domain of SRML includes configurations which specify the interactions and events that are involved in the communication between service roles, and paths of transition systems over which the executions of services are interpreted.
- *The formal specifications:* In SRML, each component of a service is specified by declaring the interactions in which the component can be involved and the properties that can be observed as the result of performing these interactions. Each wire linking any two components is specified by declaring the interactions between the two components locally.
- *Service assembly:* Since the composition of services in SRML adopts the SCA [19] assembly model, services in SRML can be created by interconnecting a set of elementary components to external services. The business logic of such a service is given by its formal specifications and is independent of the business logic of the external services. The external services are discovered and linked to the service at run time.

A detailed introduction to SRML is given in Chapter 3.

1.2 Hybrid System and its formal models

Hybrid systems arise in embedded control when discrete components are coupled with continuous components. Discrete components are for example digital controllers, computers and subsystems that can be modeled by finite-state machines. And continuous components are controllers and plants modeled by partial or ordinary differential equations or difference equations[20]. Automated highway systems [21], flight control and management systems [22, 23] and constrained robotic systems [24] are examples of hybrid systems. In general, hybrid systems are mixtures of real-time (continuous) dynamics and discrete events. Setting out from different purposes, people study hybrid systems mainly in the following three types: i) In the *computer science* area, hybrid system is modeled as a discrete (computer) program interacting with an analog environment. A leading objective is to extend standard program analysis techniques to systems which incorporate some kind of continuous dynamics. ii) In the *modeling and simulation* area, hybrid systems can often operate in different modes, and the transition from one mode to another sometimes can be idealized as an instantaneous, discrete transition. iii) In the *systems and control* area, hybrid systems are modeled as hierarchical systems with a discrete decision layer and a continuous implementation layer, or switching control schemes and relay control. The hybrid systems discussed in this thesis are of the second type, since in our approach we model hybrid systems in a higher level of abstraction than the level of computer programs.

From a general system-theoretic point of view, hybrid systems can be seen as systems with communication ports and physical ports, the former associate with variables that are symbolic in nature and the latter associate with continuous variables that are related to physical measurement. Thus a hybrid system can be regarded as a combination of discrete or symbolic dynamics and continuous dynamics. The main difficulty for modeling hybrid systems is the specification of interactions between symbolic and continuous dynamics, since this involves a variety of mathematical and engineering disciplines such as differential geometry, differential and difference equations, optimal control, automata (programs) theory, discrete event systems, data structures and computation. There are three main frameworks for modeling hybrid systems: hybrid automata [25], hybrid time evolutions and hybrid behavior, and event-flow formulas [26]. Different frameworks of the same hybrid system have different properties, depending on the purpose one wants to use it for. Hybrid automata are graph-related and provide workable representations of hybrid systems; further more, the semantics of hybrid automata is very explicit: all the locations and all the transitions from one location to another, together with all their guards and jumps are included in the semantics. Hybrid time evolutions and hybrid behavior of a hybrid system project the hybrid behavior on the behavior of the continuous variables and thus define a continuous-time behavior; they are useful from a conceptual point of view and for theoretical purposes. Event-flow formulas are equation-based and describe the various activities or modes of hybrid systems; the setting of event-flow formulas is close to that of some simulation languages such as Modelica$^{\text{TM}}$ [27], thus could easily be implemented with these languages. In this thesis, hybrid automata is related and will be introduced in Chapter 2. Also in this thesis, we aim at the formal verification of safety and guarantee properties of hybrid systems using theorem proving method.

1.3 Thesis Overview

In order to inject continuous time execution into SOC, in this thesis, we aim at providing a formal specification for hybrid systems in a Service-Oriented point of view and developing a method of verifying such hybrid systems formally. Specifically, we make a hybrid extension of SRML (introduced in Section 1.1). The syntax of SRML is extended by adding notations of the first-order derivatives of variables to time, which can be then used to form first-order ordinary differential equations that are used to describe the continuous evolutions of hybrid systems. The semantics of SRML is extended by giving a hybrid extension of the "Service-Oriented Doubly

Labeled Transition System", named as "Service-Oriented Hybrid Doubly Labeled Transition System" (see Section 4.1), over which SRML is interpreted.

We also provide a method for formally verifying the properties of hybrid systems specified with the SRML extension. This method includes the transformation from SRML specifications to the respective hybrid programs that could be verified (see Section 5.1), and verification using a set of sequent calculus rules for the logic dTL (see Section 5.2). To show how this method works, we present a case study of the European Train Control System.

Furthermore, to implement hybrid systems specified with SRML extension, we develop a set of mappings from the SRML specification to the implementation environment: IBM WebSphere Integration Developer and WebSphere Process Server. We also implement the model of a small part European Train Control System using the set of mappings and test the result in the implementation environment.

The rest of the thesis is arranged as follows:

In Chapter 2, we introduce the background knowledge of our approach. It includes: differential equations, which are used to describe the continuous evolutions of hybrid systems; hybrid automata, by which the transformation from the properties of the hybrid system specified with SRML extension to the Temporal Dynamic Logic formulas that could be verified is inspired; Computational Tree Logic and First-Order Dynamic Logic, the combination of which yields Temporal Dynamic Logic.

In Chapter 3, we review the basic concepts and the semantic domain of SRML. In SRML, services are specified as service modules and are the basic units to perform the business logic. We review the basic concepts of SRML by introducing the basic compositions of a SRML service module, which includes service components (moedeled by business roles), service interfaces (modeled by business protocols) and wires (modeled by interaction protocols and connectors). Interactions and the associated events which model the communications between different service components or between service components and service interfaces are also introduced. We review the semantic domain of SRML by introducing Service-Oriented Configurations and Service-Oriented Doubly Labeled Transition Systems, which composite the semantic domain of SRML.

In Chapter 4, we provide the hybrid extension of SRML. We first extend the semantic domain of SRML by defining the "Service-Oriented Hybrid Doubly Labeled Transition System", which is a hybrid extension of the Service-Oriented Doubly Labeled Transition Systems. Then we extend the syntax of each SRML compositions and interpreted the extended syntax over Service-Oriented Hybrid Doubly Labeled Transition Systems. Finally we formalize SRML service modules based on these definitions. The main part of Chapter 4 is similar to that in [28] (also see Appendix 3).

In Chapter 5, we provide the method for formally verifying the properties of hybrid systems that are specified with SRML extension. We first transform these properties to Temporal Dynamic Logic formulas by defining the SRML based finite automata, from which the hybrid programs that are needed to construct these formulas can be obtained by applying Brzozowski's method. Then we define a set of sequent calculus that are adapted from [29] for verifying Temporal Dynamic Logic formulas. Finally we show a case study about part of the European Train Control System. The verification of the model in the case study is similar to that in [28] (also see

Appendix 3).

In Chpater 6, we focus on the implementation of service modules that are specified with SRML extension. We choose IBM WebSphere Process Server and IBM WebSphere Integration Developer as the implementing tools. We use IBM WebSphere Integration Developer to code and assemble a service module, and use IBM WebSphere Process Server to deploy and test the module that is developed with IBM WebSphere Integration Developer. In order to make the implementation automatic, we provide a set of mappings form the SRML extension domain to the implementation domain. These mappings are defined as functions that assign SRML notations to notations in the implementing domain. Finally we show the implementation of the model in our case study that is presented in Chapter 5.

In Chapter 7, we make a summery of this thesis and look ahead to the future prospects.

Chapter 2

Background knowledge

In this chapter, we review the background knowledge that is related to our work. The background knowledge includes: differential equations which are used to describe the time-continuous processes of the behaviors of hybrid transition systems, hybrid automata which provide a basic framework for modeling hybrid transition systems and the logic basis with which properties of hybrid transition systems can be verified.

2.1 Differential Equations

A differential equation is a mathematical equation for an unknown function of one or several variables. The equation relates the values of the function itself and derivatives of that function with various orders. Differential equations play a basic and important role in many fields such as engineering, physics, and economics. In these fields, when the relationship of some continuously varying quantities and their changing rate is known, differential equations about the functions that modeled the quantities and the derivatives that model the rates can be applied to formally express this relationship. E.g., Newton's laws in classical mechanics can be expressed by a differential equation, in which the position of a moving body is modeled by a function of time, and the velocity and acceleration of the body are modeled by the first order and second order derivatives to time of that function.

Mathematically, a *differential equation* is a relationship between an independent variable x and a dependent variable y, and one or more derivatives of y with respect to x, and the order of a differential equation is given by the highest derivative involved in the equation [30]. The following equation is the differential equation of Newton's second law:

$$m\frac{\partial^2 y}{\partial t^2} = -ky \tag{2.1}$$

In function 2.1, m denotes the mass of a moving body, y denotes the displacement of the body, t denotes time, and k denotes the force constant. This is a differential equation about the displacement of the body and the second derivative of the displacement to time (the acceleration of the body) and is of second order. Usually, this second-order differential equation is used when the function $y(t)$ is unknown.

The solutions of a differential equation are functions that make the differential equation true. There are two kinds of solutions of a differential equation: a *general solution* which is a function including one or several arbitrary constants (the number of the constants in a general solution is the same as the order of that differential equation), and infinitely many *exact solutions* which are functions without any arbitrary constants. Function 2.2 is the general solution of Function 2.1.

$$y = -\frac{1}{2}kt^2 + C_1 t + C_2 \tag{2.2}$$

where C_1 and C_2 are the arbitrary constants. An exact solution of Function 2.1 can be obtained when the initial values of speed and displacement of the moving body are given.

Among differential equations, only the simplest ones have solutions that can be expressed with explicit functions and can be obtained with mathematical methods (such as *direct integration method* and *variable separation method*). Most of the differential equations can only be solved numerically to get approximated solutions. Normally this can be done computer programs.

2.2 Hybrid Automata

A hybrid automaton is a formal model of a hybrid system, which is a dynamical system with both discrete and continuous components. In a hybrid automaton, the discrete states are modeled by the vertices of a graph, the discrete dynamics is modeled by the edges of the graph, the continuous states are modeled by points in $\mathbb{R}^n$, and the continuous dynamics is modeled by flow conditions such as differential equations. The definition of hybrid automaton is from [25] is shown as follows:

Definition 2.2.1 (Hybrid automata)**.** A *hybrid automaton H* consists of the following components.

Variable. A finite set $X = \{x_1, \dots x_n\}$ of real-numbered variables. The number n is called the *dimension* of H. We write $\dot{X}$ for the set $\{\dot{x_1}, \dots \dot{x_n}\}$ of dotted variables (which represent first derivatives during continuous change), and we write X' for the set $\{{x_1}', \dots {x_n}'\}$ of primed variables (which represent values at the conclusion of discrete change).

Control graph. A finite directed multigraph (V, E). The vertices in are called *control modes*. The edges in E are called *control switches*.

Initial, invariant, and flow conditions. Three vertex labeling functions *init*, *inv*, and *flow* that assign to each control mode $v \in V$ three predicates. Each initial condition $init(v)$ is a predicate whose free variables are from X. Each invariant condition $inv(v)$ is a predicate whose free variables are from X. Each flow condition $flow(v)$ is a predicate whose free variables are from $X \cup \dot{X}$.

Jump conditions. An edge labeling function *jump* that assigns to each control switch $e \in E$ a predicate. Each jump condition *jump(e)* is a predicate whose free variables are from $X \cup X'$.

Events. A finite set Σ of events, and an edge labeling function $event : E \longrightarrow \Sigma$ that assigns to each control switch an event.

2.2.1 Safe Semantics of Hybrid Automata

The execution of a hybrid system results in continuous change and discrete change. The mixed discrete-continuous dynamics is abstracted by a fully discrete transition system: the labeled transition system. The definition of labeled transition systems is from [25], and is shown as follows:

Definition 2.2.2 (Labeled transition systems)**.** A *labeled transition system S* consists of the following components.

Variable. A (possibly infinite) set Q of states, and a subset $Q^0 \subseteq Q$ of initial states.

Control graph. A (possibly infinite) set A of labels, and for each label $a \in A$, a binary relation $\xrightarrow{a}$ on the state space Q. Each triple $q \xrightarrow{a} q'$ is called a transition.

The finite behaviors of a hybrid system are called the safe assumptions, thus the finite sequences of transitions of a labeled transition system yield the safe semantics of hybrid automata. For a given hybrid automaton, there are two labeled transition systems defined: the time transition system and the time-abstract transition system. The former abstracts continuous flows by transitions, retaining only information about the source, the target and the duration of each flow; the latter abstracts also the duration of flows. Both transition systems represent discrete jumps by transitions. The definitions of the two transition systems are from [25], and are shown as follows:

Definition 2.2.3 (Transition semantics of hybrid automata)**.** The timed transition system S^t_H of the hybrid automaton H is the labeled transition system with the components Q, Q^0, A and $\xrightarrow{a}$ for each $a \in A$, defined as follows:

- Define $Q, Q^0 \subseteq V \times \mathbb{R}^n$ such that $(v,x) \in Q$ iff the closed predicate $inv(v)[X := x]$ is true, and $(v,x) \in Q^0$ iff both $init(v)[X := x]$ and $inv(v)[X := x]$ are true;

- $A = \Sigma \cup \mathbb{R}_{\geq 0}$;

- For each event $\sigma \in \Sigma$, define $(v,x) \xrightarrow{\sigma} (v',x')$ iff there is a control switch $e \in E$ such that (1) the source of e is v and the target of e is v', (2) the closed predicate $jump(e)[X,X' := x,x']$ is true, and (3) $event(e) = \sigma$;

- For each nonnegative real $\delta \in \mathbb{R}_{\geq 0}$, define $(v,x) \xrightarrow{\delta} (v',x')$ iff $v = v'$ and there is a differentiable function $f : [0,\delta] \to \mathbb{R}^n$, with the first derivative $\dot{f} : (0,\delta) \to \mathbb{R}^n$, such that (1) $f(0) = x$ and $f(\delta) = x'$, and (2) for all reals $\varepsilon \in (0,\delta)$, both $inv(v)[X := f(\varepsilon)]$ and $flow(v)[X,\dot{X} := f(\varepsilon),\dot{f}(\varepsilon)]$ are true.

The timed transition system S_H^a of the hybrid automaton H is the labeled transition system with the components Q, Q^0, B and $\xrightarrow{\mathrm{b}}$ for each $b \in B$, defined as follows:

- Q and Q^0 are defined as above;
- $B = \Sigma \cup \{\tau\}$, for some event $\tau \notin \Sigma$;
- For each event $\sigma \in \Sigma$, define $\xrightarrow{\sigma}$ as above;
- Define $(v,x) \xrightarrow{\tau} (v',x')$ iff there is a nonnegative real $\delta \in \mathbb{R}_{\geq 0}$ such that $(v,x) \xrightarrow{\delta} (v',x')$.

2.2.2 Live Semantics of Hybrid Automata

When considering the infinite behavior of a hybrid automaton, the infinite sequences of transitions which do not converge in time need to be defined. The divergence of time is a liveness assumption. A hybrid automaton is nonzeno if it is live. The definitions of live transition systems and the trace semantics are from [25] and are shown as follows:

Definition 2.2.4 (Live transition systems)**.** Consider a labeled transition system S and a state q_0 of S. A q_0*−rooted trajectory* of S is a finite or infinite sequence of pairs $\langle a_i, q_i \rangle_{i \geq 1}$ of labels $a_i \in A$ and states $q_i \in Q$ such that $q_{i-1} \xrightarrow{\mathrm{a_i}} q_i$ for all $i \geq 1$. If q_0 is an initial state of S, then $\langle a_i, q_i \rangle_{i \geq 1}$ is an *initialized trajectory* of S. The set L of infinite initialized trajectories is *machine-closed* for S if every finite initialized trajectory of S is a prefix of some trajectory in L (with the assumption that every initial state of S has a successor state). If (S,L) is a live transition system, and $\langle a_i, q_i \rangle_{i \geq 1}$ is either a finite initialized trajectory os S or a trajectory in L, then the corresponding sequence $\langle a_i \rangle_{i \geq 1}$ of labels is called a (finite or infinite) trace of (S,L).

Definition 2.2.5 (Trace semantics of hybrid automata)**.** We associate with each transition of the timed transition system S_H^t a *duration* in $\mathbb{R}_{\geq 0}$. For events $\sigma \in \Sigma$, the duration of $q \xrightarrow{\sigma} q'$ is 0. For reals $\delta \in \mathbb{R}_{\geq 0}$, the duration of $q \xrightarrow{\delta} q'$ is δ. An infinite trajectory $\langle a_i, q_i \rangle_{i \geq 1}$ of the timed transition system S_H^t *diverges* if the infinite sum $\sum_{i \geq 1} \delta_i$ diverges, where each δ_i is the duration of the corresponding transition $q_{i-1} \xrightarrow{\mathrm{a_i}} q_i$. An infinite trajectory $\langle b_i, q_i \rangle_{i \geq 1}$ of the time-abstract transition system S_H^a *diverges* if there is a divergent trajectory $\langle a_i, q_i \rangle_{i \geq 1}$ of S_H^t such that for all $i \geq 1$, either $a_i = b_i$ or $a_i, b_i \notin \Sigma$. Let L_H^t be the set of divergent initialized trajectories of the timed transition system S_H^t, and let L_H^a be set of divergent initialized trajectories of the time-abstract transition system S_H^a. The hybrid automaton H is *nonzeno* if L_H^t is machine-closed for S_H^t (or equivalently, L_H^a is machine-closed for S_H^a). Each trace of the live transition system (S_H^t, L_H^t) is called a *timed trace* of H, and each trace of the live transition system (S_H^a, L_H^a) is called a *time-abstract trace* of H.

2.3 Computational Tree Logic

Computational Tree Logic (CTL) is a type of propositional branching-time temporal logic. It was proposed in [31, 32] and is closely related to branching-time logics proposed in [33, 34, 35].

CTL allows basic temporal operators of the form of a path quantifier—either A ("for all futures") or E ("for some future")—followed by a single one of the usual linear temporal operators G ("always"), F("sometime"), X ("nexttime"), or U ("until"). The definition of CTL formulas is from [32] and is defined as follows:

Definition 2.3.1 (CTL Formulas)**.** The set of CTL state formulas SF and path formulas PF are inductively defined as the smallest set such that:

- If $p \in AP$, then $p \in SF$;
- If $p, q \in SF$, then $p \wedge q, \neg p \in SF$;
- If $p \in PF$, then $\mathsf{E}p, \mathsf{A}p \in SF$;
- If $p, q \in SF$, then $\mathsf{X}p, p\mathsf{U}q \in PF$.

CTL is interpreted over the temporal structure $M = (S, R, L)$where

- S is the set of *states*;
- R is a total *binary relation* $\subseteq S \times S$ such that: $\forall s \in S, \exists s' \in S$, there is $(s, s') \in R$;
- L:$S \rightarrow$PowerSet(AP) is a labeling which associates with each state s an interpretation $L(s)$ of all atomic proposition symbols that are true at state s, where AP is the underlying set of atomic proposition symbols.

A *fullpath* of M is an infinite sequence $s_0, s_1, s_2, \ldots$ of states such that for $\forall i, (s_i, s_{i+1}) \in R$. $x = (s_0, s_1, s_2, \ldots)$ denotes a fullpath and $x^i = (s_i, s_{i+1}, s_{i+2}, \ldots)$ denotes the suffix path of x. The interpretation of CTL formulas is from [32] and is shown as follows:

Definition 2.3.2 (Interpretation of CTL Formulas)**.** Given a temporal structure M, the interpretation of CTL formulas $\models$ is inductively defined as follows:

- $M, s_0 \models p$ iff $p \in L(s_0)$;
- $M, s_0 \models p \wedge q$ iff $M, s_0 \models p$ and $M, s_0 \models q$;
- $M, s_0 \models \neg p$ iff not($M, s_0 \models p$);
- $M, s_0 \models \mathsf{E}p$ iff $\exists$ fullpath $x = (s_0, s_1, s_2, \ldots)$ in $M, M, x \models p$;
- $M, s_0 \models \mathsf{A}p$ iff $\forall$ fullpath $x = (s_0, s_1, s_2, \ldots)$ in $M, M, x \models p$;
- $M, x \models p$ iff $M, s_0 \models p$;
- $M, x \models p \wedge q$ iff $M, x \models p$ and $M, x \models q$;
- $M, x \models \neg p$ iff not($M, x \models p$);
- $M, x \models p \cup q$ iff $\exists i[M, x^i \models q$ and $\forall j (j < i$ implies $M, x^j \models p)]$;

- $M,x \models \mathsf{X}p$ iff $M,x^1 \models p$.

UCTL [36, 37] is a UML-oriented branching-time temporal logic which adopts the temporal and boolean operators of CTL and is interpreted over *Doubly Labeling Transition Systems* (introduced in Chapter 3).

2.4 First-Order Dynamic Logic

Dynamic Logic [38] is an extension of modal logic originally intended for reasoning about computer programs and later applied to more general complex behaviors arising in linguistics, philosophy, AI, and other fields. It is a successful approach for deductively verifying(infinite-state) systems. The temporal dynamic logic dTL is a combination of First-Order Dynamic Logic and CTL. It provides modalities for quantifying over traces of hybrid systems and will be introduced in Chapter 4.

To show the syntax and semantics of Dynamic Logic, we first introduce the concept of *program*. A *program* is a recipe written in a formal language for computing desires output data from given input data. Programs normally use *variables* to hold input and output values and intermediate results. Each variable can assume values from a *specific domain* of computation, which is a structure consisting of a set of data values along with certain distinguished constants, basic operations, and tests that can be performed on those values. A *state* of a *program* is a function that assigns a value to each program variable. The value of variable x must belong to the domain associated with x. In logic, such a function is called a *valuation*. Since our work only related with the basic version of First-Order Dynamic Logic (here denoted as *b-DL* for simplicity), we only introduce the syntax and semantics of this part, but not versions of R.E. Programs, Arrays, Stacks and Wildcard (see [38]).

2.4.1 Syntax of b-DL

Let $\Theta = \{f,g,\ldots,p,r,\ldots\}$ be a finite first-order vocabulary. Here f and g denote typical function symbols of Θ, and p and r denotes typical relation symbols. Associated with each function and relation symbol of Θ is a fixed *arity* (number of arguments). Functions and relations of arity 0, 1, 2, 3 and n are called *nullary*, *unary*, *binary*, *ternary*, and *n-ary*, respectively. The equality symbol = is assumed to be always included in Θ and has the arity 2. Nullary functions are also called *constants*. A countable set of individual variables is denoted by $V = \{x_0,x_1,\ldots\}$. The atomic formulas and programs of b-DL (basic version of DL) are from [38] and are shown as follows:

Definition 2.4.1 (Atomic formulas of b-DL)**.** In all versions of DL (including b-DL), an atomic formula is the atomic formula of the first-order vocabulary Θ, and is defined as:

$$r(t_1,\ldots,t_n)$$

where r is an *n-ary* relation symbol of Σ and $t_1,\ldots,t_n$ are terms of Θ.

Definition 2.4.2 (Atomic programs of b-DL)**.** In b-DL, an atomic program is a simple *assignment* and is defined as:

$$x := t$$

where $x \in V$and t is a term of Θ. This program assigns the value of t to the variable x.

The set of all atomic formulas is denoted by Φ_0, and the set of all atomic programs is denoted by Π_0. b-DL formulas and programs are built inductively from the atomic ones using the following operators:

- Propositional operators:
  $\rightarrow$ implication
  0 falsity
- Program operators:
  ; composition
  $\cup$ choice
  * iteration
- Mixed operators:
  [ ] necessity
  ? test

Based on the definitions of atomic formulas and programs of b-DL, the b-DL formulas Φ and programs Π are defined as follows:

Definition 2.4.3 (b-DL formulas and programs)**.** The set Φ of b-DL formulas and the set Π of b-DL programs are defined to be the smallest sets such that:

- $\Phi_0 \subseteq \Phi$;
- $\mathbf{0} \in \Phi$;
- If $\varphi, \psi \in \Phi$, then $\varphi \rightarrow \psi \in \Phi$
- If $\varphi \in \Phi$ and $x \in V$, then $\forall x \varphi \in \Phi$;
- If $\alpha \in \Pi$ and $\varphi \in \Phi$, then $[\alpha]\varphi \in \Phi$;
- $\Pi_0 \subseteq \Pi$;
- If $\varphi \in \Phi$ then $\varphi? \in \Pi$;
- If $\alpha, \beta \in \Pi$, then $\alpha; \beta \in \Pi$;
- If $\alpha, \beta \in \Pi$, then $\alpha \cup \beta \in \Pi$;
- If $\alpha \in \Pi$, then$\alpha^* \in \Pi$.

The possibility operator < > is the modal dual of the necessity operator [ ]:

$$<\alpha>\varphi \overset{\text{def}}{=} \neg[\alpha]\neg\varphi.$$

2.4.2 Semantical domain and semantics of b-DL

b-DL formulas and programs are interpreted over a first-order structure $\mathfrak{A}$ for the vocabulary Θ, where:

$$\mathfrak{A} = (A, \mathfrak{m}_{\mathfrak{A}}),$$

$\mathfrak{A}$ is called the *domain of computation*; A is a set that is called the *carrier* of $\mathfrak{A}$; $\mathfrak{m}_{\mathfrak{A}}$ is a *meaning function* such that $\mathfrak{m}_{\mathfrak{A}}(f) : A^n \rightarrow A$ is an n-ary function that interprets the n-ary function symbol f of Θ, and $\mathfrak{m}_{\mathfrak{A}}(r)$ is an n-ary relation such that $\mathfrak{m}_{\mathfrak{A}}(r) \subseteq A^n$, which interprets the n-ary relation symbol r of Θ. The equality symbol = is always interpreted as the identity relation. For $n \geq 0$, let $A^n \ \rightarrow \ A$ denote the set of all n-ary functions in A. By convention. we take $A^0 \ \rightarrow \ A = A$. Let A^* denote the set of all finite-length strings over A.

An instantaneous snapshot of all relevant information at any moment during the computation is determined by the values of the program variables. Thus *states* can be denoted by *valuations* $u, v, \ldots$ of the set of variables V over the carrier of the structure $\mathfrak{m}_{\mathfrak{A}}$. The valuations over $\mathfrak{m}_{\mathfrak{A}}$ are from [38] and are shown as follows:

Definition 2.4.4 (Valuations over structure $\mathfrak{m}_{\mathfrak{A}}$)**.** A *valuation* over $\mathfrak{m}_{\mathfrak{A}}$ is a function u assigning an n-ary function over A to each n-ary array variable, where:

- $u(x) \in A$ if $x \in V$;
- For an n-ary function symbol f and terms $t_1, \ldots, t_n$ of Θ,

$$u(f(t_1, \ldots, t_n)) \stackrel{\text{def}}{=} \mathfrak{m}_{\mathfrak{A}}(f)(u(t_1), \ldots, u(t_n));$$

- For an n-ary array variable F and terms $t_1, \ldots, t_n$ of Θ,

$$u(F(t_1, \ldots, t_n)) \stackrel{\text{def}}{=} u(F)(u(t_1), \ldots, u(t_n)).$$

A *function-patching operator* is also needed for interpreting b-DL formulas and programs. The definition of function-patching operator is from [38] and is shown as follows:

Definition 2.4.5 (function-patching operator)**.** If X and D are sets, $f : X \rightarrow D$ is any function, $x \in X$ and $d \in D$, then $f[x/d] : X \rightarrow D$ is the function defined by

$$f[x/d](y) \stackrel{\text{def}}{=} \begin{cases} d, & \text{if } x = y \\ f(y), & \text{otherwise.} \end{cases}$$

The set of states of $\mathfrak{m}_{\mathfrak{A}}$ is denoted by $S^{\mathfrak{A}}$. Thus every b-DL formula $\varphi$is associated with a set

$$\mathfrak{m}_{\mathfrak{A}}(\varphi) \subseteq S^{\mathfrak{A}};$$

and every b-DL program α is associated with a binary relation

$$\mathfrak{m}_{\mathfrak{A}}(\alpha) \subseteq S^{\mathfrak{A}} \times S^{\mathfrak{A}}.$$

The semantics of b-DL formulas and programs are defined based on the definitions above. The definition of the semantics is from [38] and is shown as follows:

Definition 2.4.6 (Semantics of b-DL formulas and programs)**.** The semantics of b-DL formulas and programs are defined over the structure $\mathfrak{A}$ of Θ with $t, t_1, \ldots, t_n$ being terms of Θ, $F(t_1, \ldots, t_n)$ being an n-ary array variable, r being a relation symbol of Θ and $x \in V$:

- $\mathfrak{m}_{\mathfrak{A}}(r(t_1, \ldots, t_n)) \overset{\text{def}}{=} \{u |$ if $u \in \mathfrak{m}_{\mathfrak{A}}(r(t_1, \ldots, t_n))\}$;
- $\mathfrak{m}_{\mathfrak{A}}(\mathbf{0}) \overset{\text{def}}{=} \emptyset$;
- $\mathfrak{m}_{\mathfrak{A}}(\varphi \to \psi) \overset{\text{def}}{=} \{u |$ if $u \in \mathfrak{m}_{\mathfrak{A}}(\varphi)$ then $u \in \mathfrak{m}_{\mathfrak{A}}(\psi)\}$;
- $\mathfrak{m}_{\mathfrak{A}}(\forall x\, \varphi) \overset{\text{def}}{=} \{u | \forall a \in A\ u[x/a] \in \mathfrak{m}_{\mathfrak{A}}(\varphi)\}$;
- $\mathfrak{m}_{\mathfrak{A}}(x := t) \overset{\text{def}}{=} \{(u, u[x/u(t)]) | u \in S^{\mathfrak{A}}\}$;
- $\mathfrak{m}_{\mathfrak{A}}(F(t_1, \ldots, t_n) := t) \overset{\text{def}}{=} \{(u, u[F/u(F)[u(t_1), \ldots, u(t_n)/u(t)]]) | u \in S^{\mathfrak{A}}\}$;
- $\mathfrak{m}_{\mathfrak{A}}([\alpha]\varphi) \overset{\text{def}}{=} \{u | \forall v$ if $(u, v) \in \mathfrak{m}_{\mathfrak{A}}(\alpha)$ then $v \in \mathfrak{m}_{\mathfrak{A}}(\varphi)\}$;
- $\mathfrak{m}_{\mathfrak{A}}(\alpha ; \beta) \overset{\text{def}}{=} \{(u, v) | \exists w (u, w) \in \mathfrak{m}_{\mathfrak{A}}(\alpha)$ and $(w, v) \in \mathfrak{m}_{\mathfrak{A}}(\beta)\}$;
- $\mathfrak{m}_{\mathfrak{A}}(\alpha \cup \beta) \overset{\text{def}}{=} \mathfrak{m}_{\mathfrak{A}}(\alpha) \cup \mathfrak{m}_{\mathfrak{A}}(\beta)$;
- $\mathfrak{m}_{\mathfrak{A}}(\alpha^*) \overset{\text{def}}{=} \mathfrak{m}_{\mathfrak{A}}(\alpha)^* = \bigcup_{n \geq 0} \mathfrak{m}_{\mathfrak{A}}(\alpha)^n$;
- $\mathfrak{m}_{\mathfrak{A}}(\varphi?) \overset{\text{def}}{=} \{(u, u) | u \in \mathfrak{m}_{\mathfrak{A}}(\varphi)\}$.

Chapter 3

An overview of SRML

In this chapter, we give an overview of SRML [17] in two aspects:

1. The basic compositions of SRML service assembly module;

2. The semantics domain of SRML.

The first aspect includes the introduction to the compositions of SRML (such as business role, business protocol and etc.) and in general how these compositions are defined. The second aspect includes the semantic domain for interactions such as configurations and interaction signatures, and the semantic domain for transitions such as Service-Oriented doubly labeling transition systems.

3.1 Introduction to SRML service module

SRML is a language for modeling composite services, whose business logic involves a number of interactions among more elementary components that composite a service, the invocations from external clients to this service, and the invocations of services provided by external services. In SRML such a service is specified by a service module. A *service module* defines how the composition of a service can be specified using a graph that is labeled by the SRML specifications of service components, service interfaces and internal wires. In general it defines a wired interconnection of service components and service interfaces. It also defines which of these interfaces connect to clients of this service, and to services invoked by this service directly. A service module provides a specification for each service component (specified with a business role), service interface (specified with a business protocol) and internal wires (specified with an interaction protocol). In this thesis, we don't consider external wires. The service interfaces in every service module are distinguished as a provides-interface and several requires-interfaces. A provides-interface exposes the service to outside parties (known as service requirers or clients) that need to call this service module; a requires-interface call an outside party (known as service providers) to use the service provided by that party. Figure 3.1 shows the composition of a SRML service module:

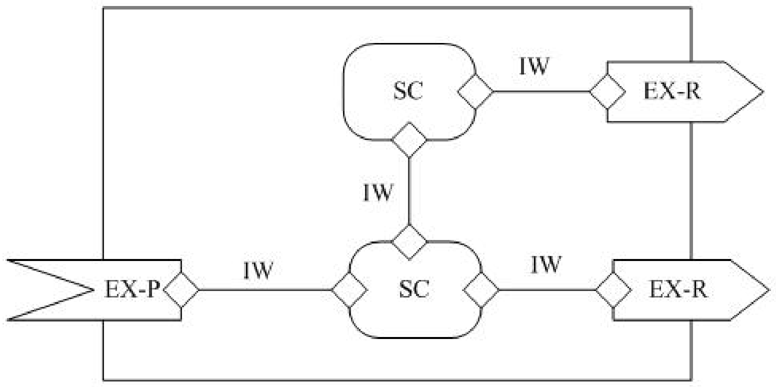


Figure 3.1: A SRML service module where every SC is a service component, every EX-P is a provides-interface, every EX-R is a requires-interface and every IW is an internal wire.

The central compositions of a service module are service components. A service component is a computational unit that in SRML is modeled with a business role, which consists of a set of interactions and an orchestration. The interactions model the communications between the components which contain these interactions and other service components or interfaces in the same service module. The orchestration includes the declaration of variables local to this component, and a set of transitions which model the activities performed by the component. These transitions are independent of the language in which the component is programmed and the platform in which it is deployed. For example, they can be implemented with a BPEL process, a Java program or a wrapped-up legacy system. In addition, an orchestration is independent of the clients or other services to which this module are connected at run-time. In general, a service component is totally independent in the sense that it does not invoke any specific service from the outside nor provide service to outside clients, it just perform computations by involving in a set of interactions.

Service interfaces do not provide any business logic. They only specify the way that a certain service component interacts with an outside party (a client or an other service) according to a given business protocol. A service interface in SRML is modeled with a business protocol, which consists of a set of interactions in which the service component and the outside party engage and a set of behaviors that the outside party can expect this service module to perform.

Service components and service interfaces within a service module are connected to each other through internal wires. These wires specify bindings of the interactions that are declared in both sides, and coordinate them according to a given interaction protocol. A wire in SRML is modeled with a interaction protocol and a connector.

As mentioned above, the different types of entities involved in a service module – service components, service interfaces, and wires – are specified in SRML using three different but related languages: business role, business protocol and interaction protocol which are introduced as follows:

- *Business roles* are instantiated with service components. A business role is a modeling language that consists of a set of interactions and an orchestration, which specifies the way these interactions are orchestrated. The orchestration is specified by declaring a set of variables which provides an abstract view of the current state of the component, and a set of transitions that model the activities performed by the component. A transition may have a trigger, which is an event or a state condition specifying the condition of the occurrence of the transition; a guard, which is a condition that identifies the state in which the transition can take place; and a set of effects which are propositions that are true in the local state and specify the effects of the transition in that state.

- *Business protocols* are instantiated with service interfaces. Like business roles, business protocols declare the interactions in which the external parties (services and clients) can be involved. The difference is that, instead of an orchestration, a business protocol provides a set of properties that the external parties can expect the service module in which this business protocol is declared to adhere to.

- *Interaction protocols* are instantiated with wires. In a service module, a number of service components and service interfaces are connected to one another by wires. Interaction protocols are labeled with connectors that coordinate the interactions in which the service components or service interfaces linked by the wire are involved.

Figure 3.2 shows an example of a service module *Train-Control* from Appendix A. *Train-Control* models a small part of the *European train Control System*. It includes a service component *ETCSC*, which is modeled by business role *ETCSCenter*; a service interface *MOC*, which is modeled by business protocol *MonitoringCenter*; a service interface *RBC*, which is modeled by business protocol *RadioBlockCenter*; a wire ⟨*MOC,ETCSC*⟩, which is modeled by connector *ME*; and a wire ⟨*ETCSC,RBC*⟩, which is modeled by connector *ER*.

In SRML, each interaction involves two parties (each party can be either a service component or a service interface), and can be in both directions. An interaction is described from the point of view of the party in which it is declared. If an interaction is declared in one party, the other party involved in this interaction is called a *co-party* of this party. As that is defined in [17], the types of interactions are distinguished as follows:

- **r&s**: The interaction is initiated by the co-party, which expects a reply. The co-party does not block while waiting for the reply;

- **s&r**: The interaction is initiated by the party and expects a reply from its co-party. While waiting for the reply, the party does not block;

- **rcv**: The co-party initiates the interaction and does not expect a reply;

- **snd**: The party initiates the interaction and does not expect a reply;

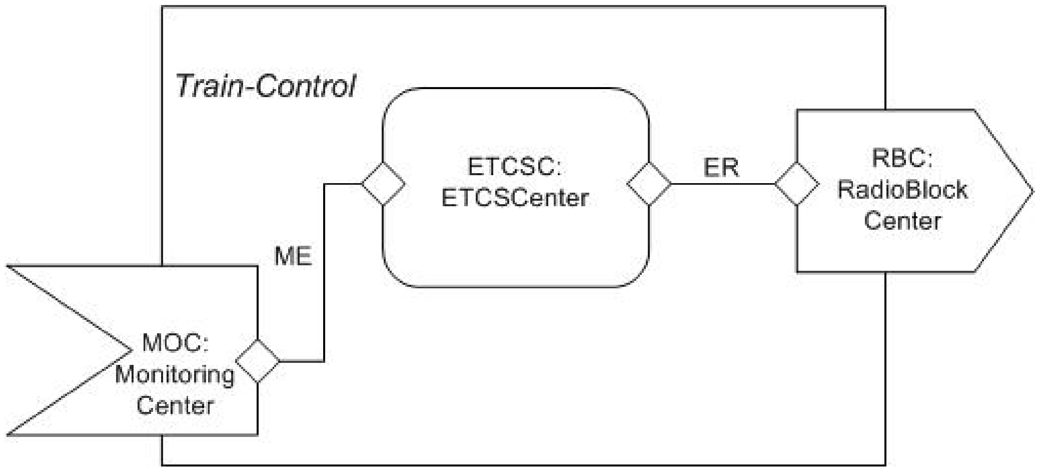


Figure 3.2: SRML service module *Train-Control*

- **ask**: The party synchronies with the co-party to obtain data;
- **rpl**: The party synchronies with the co-party to transmit data;
- **tll**: The party requests the co-party to perform an operation and blocks;
- **prf**: The party performs an operation and frees the co-party that requested it.

Among the interaction types above, **r&s** and **s&r** are two-way interactions, which can take place in two directions, and the others are one-way interactions, which can take place only in one direction.

In SRML, an interaction is associated with one or several events, which occur during state transitions in both parties involved in the interaction. For a given interaction named *a*, the events that associate with this interaction are distinguished as follows:

- *a*🔔: The event of initiating *a*;
- *a*⊠: The reply-event of *a*;
- *a*✓: The commit-event of *a*;
- *a*✗: The cancel-event of *a*;
- *a*✞: The revoke-event of *a*.

An event can be associated with one or several parameters. For example, if an event *a*🔔has an associating parameter *cost*, the parameter in SRML is denoted by *a*🔔*.cost*. Especially, every reply-event *a*⊠ has a distinguished Boolean parameter *Reply* that indicates whether the reply is positive or not.

3.2 Semantic domain of SRML

The semantic domain of SRML is the domain over which SRML is interpreted. It consists of an abstract data signature, a service-oriented configurations and a service-oriented doubly labeling transition system.

SRML focuses on the patterns of message exchange in the service-oriented frame work. In order to model this, we first abstract the data and operation domain with a fixed data signature

$$\Omega = \langle D, F \rangle$$

where D is a set of data types (such as *int*, *double* and so on) and F is a $D^* \times D$-indexed family of sets of operations over the types (such as addition, conjunction, derivative and so on). The definition of Ω is adapted from [17]. In this thesis we assume that *time*, *boolean* $\in D$ are data types that represent the usual concepts of time and true/false values, and the derivative operator $\cdot$ is in F. We further assume that a fixed algebra $\mathscr{U}$ interprets Ω.

Based on the abstract data signature Ω, service-oriented configurations and service-oriented doubly labeling transition systems can be defined. They are the compositions of the semantic domain of SRML. A service-oriented configuration is an extension of configuration [17] which defines a simple graph with nodes and wires. A service-oriented doubly labeling transition system is a combination of service-oriented transition system [17] and doubly labeling transition system [36]. We introduce them separately in the following sub-sections.

3.2.1 Service-Oriented Configurations

The semantic domain of interactions in SRML is defined by Service-Oriented Configurations. Service-Oriented Configurations are extensions of configurations, which is defined in [17].

A *service-oriented configuration (SO-configuration)* is a graph in which each node represents a party (a service component or a service interface) that is capable of performing computations in a service module, and each edge represents a wire that connects the parties. The graph is a simple graph, which means it has no multiple edges and no loops, and is undirected. Interactions performed by each party are also contained in the SO-configuration, and interactions that can take place between each pair of the parties are fixed. Each interaction has a direction, denoting that the message is sent by which party and is received by which party. In SO-configurations, interactions are classified into one-way interactions (interactions which have only one direction) and two-way interactions (interactions which have two directions). Associated with each wire, there is a time delay, which denote the delay of time when messages are propagated asynchronously.

The following definition of SO-configuration is a combination of Definition 3.2.1 and 3.2.2 from [17]:

Definition 3.2.1 (Service-Oriented Configuration)**.** A Service-Oriented Configuration *is a tuple*

$$\langle N, WIRE, PLL, \Psi, 2WAY, 1WAY \rangle$$

where

- $\langle N, WIRE\rangle$ *is a simple graph, in which N denotes the nodes and WIRE denotes the wires connecting the nodes.*
- $PLL \subseteq N$ *are distributed nodes.*
- Ψ *assigns to every* $w \in WIRE$ *an element* $w.delay^{\Psi} \in time_{\mathscr{U}}$ *that denotes the delay associated with wire w.*
- $2WAY$ *and* $1WAY$ are $N \times N$*-indexed families of mutually disjoint sets of "two-way" and "one-way" interactions, respectively; we use INT to refer to* $2WAY \cup 1WAY$*;*
- *For every* $n, n' \in N$ *if* $\langle n, n'\rangle \notin WIRE$ *then* $INT_{\langle n,n'\rangle} = \emptyset$*, i.e. there is no interaction between nodes that is connected by a wire.*

In every two-way interaction, a party sends a request to its co-party and receives either a positive or a negative reply from the co-party. There is always a pledge associates with the positive reply, which is a time duration within which the next message is guaranteed to be deliver by that party.

Based on the definition of SO-configurations, the events that associate with interactions in SRML can be defined. Each event has the same direction as the interaction with which it associates. If interaction $a \in 1WAY_{\langle n,n'\rangle}$, which means the direction of a is from party n to party *n'*, it is obvious that only the initiation-event is sent from n to *n'*. If interaction $a \in 2WAY_{\langle n,n'\rangle}$, which means the directions of a are both from party n to party *n'* and from party *n'* to party n, the initiation-event, the commit-event, the cancel-event and the revoke-event can be sent from n to n', and the reply-event is sent from n' to n. In short, among the five types of events introduced in section 3.1, if interaction a is a one-way interaction, there is only one event, $a\text{🔔}$, associates with it; if interaction a is a two-way interaction, the set of events associates with it is a subset of $\{a\text{🔔}, a\text{✉}, a\text{✓}, a\text{✗}, a\text{✞}\}$.

The formal definition of the events that associate with interactions is the same as Definition 3.3.1 from [17], and is shown as follows:

Definition 3.2.2 (Relating interactions with events)**.** *For every interaction* $a \in INT$ *and node* $n \in N$*, the function* E_n *that maps from the interaction a to events associated with a that can be received by* n' *is defined as follows:*

$$\begin{aligned}
&\textit{If } a \in 2WAY_{\langle n,n'\rangle} \textit{ then}\\
&\quad E_n(a) = \{a\text{🔔}, a\text{✓}, a\text{✗}, a\text{✞}\}\\
&\quad E_{n'}(a) = \{a\text{✉}\}\\
&\quad E_{n''}(a) = \emptyset \textit{ for any other } n'' \in N\\
&\textit{If } a \in 1WAY_{\langle n,n'\rangle} \textit{ then}\\
&\quad E_n(a) = \{a\text{🔔}\}\\
&\quad E_{n'}(a) = \emptyset\\
&\quad E_{n''}(a) = \emptyset \textit{ for any other } n'' \in N
\end{aligned}$$

We also define the following sets:

- $E_{\langle n\rangle} = \bigcup\{E_n(a) : a \in INT_{\langle n,n'\rangle}\}$ *is the set of all events that can be received by node n;*
- $E_a = E_n(a) \cup E_{n'}(a)$ where *is the set of events associated with interaction a;*
- $E_{\langle n,n'\rangle} = \bigcup\{E(a) : a \in_{\langle n,n'\rangle} \vee a \in_{\langle n',n\rangle}\}$ *is the set of all events that are carried by wire* $\langle n,n'\rangle$*;*
- $E = \bigcup\{E(a) : a \in INT\}$ *is the set of all events that can occur.*

Based on the definition of SO-configurations and the relationships between interactions and events, parameters can be defined. In the definition of parameters, to avoid ambiguity, all the parameters associating with the same interaction are assumed to have different names. In addition, every reply-event that associates with a two-way interaction has at least two parameters: *Reply* and *useBy*. *Reply* is a parameter of type boolean, which indicates if the reply is positive or negative. *useBy* is a parameter of type time, which stores the expiration time until when the pledge holds.

The formal definition of semantic domain of parameters is the same as Definition 3.3.2 from [17] and is shown as follows:

Definition 3.2.3 (Parameters)**.** *A parameter-assigning function PP assigns a D-indexed family of disjoint sets of parameters to each event in E such that:*

- *For every* $a \in INT, e, e' \in E(a)$, $P \in PP(e)$ *and* $P' \in PP(e')$, P *and* P' *are disjoint.*
- *For every* $a \in 2WAY, Reply \in PP(a\boxtimes)_{boolean}$ *and* $useBy \in PP(a\boxtimes)_{time}$.

3.2.2 Service-Oriented Doubly Labeled Transition System

Except interactions, other parts of SRML are interpreted over *Service-Oriented Doubly Labeled Transition systems (SO-L^{2}TSs)*. SO-L^{2}TSs is a discrete model of computation for SO-configurations, and it is a combination of *Service-Oriented Transition System (SO-TSs)* [17] and *Doubly Labeled Transition Systems (L^{2}TSs)* [37]. In this section, we first introduce *computation state* [17] and *computation step* [17], on the basis of which SO-TSs are defined.

When a configuration computes, it goes through a sequence of states. These states are characterized by events that are pending in wires and are buffered in the nodes of the configuration, and also by pledges that hold in that state and the history and parameters of events. The formal definition of computation state is the same as Definition 3.4.1 from [17] and is shown as follows:

Definition 3.2.4 (Computation state)**.** *A computation state is a tuple*

$$\langle PND, INV, TIME, PLG, HST, \Pi\rangle$$

where:

- $PND \subseteq E$ *is the set of events pending in that state, i.e. the events that are waiting to be delivered by the corresponding wire;*

- *$INV \subseteq E$ is the set of events invoked in that state, i.e. the events that have been delivered and are waiting to be processed;*
- *$TIME \in time_{\mathscr{U}}$ is the time in that state;*
- *$PLG \subseteq \{a.pledge : a \in 2WAY\}$is the set of pledges that hold in that state;*
- *HST consists of four subsets of E, HST!,HST¡,HST? and HST¿ that keep the history of event propagation; they contain the events that have been published, delivered, executed and discarded, respectively;*
- Π *assigns to each parameter* $p \in PP(e)_d$ of datatype $d \in D$, *with* $e \in E(a)$ *and* $a \in INT$, *a value* $a.p^{\Pi} \in d_{\mathscr{U}}$, *i.e.* Π *keeps the value of each parameter.*

If $s = \langle PND, INV, TIME, PLG, HST, \Pi \rangle$ is a computation state we use PND^s, INV^s, $TIME^s$, PLG^s, HST^s and Π^s to refer to the components in that state.

Computation states evolve through computation steps. Given a sequence of computation states, a computation step specifies the evolution from one state (called a source state) to its successor (called a target state), and time moves forward from a source state to its target state. In each computation step, events can be published, delivered or processed. Among the events that are processed, some are executed and the others are simply discarded. Nodes that do not perform parallel computation can only process one event during each computation step. The formal definition of computation step is the same as Definition 3.4.2 from [17] and is shown as follows:

Definition 3.2.5 (Computation step)**.** A computation step *is a tuple*

$$\langle SRC, TRG, DLV, PRC, EXC, \Theta \rangle$$

where:

- *SRC and TRG are the source and target states;*
- *$DLV \subseteq PND^{SRC}$ is the set of events that are selected for delivery during that step;*
- *PRC is a partial function that selects for each node n such that INV_n^{SRC} is non-empty, a subset of INV_n^{SRC}, such that if $|PRC(n)| > 1$ then $n \in PLL$, i.e. PRC selects which events will be processed during that step such that only one event can be processed by each sequential node;*
- *$EXC \subseteq PRC$ is the set of events that are executed during that step; $DSC = PRC \backslash EXC$ is the set of events that are discarded, i.e. the events that are processed but are not executed;*
- *$PND^{TRG} = (PND^{SRC} \backslash DLV) \uplus PUB$ where $PUB \subseteq E$, i.e. the events that were selected for delivery will no longer be pending in the target state; the new events that become pending in the target state are those that are published during the computation step;*

- *There is a set of actually-delivered events $ADLV \subseteq DLV$ such that for every $n \in N$;*
  - *If PRC(n) is defined then $INV_n^{TRG} = (INV_n^{SRC} \backslash \{PRC(n)\}) \cup ADLV_n$*
  - *If PRC(n) is undefined then $INV_n^{TRG} = INV_n^{SRC} \cup ADLV_n$*

  i.e. the events that were processed will no longer be waiting in the target state; the events that are actually delivered to a component will have to wait until they are processed;

- *Θ assigns to each parameter in $PP(e)_d$ such that $e \in PUB$ and $d \in D$ (is the datatype of the parameter), an element in $d_{\mathscr{U}}$, i.e. the value of the parameter;*

- *$TIME^{SRC} < TIME^{TRG}$, i.e. time moves forward;*

- *SRC and TRG are such that:*
  - $HST!^{TRG} = HST!^{SRC} \cup PUB$
  - $HST¡^{TRG} = HST¡^{SRC} \cup ADLV$
  - $HST?^{TRG} = HST?^{SRC} \cup EXC$
  - $HST¿^{TRG} = HST¿^{SRC} \cup DSC$
  - $\Pi(e)^{TRG} = \Theta(e)$ for each $e \in PUB$
  - $\Pi(e)^{TRG} = \Pi(e)$ for each $e \notin PUB$

A SO-TS defines the different possibilities that its configuration can have when computing. In a SO-TS, every state is labeled with a computation state, and every transition is labeled with a computation step. A path of a SO-TS consists of a sequence of computation states and the computation steps between each two adjacent states, and represents a possible computation that the SO-TS can perform. The formal definition of SO-TS is the same as Definition 3.4.4 from [17] and is shown as follows:

Definition 3.2.6 (Service-Oriented Transition System)**.** A Service-Oriented Transition System (SO-TS) *is a tuple*

$$\langle S, \rightarrow, s_0, G \rangle$$

where:

- *$\langle S, \rightarrow \rangle$ is a directed acyclic graph, where S is the set of vertices (the transitions between states);*

- *$s_0 \in S$ is the initial state;*

- *G is a labeling function that assigns a computation state to every state $s \in S$ and a computation step to every transition $s \rightarrow s'$, such that $G(s \rightarrow s') = \langle G(s), G(s'), \_, \_, \_, \_ \rangle$, i.e. the source and target computation states associated with a transition are the ones that label the states of that transition.*

We use the following notation to refer to the components of the labels:

- *If s is a state such that* $(s \in S)$ *we use* $PND^s, INV^s, TIME^s, PLG^s, HST^s$ *and* Π^s *to refer to the components of computation state G(s).*

- *If r is a transition* $(r \in R)$ *we use* $SRC^r, TRG^r, DLV^r, PRC^r, EXC^r, DSC^r$ *and* PUB^r *to refer to the components of the computation step* $G(r)$*.*

- *If s and s' are states such that* $(s, s' \in S)$*, s and s' are similar up to time, denoted by* $s \sim s'$*, iff* $G(s)$ *and* $G(s')$ *are the same except for the TIME components, i.e,* $PND^s = PND^{s'}, INV^s = INV^{s'}, PLG^s = PLG^{s'}, HST^s = HST^{s'}$*.*

Note that two states *s* and *s'* of a SO-TS are different, if any of the components of their computation states *G(s)* and *G(s')* are different, e.g. if *s* and *s'* occur at different time instants then they are different.

Doubly Labeled Transition Systems (L^{2}TSs) [37] extend *Labeled Transition Systems (LTSs)* [39]. The same as in LTSs, in L^{2}TSs a transition from one state to another is labeled with input that are expected, conditions that must be true to trigger the transition, or events that are performed during the transition. The difference is that: in L^{2}TSs transitions can be labeled by sets of events rather than single events, and states are labeled with atomic propositions. The formal definition of Doubly Labeled Transition System is the same as Definition 4.1.1 from [17], and is shown as follows:

Definition 3.2.7 (Doubly Labeled Transition System)**.** A Doubly Labeled Transition System *(L^{2}TS) is a tuple*

$$\langle S, s_0, Act, R, AP, L \rangle$$

where:

- *S is a set of states;*

- $s_0 \in S$ *is the initial state;*

- *Act is a finite set of observable actions;*

- $R \subseteq S \times 2^{Act} \times S$ *is the transition relation. A transition* $s \xrightarrow{\alpha} s'$ *is denoted by* $(s, \alpha, s') \in R$;

- *AP is a set of atomic propositions;*

- $L : S \to 2^{AP}$ *is a labeling function such that* $L(s)$ *is the subset of all atomic propositions that are true in state s.*

A *Service-Oriented Doubly Labeled Transition System (SO-L^{2}TS)* is a L^{2}TS-based refinement of a SO-TS. The actions labeling transitions in a SO-L^{2}TS denote different stages of event propagation: publication, delivery, execution and discard. In addition, every state in a SO-L^{2}TS is also labeled with information about the actions that have happened before that state is reached. The formal definition of SO-L^{2}TS is the same as Definition 4.2.1 from [17] and is shown as follows:

Definition 3.2.8 (Service-Oriented L^2TS)**.** *A* Service-Oriented L^{2}TS (SO-L^{2}TS) *is a tuple*

$$\langle S, s_0, Act, R, AP, L, TIME, \Pi \rangle$$

where:

- *The SO-L^{2}TS refines a SO-TS $\langle S, \rightarrow, s_0, G \rangle$;*
- $Act = \{e! : e \in E\} \cup \{e¡ : e \in E\} \cup \{e? : e \in E\} \cup \{e¿ : e \in E\}$;
- *$R \subseteq S \times 2^{Act} \times S$ is such that:*
  - *$s \rightarrow s'$* iff *$(s, \alpha, s') \in R$ for some $\alpha \in 2^{Act}$;*
  - *For every $(s, \alpha, s') \in R$:*

$$\begin{aligned}\alpha = &\{e! : e \in PUB^{s \rightarrow s'}\} \cup \{e¡ : e \in ADLV^{s \rightarrow s'}\} \cup \\ &\{e? : e \in EXC^{s \rightarrow s'}\} \cup \{e¿ : e \in DSC^{s \rightarrow s'}\}\end{aligned}$$

- *$AP = \{e! : e \in E\} \cup \{e¡ : e \in E\} \cup \{e? : e \in E\} \cup \{e¿ : e \in E\} \cup \{a.pledge : a \in 2WAY\}$;*
- *$L : S \rightarrow 2^{AP}$ is such that:*

$$\begin{aligned}L(s) = &\{e! : e \in HST!^{s}\} \cup \{e¡ : e \in HST¡^{s}\} \cup \\ &\{e? : e \in HST?^{s}\} \cup \{e¿ : e \in HST¿^{s}\} \\ &\cup PLG^{s}\end{aligned}$$

- *$TIME : S \rightarrow \{r \in \mathbb{R} | r \geq 0\}$ is a function that assigns to each state $s \in S$ the instant $TIME^s$;*
- Π *assigns to each state $s \in S$ the parameter interpretation* Π^s.

Chapter 4

Hybrid extension of SRML and its semantic domain

SRML is an orchestration-oriented modeling language that formally specifies the discrete behaviors of services. In order to be able to also formally specify the hybrid behaviors (behaviors that include both time-continuous processes and discrete events) of services, in this chapter we provide a hybrid extension of SRML and an extension of SRML semantic domain over which the hybrid extension of SRML can be interpreted. More specifically, we define the "Service-Oriented Hybrid Doubly Labeled Transition System" as the semantic domain of hybrid extension of SRML, make hybrid extension for each composition of SRML service module and finally define the whole service module over the definitions of theses compositions. This Chapter is arranged as follows:

- In Section 4.1 we extend the semantic domain of SRML by defining the Service-Oriented Hybrid Doubly Labeled Transition System and its paths over which the hybrid extension of SRML is interpreted;

- In Section 4.2 we review the definitions on interaction signature and interaction interpretation which are defined in [17]. Interactions declared in business roles and business protocols are interpreted over interaction signatures;

- In Section 4.3 we define the syntax and semantics of the hybrid extension of business roles, which are orchestration-oriented languages for specifying the computations performed by the internal components of service modules;

- In Section 4.4 we define the syntax and semantics of the hybrid extension of business protocols, which are temporal dynamic logic (dTL) based languages for specifying service interfaces;

- In Section 4.5 we define the syntax and semantics of the hybrid extension of interaction protocols, which are interaction based languages for specifying internal wires between components in service modules;

- In Section 4.6 we formalize service module by defining it over the hybrid extensions of business roles, business protocols and interaction protocols.

The main contents of this chapter are published in [28], but are more sound than that in [28].

4.1 The extension of SRML semantic domain

In order to interpret hybrid systems that are specified by SRML, we first define the *Hybrid Doubly Labeled Transition System (HL^{2}TS)*, which is an extension of the L^{2}TS. An HL^{2}TS extends the L^{2}TS in that it defines a set of functions Σ. These functions map from the real number domain to the state domain of the HL^{2}TS and can be used to interpret the evolutions of hybrid systems specified by certain modeling languages. HL^{2}TS is defined as follows:

Definition 4.1.1 (Hybrid Doubly Labeled Transition System)**.** A Hybrid Doubly Labeled Transition System *(HL^{2}TS) is a tuple*

$$\langle S, s_0, \Sigma, Act, R, AP, L\rangle$$

where:

- *S is a set of states;*
- *$s_0 \in S$ is the initial state;*
- *Σ is a set of functions and for every function $\sigma \in \Sigma$ there is $\sigma : [0, r_\sigma] \to S$ with $r_\sigma \in \mathbb{R}$ and $r_\sigma \geq 0$;*
- *Act is a finite set of observable actions;*
- *$R \subseteq \{\sigma(r_\sigma) : \sigma \in \Sigma\} \times 2^{Act} \times \{\sigma(0) : \sigma \in \Sigma\}$ is the transition relation. A transition $\sigma(r_\sigma) \xrightarrow{\alpha} \sigma'(0)$ is denoted by $(\sigma(r_\sigma), \alpha, \sigma'(0)) \in R$* where *$\alpha \subset Act$;*
- *AP is a set of atomic propositions;*
- *$L : S \to 2^{AP}$ is a labeling function such that $L(s)$ is the subset of all atomic propositions that are true in state s.*

In Definition 4.1.1, the set of state S can be finite or infinite; for every $\sigma \in \Sigma$, the real number r_σ is unique, and σ on the interval $[0, r_\sigma]$ represents the prolongation of states in the duration whose length is r_σ; function L is a state labeling function that defines which propositions are true in each state, and these propositions are from the fixed set of atomic propositions AP.

As introduced in Chapter 3, the models of computation that are used for configurations are the transition system labeled with sets of events — what is defined as SO-TSs. In order to add the service-oriented feature to HL^{2}TSs, we need to define a HL^{2}TS-based refinement of a SO-TS, which is named as *Service-Oriented HL^{2}TS (SO-HL^2TS)* and can be also seen as an extension of the HL^{2}TS. The SO-HL^2TS is defined as follows:

Definition 4.1.2 (Service-Oriented Hybrid L^2TS)**.** *A* Service-Oriented Hybrid L^{2}TS(SO-HL^{2}TS) *that refines a SO-TS $\langle S, \rightarrow, s_0, G\rangle$ is a tuple*

$$\langle S, s_0, \Sigma, Act, R, AP, L, TIME, \Pi\rangle$$

where:

- $Act = \{e! : e \in E\} \cup \{e¡ : e \in E\} \cup \{e? : e \in E\} \cup \{e¿ : e \in E\}$;
- $R \subseteq \{\sigma(r_\sigma) : \sigma \in \Sigma\} \times 2^{Act} \times \{\sigma(0) : \sigma \in \Sigma\}$ *is such that:*
  - $\sigma(r_\sigma) \rightarrow \sigma'(0)$ iff $(\sigma(r_\sigma), \alpha, \sigma'(0)) \in R$ *for some* $\alpha \in 2^{Act}$*;*
  - *For every* $(\sigma(r_\sigma), \alpha, \sigma'(0)) \in R$*:*

$$\begin{aligned}\alpha =& \{e! : e \in PUB^{\sigma(r_\sigma)\rightarrow\sigma'(0)}\} \cup \{e¡ : e \in ADLV^{\sigma(r_\sigma)\rightarrow\sigma'(0)}\} \cup \\ & \{e? : e \in EXC^{\sigma(r_\sigma)\rightarrow\sigma'(0)}\} \cup \{e¿ : e \in DSC^{\sigma(r_\sigma)\rightarrow\sigma'(0)}\}\end{aligned}$$

- $AP = \{e! : e \in E\} \cup \{e¡ : e \in E\} \cup \{e? : e \in E\} \cup$ $\{e¿ : e \in E\} \cup \{a.pledge : a \in 2WAY\}$*;*
- For every $\sigma \in \Sigma$ and every $\zeta \in [0, r_\sigma]$, $L : S \rightarrow 2^{AP}$ *is such that:*

$$\begin{aligned}L(\sigma(\zeta)) =& \{e! : e \in HST!^{\sigma(\zeta)}\} \cup \{e¡ : e \in HST¡^{\sigma(\zeta)}\} \cup \\ & \{e? : e \in HST?^{\sigma(\zeta)}\} \cup \{e¿ : e \in HST¿^{\sigma(\zeta)}\} \\ & \cup PLG^{\sigma(\zeta)}\end{aligned}$$

- For every $\sigma \in \Sigma$ and every $\zeta \in [0, r_\sigma]$, $TIME$ *assigns to state* $\sigma(\zeta)$ *the instant* $TIME^{\sigma(\zeta)}$;
- For every $\sigma \in \Sigma$ and every $\zeta \in [0, r_\sigma]$, Π *assigns to each state* $\sigma(\zeta) \in S$ *the parameter interpretation* $\Pi^{\sigma(\zeta)}$.

SO-HL^{2}TSs have the same structure as HL^{2}TSs. An SO-HL^{2}TS extends the associated HL^{2}TS in that: the transitions in an SO-HL^{2}TS are labeled with actions which correspond to the propagation of events (such as published, delivered, executed and discarded); each state of an SO-HL^{2}TS is labeled with the history of events propagation before this state is reached by the system and the pledge of that state ($L(\sigma(\zeta))$), and is also labeled with the time of that state ($TIME^{\sigma(\zeta)}$) and the values of the parameters in that state ($\Pi^{\sigma(\zeta)}$).

The definition of SO-HL^{2}TSs is adapted from the definition of SO-L^{2}TSs in [17]. SO-HL^{2}TSs differ from SO-L^{2}TSs in that: because in every SO-HL^{2}TS the set of functions Σ is defined, every function $\sigma \in \Sigma$ maps to infinitely many states in S, and transitions only take place at state $\sigma(0)$ and $\sigma(r_\sigma)$, no transition take place at the intermediate states, e.g. for a function $\sigma_0 : [0,3] \rightarrow S$, there is a transition to state $\sigma_0(0)$ and a transition from $\sigma_0(3)$ to other states, and there is no transition to state $\sigma_0(1)$ or from state $\sigma_0(1)$ to other states. While in SO-L^{2}TSs there is no such restriction, a transition can take place at any state. In addition, functions L, *TIME* and Π assign labels to the infinitely many states that are obtained by applying the functions in Σ.

When executing a SO-HL^{2}TS, several paths can be generated. A path of a SO-HL^{2}TSs consists of a finite sequence of traces of states which are obtained by applying the functions defined in the SO-HL^{2}TS to their intervals of real numbers (i.e. to apply $\sigma \in \Sigma$ on the interval $[0, r_\sigma]$) and are ordered lexicographically. Moreover, in a path of a SO-HL^{2}TS, there is a transition relation between the last state of one trace of states and the first state of the consequent trace of states. Paths of SO-HL^{2}TSs are defined as follows:

Definition 4.1.3 (Paths of SO-HL^{2}TSs). Given a SO-HL^{2}TS $m = \langle S, s_0, \Sigma, Act, R, AP, L, TIME, \Pi\rangle$, paths of m are defined as follows:

- For every $\sigma \in \Sigma$, $\sigma(0)\ldots\sigma(r_\sigma)$ denotes the trace of states of σ where $\sigma(0)$ is the first state of the trace and $\sigma(r_\sigma)$ is the last state of the trace. $\sigma(0)\ldots\sigma(r_\sigma)$ includes all the states in the set $\{\sigma(\xi)|0 \leq \xi \leq r_\sigma\}$, which can be finite or infinite.
- $\rho = (\sigma_1(0)\ldots\sigma_1(r_{\sigma_1}), \sigma_2(0)\ldots\sigma_2(r_{\sigma_2}), \ldots)$ is a path of m if there exists an $\alpha \in 2^{Act}$ such that for every $\sigma_i(r_{\sigma_i})$ and $\sigma_{i+1}(0)$ with $i \in \mathbb{N}$, there exists an $\alpha \in 2^{Act}$ such that $(\sigma_i(r_{\sigma_i}), \alpha, \sigma_{i+1}(0)) \in R$;
- We use $[\sigma(0)\ldots\sigma(r_\sigma)]$ to denote the equivalence class of traces of states, such that each element is similar up to time with $\sigma(0)\ldots\sigma(r_\sigma)$. $[\sigma(0)\ldots\sigma(r_\sigma)]$ is defined as: $[\sigma(0)\ldots\sigma(r_\sigma)] = \{\sigma'(0)\ldots\sigma'(r_{\sigma'})|\sigma(0) \sim \sigma'(0), \ldots, \sigma(r_\sigma) \sim \sigma'(r_{\sigma'})\}$;
- The states in a path are ordered lexicographically such that, for every $i, j = 1, 2, \ldots$ and $\zeta \in [0, r_{\sigma_i}], \xi \in [0, r_{\sigma_j}]$, there is $\sigma_i(\zeta) \prec \sigma_j(\xi)$ iff either $i < j$, or $i = j$ and $\zeta < \xi$;
- For the states in a path, there is:
  - for every $i, j = 1, 2, \ldots$ and $\zeta \in [0, r_{\sigma_i}], \xi \in [0, r_{\sigma_j}]$, there is $TIME^{\sigma_i(\zeta)} \leq TIME^{\sigma_j(\xi)}$ if $i < j$, and $TIME^{\sigma_i(\zeta)} < TIME^{\sigma_j(\xi)}$ if $i = j$ and $\zeta < \xi$;

  In particular, if $TIME^{\sigma_i(r_{\sigma_i})} < TIME^{\sigma_{i+1}(0)}$ then the transition $(\sigma_i(r_{\sigma_i}), \alpha, \sigma_{i+1}(0))$ takes time, otherwise if $TIME^{\sigma_i(r_{\sigma_i})} = TIME^{\sigma_{i+1}(0)}$ the transition is executed in zero time;
- A path ρ terminates if it is a finite sequence $\sigma_1(0)\ldots\sigma_1(r_{\sigma_1}), \ldots, \sigma_n(0)\ldots\sigma_n(r_{\sigma_n})$. In such case the first state of the trace $\sigma_1(0)$ is denoted by $first\rho$ and the last state $\sigma_n(r_{\sigma_n})$ is denoted by $last\rho$;
- The concatenation of traces $\rho_1 = (\sigma_1(0)\ldots\sigma_1(r_{\sigma_1}), \sigma_2(0)\ldots\sigma_2(r_{\sigma_2}), \ldots)$ and $\rho_2 = (\varsigma_1(0)\ldots\varsigma_1(r_{\varsigma_1}), \varsigma_2(0)\ldots\varsigma_2(r_{\varsigma_2}), \ldots)$, denoted by $\rho_1 \circ \rho_2$, is defined as follows:
  - $\rho_1 \circ \rho_2 = (\sigma_1(0)\ldots\sigma_1(r_{\sigma_1}), \ldots, \sigma_n(0)\ldots\sigma_n(r_{\sigma_n}), \varsigma_1(0)\ldots\varsigma_1(r_{\varsigma_1})\ldots)$ iff ρ_1 terminates at $\sigma_n(r_{\sigma_n})$ and $(\sigma_n(r_{\sigma_n}), \alpha, \varsigma_0(0)) \in R$;
  - $\rho_1 \circ \rho_2 = \rho_1$ iff ρ_1 does not terminate;
  - $\rho_1 \circ \rho_2$ is not defined in other cases;
- λ is an empty hybrid trace such that for any arbitrary hybrid trace ρ, $\rho \circ \lambda = \lambda \circ \rho = \rho$.

4.2 Interaction signatures

As introduced in Chapter 3, service components and service interfaces are modeled by business roles and business protocols, both of which contain a set of interactions. The semantic domain of the interactions is defined by Service-Oriented Configurations. In this section, we show the syntax of interactions, which can be interpreted over the Service-Oriented Configurations. Throughout this section we consider a fixed configuration

$$\Xi = \langle N, WIRE, PLL, \Psi, 2WAY, 1WAY \rangle$$

over which all definitions are given.

A set of interactions declared in SRML is named as an interaction signature. An interaction signature consists of a set of *interaction names* and set of functions that assign each interaction name with the associated parameters. The interaction names are local to the party that is specified, and are coordinated with the corresponding interaction names of the co-party by the wire that connects the two parties. In addition, there is a type associated with each interaction name. This type is essential to derive the events and actions that associate with the interaction.

To define the interaction signature, we first define interaction types. For simplicity, in this thesis we only consider the unsynchronized types (*s&r,r&s,snd,rcv*). The formal definitions of interaction types and interaction signature are the same as Definition 5.1.1 and Definition 5.1.2 from [17], and are defined as follows:

Definition 4.2.1 (Interaction Types)**.** *Interactions can be of one of the following types* $TYPE = \{s\&r, r\&s, snd, rcv\}$.

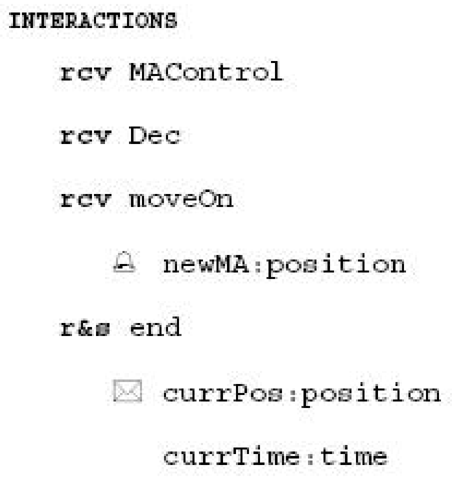
Definition 4.2.2 (Interaction Signature)**.** An interaction signature *is a pair*

$$\langle NAME, PARAM \rangle$$

where:

- *NAME is a* $TYPE-$*indexed family of sets of interaction names;*
- *PARAM consists of five functions* $PARAM_{\text{🕭}}, PARAM_{\text{✉}}, PARAM_{\checkmark}$, $PARAM_{\text{✞}}$ *and* $PARAM_{\text{✗}}$ *such that*:
  - $PARAM_{\text{🕭}}$ *assigns to each name in NAME a D-indexed family of sets of* 🕭*-parameters;*
  - $PARAM_{\text{✉}}, PARAM_{\checkmark}, PARAM_{\text{✞}}$ *and* $PARAM_{\text{✗}}$ *assign to each name* $a \in NAME_{s\&r} \cup NAME_{r\&s}$ *a D-indexed family of sets of* ✉*-parameters,* ✓*-parameters,* ✞*-parameters, and* ✗*-parameters, respectively, such that* $Return \in PARAM_{\text{🕭}}(a)_d$, $Reply \in PARAM_{\text{✉}}(a)_{boolean}$ *and* $useBy \in PARAM_{\text{✉}}(a)_{time}$;
  - *For every* $a \in NAME$ and $P, P' \in PARAM_{\text{🕭}}(a) \cup PARAM_{\text{✉}}(a) \cup PARAM_{\checkmark}(a) \cup PARAM_{\text{✞}}(a) \cup PARAM_{\text{✗}}(a)$, *P and P′ are disjoint.*

In Figure 4.1 we show the set of interactions declared in business role *ETCSC*, which is part of the *Train-Control* module in Appendix A. Business role *ETCSC* can be involved in the interactions *MAControl, Dec, moveOn* and *end*, which have types *rcv, rcv, rcv* and *r&s* correspondingly. *moveOn* is a one-way interaction which has parameter *newMA* associating with its request-event; *end* is a two-way interaction which has parameter *currPos* and *currTime* associating with its request-event.



```
INTERACTIONS
    rcv MAControl
    rcv Dec
    rcv moveOn
        🔔 newMA:position
    r&s end
        ✉ currPos:position
          currTime:time
```

Figure 4.1: The set of interactions of Business Role *ETCSC*

In the rest of this section we consider a fixed signature

$$s = \langle NAME, PARAM \rangle$$

over which all definitions will be given.

As introduced in Chapter 3, in an SO-configuration there are several events and actions associating with every interaction. And in an interaction signature, the type of each interaction determines the role of the party being specified —- which events are published by the party, which events are received by the party. In SRML, these two types of events are distinguished so that certain features of service modules are able to be specified. The syntax of events are defined as *event names* and is the same as Definition 5.1.3 from [17].

Definition 4.2.3 (Event names). *The NAME-indexed families of sets En^{PUB} and En^{RCV} of names of events that can be initialized and received, respectively, is defined as follows:*

If $a \in NAME_{s\&r}$ then $En_a^{PUB} = \{a\text{🔔}, a\checkmark, a\text{✞}, a\text{✗}\}$ and $En_a^{RCV} = \{a\text{✉}\}$;
If $a \in NAME_{r\&s}$ then $En_a^{PUB} = \{a\text{✉}\}$ and $En_a^{RCV} = \{a\text{🔔}, a\checkmark, a\text{✞}, a\text{✗}\}$;
If $a \in NAME_{snd}$ then $En_a^{PUB} = \{a\text{🔔}\}$ and $En_a^{RCV} = \emptyset$;
If $a \in NAME_{rcv}$ then $En_a^{PUB} = \emptyset$ and $En_a^{RCV} = \{a\text{🔔}\}$;

We Define $En = En^{PUB} \cup En^{RCV}$ *as the NAME-indexed family of sets of all event names. Every parameter* $p \in PARAM_{\#}(a)$ *where* $\# \in \{$🔔,✉,✓,✞,✗$\}$ *is said to be a parameter of event* $a\#$.

The type of each interaction also determines the actions that can be performed during the interaction by the party being specified: which actions can be published by the party, which actions can be executed by the party, and etc. In order to be sufficient to specify certain features of service modules, actions are distinguished into four types. The syntax of actions are defined as *action names* and is the same as Definition 5.1.5 from [17]:

Definition 4.2.4 (Action Names)**.** *The NAME-indexed families of sets of publication, delivery, execution and discard action names are defined as follows, where* $a \in NAME$:

$$
\begin{aligned}
Act_a^{PUB} &= \{e! : e \in En_a^{PUB}\} \\
Act_a^{DLV} &= \{e¡ : e \in En_a^{RCV}\} \\
Act_a^{EXC} &= \{e? : e \in En_a^{RCV}\} \\
Act_a^{DSC} &= \{e¿ : e \in En_a^{RCV}\}
\end{aligned}
$$

We define $Act = Act^{PUB} \cup Act^{DLV} \cup Act^{EXC} \cup Act^{DSC}$ *as the NAME-indexed family of sets of all action names associated with signature s.*

As introduced in Chapter 3, in an SO-configuration when a two-way interaction has a positive reply, there is a pledge that associates with the reply-event. In SRML, the syntax of pledge is defined as pledge names and is the same as Definition 5.1.4 from [17]:

Definition 4.2.5 (Pledge names)**.** *The set PLNames of pledge names associated with interaction signature s is* $\{a.pledge : a \in NAME_{r\&s}\}$

In Definition 4.2.5, the reason why a pledge is linked to an interaction name but not an event name is that: a pledge always associate with an reply-event, so linking to the name of the interaction with which the reply-event associates would not cause ambiguity.

The syntax of interactions, including interaction signatures, event names, actions names and pledge names, are interpreted over SO-configurations, and this is called *interaction interpretations* over SO-configurations, events, and parameters. The interaction interpretation of a SRML interaction specification over its semantic domain is a function, which assigns every interaction name with an interaction, assigns every parameter name with a parameter, assigns every event name with an event, and assigns every action name with an action. The definition of interaction interpretation is the same as Definition 5.1.6 from [17] and is shown as follows:

Definition 4.2.6 (Interaction Interpretation)**.** *An* Interaction Interpretation *II* f*or s over a configuration* $\langle N, W, PLL, \Psi, 2WAY, 1WAY\rangle$ *is an injective function that:*

- *assigns an interaction in* $2WAY \cup 1WAY$ *to each name in NAME such that:*
  - *for every* $a \in NAME_{s\&r} \cup NAME_{r\&s}$, $II(a) \in 2WAY$, i.e. *interaction names with type s&r or r&s denote two-way interactions;*

  - *for every $a \in NAME_{snd} \cup NAME_{rcv}$, $II(a) \in 1WAY$, i.e. interaction names with type snd or rcv denote one-way interactions;*

- *assigns an event in E to each event name in En such that for every $a \in NAME$ and event name with form a# for some # $\in \{$🔔$,\boxtimes,$✓$,$✞$,$✗$\}$;*

$$II(a\#) = II(a)\#$$

- *assigns an action to each action name in Act such that for every aNAME and action name with form a#%, for some # $\in \{$🔔$,\boxtimes,$✓$,$✞$,$✗$\}$ and % $\in \{!, ¡, ?, ¿\}$*

$$II(a\#\%) = II(a)\#\%$$

- *assigns a pair $\langle p', view\rangle$ to each parameter name p in PARAM, where:*
  - *p' is a parameter in PP such that:*
    - *if $p \in PARAM_{🔔}(a)_d$ then $p' \in PP(II(a🔔))_d$*
    - *if $p \in PARAM_{\boxtimes}(a)_d$ then $p' \in PP(II(a\boxtimes))_d$, and $II(Reply) = Reply$, $II(useBy) = useBy$*
    - *if $p \in PARAM_{✓}(a)_d$ then $p' \in PP(II(a✓))_d$*
    - *if $p \in PARAM_{✞}(a)_d$ then $p' \in PP(II(a✞))_d$*
    - *if $p \in PARAM_{✗}(a)_d$ then $p' \in PP(II(a✗))_d$*
  - *$view : d_{\mathscr{U}} \longrightarrow d_{\mathscr{U}}$ is the function that defines how the parameter is observed, where $p \in PARAM_d$, is such that if $p = Reply$, or $p = useBy$ then $view = id$,*

4.3 Extension of Business Roles

In the hybrid extension of SRML, a business role specifies the interactions performed by the service component that is modeled by the business role, and the orchestration of changing of states caused by these interactions. As introduced in Chapter 3, a business role contains the declaration of a set of interactions performed by the service component, a set of variables that model the state of the service component and a set of transitions that model the changing of states. Figure 4.2 shows an example of a business role which is part of the *Train-Control* module in Appendix A.

In a business role, the set of variables that model the states of the corresponding service component are defined by an *attribute declaration*. The definition of attribute declaration is the same as Definition 5.2.1 from [17] and is shown as follows:

Definition 4.3.1 (Attribute declaration)**.** *An* attribute declaration *VAR is a D-indexed family of disjoint sets (where D is the set of data types).*

SPECIFICATIONS

BUSINESS ROLE ETCSC **is**

INTERACTIONS

rcv MAControl

rcv Dec

rcv moveOn

🔔 newMA:position

r&s end

✉ currPos:position

currTime:time

ORCHESTRATION

var C:position, V:speed, M:position, C_{dot}:speed, v_0:speed, V_{dot}:acc

initial v_0=100, C=0, t=0, M=500

transition Nego

trigger MAControl🔔?
guard $C_{dot}=v_0$

effect $C_{dot}=v_1$

transition Corr

trigger Dec🔔?
guard $C_{dot}=v_1$

effect $C_{dot}=V$
$\wedge$ $V_{dot}=b$

transition Cont

trigger moveOn🔔?
guard $C_{dot}=v_0$

effect $C_{dot}=v_0$

transition Stop

trigger end🔔?
guard $V_{dot}=b$

effect end✉.currPos=C+M
$\wedge$ end✉.currTime=t

Figure 4.2: Business Role: ETCSC

An example of an attribute declaration is the "*var*" part in Figure 4.2. The data type of each variable is assumed to be included in the set of data types D, and is used for storing data that is needed at different stages in the system evolution.

An attribute declaration is interpreted over a SO-HL^{2}TS, and is defined as an *attribute interpretation*. The attribute interpretation assigns a value to each variable in each state of the SO-HL^{2}TS. Since these variables are local to the service component being modeled, they can be used to model the internal changes of the service component in the computation. The formal definition of attribute interpretation extends Definition 5.2.2 from [17] in that, attribute declarations are interpreted over SO-HL^{2}TSs instead of over SO-TSs, thus each state in which a value is assigned to each variable is represented by function mapping but not a simple state.

Definition 4.3.2 (Attribute interpretation)**.** *An* attribute interpretation Δ *for an attribute declaration VAR over a SO-HL^{2}TS* $\langle S, s_0, \Sigma, Act, R, AP, L, TIME, \Pi\rangle$ *assigns to every state* $\sigma(\zeta) \in S$ *and every variable* $v \in VAR_d$ *an element* $v^{\Delta(\sigma(\zeta))} \in d_{\mathscr{U}}$ *(the value of the variable in that state). Where* $d \in D$ *is the data type,* $d_{\mathscr{U}}$ *is the interpretation of* d *over the fixed algebra* $\mathscr{U}$*,* $\sigma \in \Sigma$ *and* $\zeta \in [0, r_\sigma]$.

The variable v can be seen as a function of time as follows: Let $\rho = (\sigma_1(0)\ldots\sigma_0(r_{\sigma_0}),$ $\sigma_1(0)\ldots\sigma_1(r_{\sigma_2}),\ldots)$ be a path of the SO-HL^{2}TS over which v is interpreted, then the interpretation of $v_\rho : TIME \to d_{\mathscr{U}}$ is defined as:

$$v_\rho(t) = v^{\Delta(\sigma_i(\zeta))} \text{ iff } t = TIME^{\sigma_i(\zeta)}, \text{ for } \forall \sigma_i \text{ in path } \rho \text{ and } \zeta \in [0, r_{\sigma_i}]$$

Note that v_ρ may not be well-defined, and may be a partial function. v_ρ is not well-defined if there exist a $j \geq 0$ such that $TIME^{\sigma_j(r_{\sigma_j})} = TIME^{\sigma_{j+1}(0)}$ and $v^{\Delta(\sigma_j(r_{\sigma_j}))} \neq v^{\Delta(\sigma_{j+1}(0))}$. A well-defined v_ρ is partial if at least one action between the states in ρ takes time, i.e., there exists a $j \geq 0$ such that $TIME^{\sigma_j(r_{\sigma_j})} < TIME^{\sigma_{j+1}(0)}$.

Besides an attribute declaration, an orchestration also includes a set of transitions. As introduced in Chapter 3, a transition consists of a trigger, a guard and a set of effects. In the language specification, triggers and guards are defined over the *language of states*, and effects are defined over the *language of effects*, then transitions are defined over triggers, guards, and effects. We introduce them separately in the following sub-sections. Throughout the remaining of this section we consider the following fixed structures:

- $sig = \langle NAME, PARAM\rangle$ to be an interaction signature where Act is the set of actions associated with sig;
- VAR to be an attribute declaration.

over which all the definitions will be given.
To interpret each part of a business role, we also consider:

- $\Xi = \langle N, W, PLL, \Psi, 2WAY, 1WAY\rangle$ to be a configuration;
- II to be an interaction interpretation for sig over $2WAY \cup 1WAY$ local to some node $n \in N$;

- TR to be a set of transition names;
- $m = \langle S, s_0, \Sigma, Act, R, AP, L, TIME, \Pi \rangle$ to be a SO-HL^{2}TS for Ξ;
- Δ to be an attribute interpretation over m.

4.3.1 Language of states

The language of states (LS) is defined over *state terms*. State terms denote the values of the variables and parameters that associate with events in certain states. The types of state terms include constant, function mapping, variables and parameters. The formal definition of state terms extends Definition 5.2.3 from [17] in that, we define a new type of state term of the form v_{dot}, which is the derivative of a certain variable v to time. This enables us to specify the variables that change with time.

Definition 4.3.3 (State Terms)**.** The *$D-$indexed family of sets $STERM$ of state terms is defined as follows:*

- *If $c \in F_d$ then*

$$c \in STERM_d$$

  for every $d \in D$

- If $f \in F_{<d_1,\ldots,d_n,d_{n+1}>}$ *and* $\overrightarrow{p} \in STERM_{<d_1,\ldots,d_n>}$, *then*

$$f(\overrightarrow{p}) \in STERM_{d_{n+1}}$$

  for every $d_1,\ldots,d_n,d_{n+1} \in D$

- *If $v \in VAR_d$ and $v_{dot} \in VAR_{d'}$, then*

$$v_{dot} \in STERM_{d'}$$

  for every $d, d' \in D$

- *If $a \in NAME$ and $param \in PARAM(a)_d$, then*

$$a.param \in STERM_d$$

  for every $d \in D$

- $t \in STERM_{time}$

- *If $v \in VAR_d$, then*

$$v \in STERM_d$$

  for every $d \in D$

In Definition 4.3.3, $t \in STERM_{time}$ is the variable that stores the time of a certain state. Variable $v_{dot} \in STERM$ is the derivative of a variable $v \in STERM$ to time. By defining the derivative of variables to time in state terms, differential equations which represent the evolutions of SO-HL^{2}TSs in which time-continuous variables involve can be specified with the language of states (see Definition 4.3.5 and Definition 4.3.6).

The interpretation of state terms is adapted from Definition 5.2.4 from [17]. In our definition, state terms are interpreted over the states of SO-HL^{2}TSs, thus they are denoted in the form $\sigma(\zeta)$ ($\sigma \in \Sigma$ in m and $\zeta \in [0, r_\sigma]$). We also interpret the state term v_{dot} in the same way of state term v.

Definition 4.3.4 (Interpretation of State Terms)**.** *The interpretation of a state term $T \in STERM$ in a state $\sigma(\zeta)$ with $\sigma \in \Sigma$ and $0 \le \zeta \le r_\sigma$,* written $[\![T]\!]_{\sigma(\zeta)}$*, is defined as follows, where $II(param) = \langle param', view \rangle$:*

- $[\![c]\!]_{\sigma(\zeta)} = c_{\mathscr{U}}$
- $[\![f(T_1, \dots, T_n)]\!]_{\sigma(\zeta)} = f_{\mathscr{U}}([\![T_1]\!]_{\sigma(\zeta)}, \dots, [\![T_n]\!]_{\sigma(\zeta)})$
- $[\![v_{dot}]\!]_{\sigma(\zeta)} = (v_{dot})^{\Delta(\sigma(\zeta))}$
- $[\![a.param]\!]_{\sigma(\zeta)} = view(II(a).param'^{\Pi^{\sigma(0)}})$
- $[\![t]\!]_{\sigma(\zeta)} = TIME^{\sigma(\zeta)}$
- $[\![v]\!]_{\sigma(\zeta)} = v^{\Delta(\sigma(\zeta))}$

In Definition 4.3.4, $\mathscr{U}$ is the fixed algebra for interpreting the data signature $\langle D, F \rangle$ as defined in Chapter 3, Δ is the attribute interpretation over the SO-HL^{2}TS m and $\Delta(\sigma(\zeta))$ denotes the attribute interpretation at state $\sigma(\zeta)$, *view* is the function that maps a parameter name to the value of that parameter, $TIME^{\sigma(\zeta)}$ is the time at state $\sigma(\zeta)$.

The language of states specifies the possible states a service component can be in with propositions of state terms. It is defined as a set of formulas and extends Definition 5.2.5 from [17]. In our definition, we define more formulas than that in [17], and this makes the language more expressive and thus sufficient for our applications.

Definition 4.3.5 (Language of States)**.** *The language of states LS is defined as follows:*

- $\phi ::= true \mid T_1 = T_2 \mid T_1 < T_2 \mid \phi \wedge \phi \mid \neg\phi$

with $T_1, T_2 \in STERM_d$ for some $d \in D$

Since state terms are interpreted over states of SO-HL^{2}TSs, LS is satisfied by states. The satisfaction of state terms is adapted from Definition 5.2.6 from [17]. In our definition, instead of simple states, LS is satisfied by states obtained by function mapping. The satisfaction relation of *LS* is defined as follows:

Definition 4.3.6 (Satisfaction of the language of states)**.** *The satisfaction relation between a state $\sigma(\zeta)$ and an LS formula $\phi \in LS$, denoted with $\sigma(\zeta) \models \phi$, is defined as follows, where $\sigma \in \Sigma$ and $\zeta \in [0, r_\sigma]$:*

- $\sigma(\zeta) \models true$
- $\sigma(\zeta) \models T_1 = T_2$ *iff* $[\![T_1]\!]_{\sigma(\zeta)} = [\![T_2]\!]_{\sigma(\zeta)}$
- $\sigma(\zeta) \models T_1 < T_2$ *iff* $[\![T_1]\!]_{\sigma(\zeta)} < [\![T_2]\!]_{\sigma(\zeta)}$
- $\sigma(\zeta) \models \neg\phi$ *iff not* $\sigma(\zeta) \models \phi$
- $\sigma(\zeta) \models \phi \wedge \phi'$ *iff* $\sigma(\zeta) \models \phi$ *and* $\sigma(\zeta) \models \phi'$

In *LS*, a differential equation can be specified in the form $v_{dot} = T$, where v_{dot} is the derivative of state term v ($v \in VAR_d$) to time, and T is an arbitrary state term. The semantic of $v_{dot} = T$ in an arbitrary state $\sigma(\zeta)$ is: $[\![v_{dot}]\!]_{\sigma(\zeta)} = [\![T]\!]_{\sigma(\zeta)}$. For example in Figure 4.2, the guard condition $C_{dot} = v_0$ in transition *Nego* is a differential equation of variable C, and is interpreted as $[\![C_{dot}]\!]_{\sigma_1(r_{\sigma_1})} = [\![v_0]\!]_{\sigma_1(r_{\sigma_1})}$.

4.3.2 Language of effects

The language of effects (LE) is defined over *effect terms*. Effect terms denote the values of the variables and parameters that associate with events during certain transitions. The values of the variables in the source state of a transition may be different from that in the target state of the transition, that is, the values may change during a transition. In order to capture this feature, we define effect terms by extending state terms with two new terms v' and t'. Term v' relates to term v in that: term v denotes the value of variable v in the source state of the transition while term v' denotes the value of variable v in the target state of the transition. Term t' relates to term t in that: term t denotes the time instance of the source state while term t' denotes the the time instance of the target state.

The formal definition of effect terms is is adapted from Definition 5.2.7 from [17]. Similar to state terms, in our definition we also have to define a new type of effect term which is the derivative of a certain variable to time .

Definition 4.3.7 (Effect Terms)**.** *The D-indexed family of sets $ETERM$ of effect terms is defined inductively as follows:*

- *If $c \in F_d$ then*
$$c \in ETERM_d$$
for every $d \in D$
- If $f \in F_{<d_1,\ldots,d_n,d_{n+1}>}$ *and* $\overrightarrow{p} \in ETERM_{<d_1,\ldots,d_n>}$, *then*
$$f(\overrightarrow{p}) \in ETERM_{d_{n+1}}$$
for every $d_1, \ldots, d_n, d_{n+1} \in D$

- *If $v \in VAR_d$ and $v_{dot} \in VAR_{d'}$, then*

$$v_{dot} \in ETERM_{d'}$$

for every $d, d' \in D$

- *If $a \in NAME$ and $param \in PARAM(a)_d$, then*

$$a.param \in ETERM_d$$

for every $d \in D$

- $t, t' \in ETERM_{time}$
- *If $v \in VAR_d$, then*

$$v, v' \in ETERM_d$$

for every $d \in D$

Similar to that defined in Definition 4.3.3, effect term v_{dot} is the derivative of variable v to time. In Figure 4.2, terms in effect of transition *Nego*, *Corr*, *Cont*, and *Stop* (such as C_{dot}, V and b) are effect terms.

The interpretation of effect terms is adapted from Definition 5.2.8 from [17]. In our definition, effect terms are interpreted over the transitions of SO-HL^{2}TSs. In the SO-HL^{2}TSs m, these transitions are denoted in the form $\sigma(r_\sigma) \rightarrow \sigma'(0)$, where $\sigma(r_\sigma)$ is the source state of the transition and $\sigma'(0)$ is the target state of the transition. The interpretation of state terms is defined as follows:

Definition 4.3.8 (Interpretation of effect terms)**.** *The interpretation of an effect term $T \in ETERM$ over a transition $\sigma(r_\sigma) \rightarrow \sigma'(0)$,* written $[\![T]\!]_{\sigma(r_\sigma)\rightarrow\sigma'(0)}$, *is defined as follows, where $II(param) = \langle param', view \rangle$:*

- $[\![c]\!]_{\sigma(r_\sigma)\rightarrow\sigma'(0)} = c_{\mathscr{U}}$
- $[\![f(T_1, \ldots, T_n)]\!]_{\sigma(r_\sigma)\rightarrow\sigma'(0)} = f_{\mathscr{U}}([\![T_1]\!]_{\sigma'(0)}, \ldots, [\![T_n]\!]_{\sigma'(0)})$
- $[\![v_{dot}]\!]_{\sigma(r_\sigma)\rightarrow\sigma'(0)} = (v_{dot})^{\Delta(\sigma'(0))}$
- $[\![a.param]\!]_{\sigma(r_\sigma)\rightarrow\sigma'(0)} = view(II(a).param'^{\Pi^{\sigma'(0)}})$
- $[\![v]\!]_{\sigma(r_\sigma)\rightarrow\sigma'(0)} = v^{\sigma(r_\sigma)}$
- $[\![v']\!]_{\sigma(r_\sigma)\rightarrow\sigma'(0)} = v^{\Delta(\sigma'(0))}$
- $[\![t]\!]_{\sigma(r_\sigma)\rightarrow\sigma'(0)} = TIME^{\sigma(r_\sigma)}$

- $[\![t']\!]_{\sigma(r_\sigma)\to\sigma'(0)} = TIME^{\sigma'(0)}$

The subscript of the interpretation of effect terms in definition 4.3.8 means that the terms are interpreted over transition $\sigma(r_\sigma) \to \sigma'(0)$, whose target state is the first state of the trace of states $\sigma'(0)\ldots\sigma'(r_{\sigma'})$.

The language of effects (LE) specifies the results of transitions with propositions of effect terms. It is defined as a set of formulas and adapted from Definition 5.2.5 from [17]. Since in our case the publication of events as the result of transition is not studied, we exclude such events from the definition of LE.

Definition 4.3.9 (Language of Effects)**.** *The Language of effects LE is defined inductively as follows:*

- $\phi ::= true \mid T_1 = T_2 \mid \phi \wedge \phi' \mid \neg\phi$

where $T_1, T_2 \in ETERM_d$ for some $d \in D$.

Since effect terms are interpreted over transitions of SO-HL^{2}TSs, LE is satisfied by transitions. The satisfaction is adapted from Definition 5.2.10 from [17]. In our definition, instead of transitions between simple states, LE is satisfied by transitions between states obtained by function mapping. The satisfaction relation of *LE* is defined as follows:

Definition 4.3.10 (Satisfaction for the language of effects)**.** *The satisfaction relation between a transition $\sigma(r_\sigma) \to \sigma'(0)$ and an LE formula $\phi \in LE$, denoted with $\sigma(r_\sigma) \to \sigma'(0) \models \phi$, is defined as follows, where $\sigma \in \Sigma$:*

- $\sigma(r_\sigma) \to \sigma'(0) \models true$
- $\sigma(r_\sigma) \to \sigma'(0) \models T_1 = T_2$ *iff* $[\![T_1]\!]_{\sigma(r_\sigma)\to\sigma'(0)} = [\![T_2]\!]_{\sigma(r_\sigma)\to\sigma'(0)}$
- $\sigma(r_\sigma) \to \sigma'(0) \models \phi \wedge \phi'$ *iff* $\sigma(r_\sigma) \to \sigma'(0) \models \phi$ *and* $\sigma(r_\sigma) \to \sigma'(0) \models \phi'$
- $\sigma(r_\sigma) \to \sigma'(0) \models \neg\phi$ *iff not* $\sigma(r_\sigma) \to \sigma'(0) \models \phi$

Like in *LS*, a differential equation in *LE* can be specified in the form $v_{dot} = T$, where v_{dot} is the derivative of variable v ($v \in ETERM_d$) to time, and T is an arbitrary effect term. The satisfaction of $v_{dot} = T$ by a transition $\sigma(r_\sigma) \to \sigma'(0)$ is: $[\![v_{dot}]\!]_{\sigma(r_\sigma)\to\sigma'(0)} = [\![T]\!]_{\sigma(r_\sigma)\to\sigma'(0)}$, which means after transition $\sigma(r_\sigma) \to \sigma'(0)$, differential equation $v_{dot} = T$ holds along the trace of states $\sigma'(0)\ldots\sigma'(r_{\sigma'})$. For example in Figure 4.2, the effect $C_{dot} = v_1$ of transition *Nego* is a differential equation of variable C, and is satisfied as $[\![C_{dot}]\!]_{\sigma_0(r_{\sigma_0})\to\sigma_1(0)} = [\![v_1]\!]_{\sigma_0(r_{\sigma_0})\to\sigma_1(0)}$, which means after transition $\sigma_0(r_{\sigma_0}) \to \sigma_1(0)$, differential equation $C_{dot} = v_1$ holds along the trace of states $\sigma_1(0)\ldots\sigma_1(r_{\sigma_1})$.

4.3.3 Transition specifications

Transitions of SRML are defined on the basis of *LS* and *LE*. As introduced in Chapter 3, a transition of SRML includes a *trigger*, which specifies the events that cause the transition; a *guard*, which specifies the condition in which the transition can take place; an *effect*, which specifies the new values of variables and parameters as the result of the transition. For example in Figure 4.2, transition *Nego* takes place when trigger *MAControl*🔔*?* happens and guard $C_{dot} = v_0$ is satisfied, as a result effect $C_{dot} = v_1$ holds in each state of the trace of states whose first state is the target state of this transition.

A transition specification includes three functions: *tri*, *gua* and *eff*, which maps a transition name in *TR* to trigger, guard, effect of that transition correspondingly.

Definition 4.3.11 (Transition specification)**.** Given a transition name $tr \in TR$, a transition specification *is a tuple*

$$\langle tri(tr), gua(tr), eff(tr)\rangle$$

where:

- *tri* is a function such that: $tri(tr) \in Act^{EXC}$ *(see Definition 4.2.4 for the definition of* Act^{EXC}*) specifies the events that triggers the transition,respectively;*
- *gua* is a function such that: $gua(tr) \in LS$ specifies the conditions that must be satisfied for the transition to take place;
- *eff* is a function such that: $eff(tr) \in LE$ specifies the results of the transition.

In Definition 4.3.11, Act^{EXC} is the set of actions that are executed during transitions. Only when the trigger conditions are from the set of actions that are being executed, they could be the possible actions that trigger the transitions.

In a transition specification, since guard condition is specified by LS and effect condition is specified by LE, LS is satisfied by states of SO-HL^{2}TSs and LE is satisfied by transitions of SO-HL^{2}TSs, transition specifications are satisfied by SO-HL^{2}TSs. A SO-HL^{2}TS *m* satisfies a transition specification if, when the events of trigger are executed and processed, and the conditions of guard are satisfied by the current state, then the results of effect can be observed. This is defined formally as follows:

Definition 4.3.12 (Transition satisfaction)**.** *The SO-HL*^{2}*TS m satisfies a transition specification*

$$\langle tri(tr), gua(tr), eff(tr)\rangle$$

iff : for every transition $\sigma(r_\sigma) \to \sigma'(0)$ the following properties hold:

If $tri(tr) \in Act^{EXC}$, $II(tri(tr)) \in PRC^{\sigma(r_\sigma)\to\sigma'(0)}$, *and* $\sigma(r_\sigma) \models gua(tr)$, *then*

$$II(tri(tr)) \in EXC^{\sigma(r_\sigma)\to\sigma'(0)} \text{ and}$$
$$\sigma(r_\sigma) \to \sigma'(0) \models eff(tr);$$

4.3.4 Formalization of business roles

As introduced in the beginning of this section, a business role models a service component by declaring a set of interactions in which the service component is involved and an orchestration. In detail, the orchestration include an attribute declaration and a set of transitions. Based on the definition of interactions, attribute declarations and transition specification, a business role can be formalized as follows:

Definition 4.3.13 (Business role)**.** *A* business role *is a triple*

$$\langle sig, VAR, ORCH \rangle$$

where:

- *sig is an interaction signature;*
- *VAR is an attribute declaration;*
- *ORCH is a set of transition specifications for sig and VAR.*

The form of Definition 4.3.4 is the same as Definition 5.2.13 from [17]. But in this thesis, *VAR* (the attribute declaration) and *ORCH* (the set of transition specifications) are defined differently from that in [17].

In a business role, the orchestration specifies the computation performed by the service component which is modeled by the business role, thus the satisfaction of a business role can be seen equal to the satisfaction of the transitions declared in its orchestration. Since transitions of orchestrations are satisfied by SO-HL^{2}TSs, business roles should also be satisfied by SO-HL^{2}TSs. A SO-HL^{2}TS satisfies a business role if and only if it satisfies every transition specification in that business role.

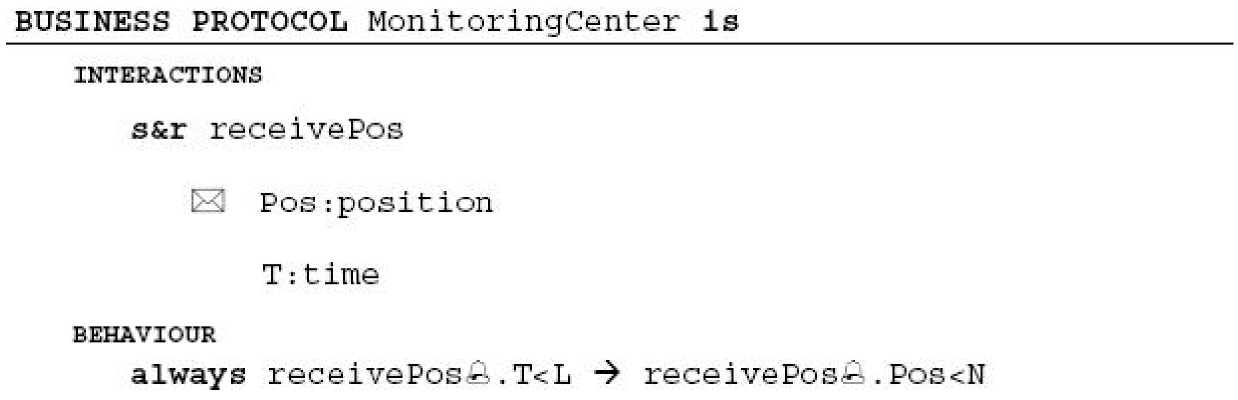
Definition 4.3.14 (Satisfaction of business role)**.** *The SO-HL^{2}TS m is said to satisfy a business role $\langle sig, VAR, ORCH \rangle$ iff $m \models \langle tri(tr), gua(tr), eff(tr) \rangle$ for every $\langle tri(tr), gua(tr), eff(tr) \rangle \in ORCH$* ($tr \in TR$ is a transition name).

Take business role *ETCSC* in Figure 4.2 for example, *tri(Nego)* $\equiv$ *MAControl🔔?*, *gua(Nego)* $\equiv C_{dot} = V_0$, *eff(Nego)* $\equiv C_{dot} = V_1$. The SO-HL^{2}TS m satisfies transition specification of *Nego*: $m \models \langle tri(Nego), gua(Nego), eff(Nego) \rangle$. The same as transition Nego, there is $m \models \langle tri(Corr), gua(Corr), eff(Corr) \rangle$, $m \models \langle tri(Cont), gua(Cont), eff(Cont) \rangle$, and $m \models \langle tri(Stop), gua(Stop), eff(Stop) \rangle$. Thus m satisfies business role *ETCSC*.

4.4 Extension of Business Protocols

In a service module, a business protocol consists of a set of interactions and a set of *behaviors*. These interactions are involved in by the service interface (requires-interface or provides-interface) that is modeled by the business protocol. The behaviors specifies the way the service that is modeled by the service module engages in the interactions, that is, they specify the features that the outside party can expect the service to hold.

Figure 4.3 shows an example of the business protocol *MonitoringCenter* which is part of the *Train-Control* module in Appendix A. In the example, *receivePos* is an interaction and ***always*** *receivePos🔔.T<L* $\rightarrow$ *receivePos🔔.Pos<N* is a behavior constraint.



```
BUSINESS PROTOCOL MonitoringCenter is
  INTERACTIONS
    s&r receivePos
      ✉  Pos:position
         T:time
  BEHAVIOUR
    always receivePos🔔.T<L → receivePos🔔.Pos<N
```

Figure 4.3: Business Protocol: MonitoringCenter

In this section, we define a *hybrid behavior constraint* with which the behaviors that specify the time-continuous evolution of services (also can be seen as hybrid systems) can be declared. In order to do this, we have to define state predicates, hybrid programs and dTL formulas, based on which the hybrid behavior constraint is defined. Particularly, hybrid programs and dTL formulas are essential in the verification of behaviors declared with the hybrid behavior constraint. We introduce these in the following sub-sections. Throughout the remaining of this section we consider the following fixed structures:

- $\Xi = \langle N, WIRE, PLL, \Psi, 2WAY, 1WAY \rangle$ to be a configuration;
- $sig = \langle NAME, PARAM \rangle$ to be an interaction signature, where Act is the set of actions associated with sig;
- II to be an interaction interpretation for sig over $2WAY \cup 1WAY$ local to some node $n \in N$;
- $m = \langle S, s, Act', R, \Sigma, L, AP, TIME, \Pi \rangle$ to be the SO-HL^{2}TS that abstracts some model of computation for Ξ; $II[Act] \subseteq Act'$ by definition. (Act' are superset of the actions that can be performed by node n).

over which all the definitions are given.

4.4.1 State predicates

State predicates are the predicates that holds in certain states of SO-HL^{2}TSs. They are defined on the basis of *terms*. A *term* can be a constant, an operation, the value of some parameter or the time associated with a state. Different from state terms, terms doesn't include variables. This is because business protocols model service interfaces which perform no business process. The formal definition of terms is the same as Definition 5.3.1 from [17], and is shown as follows:

Definition 4.4.1 (Terms)**.** *The D-indexed family of sets TERM is defined inductively as follows:*

- *If* $c \in F_d$ *then*
$$c \in TERM_d$$
for every $d \in D$
- *If* $f \in F_{<d_1,\ldots,d_n,d_{n+1}>}$ *and* $\overrightarrow{p} \in TERM_{<d_1,\ldots,d_n>}$ *then*
$$f(\overrightarrow{p}) \in TERM_{d_{n+1}}$$
for every $d_1,\ldots,d_n,d_{n+1} \in D$
- *If* $a \in NAME$ *and* $p \in PARAM(a)_d$*, then*
$$a.p \in TERM_d$$
for every $d \in D$
- $t \in TERM_{time}$

The interpretation of terms is adapted from Definition 5.3.2 from [17]. In our definition, terms are interpreted over the states of SO-HL^{2}TSs, thus they are denoted in the form $\sigma(\zeta)$ ($\sigma \in \Sigma$ in m and $\zeta \in [0, r_\sigma]$). The interpretation of terms is shown as follows:

Definition 4.4.2 (Interpretation of terms)**.** *The interpretation of a term* $T \in TERM$ *in a state* $\sigma(\zeta)$ *with* $\sigma \in \Sigma$ *and* $0 \le \zeta \le r_\sigma$*, written* $[\![T]\!]_{\sigma(\zeta)}$, is defined as follows, where $II(param) = \langle param', view \rangle$

- $[\![c]\!]_{\sigma(\zeta)} = c_{\mathscr{U}}$
- $[\![f(T_1,\ldots,T_n)]\!]_{\sigma(\zeta)} = f_{\mathscr{U}}([\![T_1]\!]_{\sigma(\zeta)},\ldots,[\![T_n]\!]_{\sigma(\zeta)})$
- $[\![a.p]\!]_{\sigma(\zeta)} = view(II(a).p'^{\Pi^{\sigma(\zeta)}})$
- $[\![time]\!]_{\sigma(\zeta)} = TIME_{\sigma(\zeta)}$

State predicates specify the possible states a service interface can be in with the atomic predicates of the corresponding SO-HL^{2}TSs and propositions of terms. State predicates are defined as a set of formulas and extend Definition 4.2.6 from [17]. In our definition, we define more formulas than that in [17], and this makes the language more expressive and thus sufficient for our applications. State predicates are defined as follows:

Definition 4.4.3 (State predicates). *The language SP of state predicates is defined as follows:*

$$SP ::= ap \mid T_1 = T_2 \mid T_1 < T_2 \mid \phi \to \phi'$$

with $ap \in AP$, and $T_1, T_2 \in TERM_d$ for some $d \in D$.

Since terms are interpreted over states of SO-HL^{2}TSs, state predicates are satisfied by states. The satisfaction of state predicates is adapted from Definition 5.3.6 from [17]. In our definition, we interpreted also the extended formulas of state predicates. And instead of simple state, state predicates are satisfied by states obtained by function mapping.

Definition 4.4.4 (Satisfaction of state predicates). *The satisfaction relation between a state $\sigma(\zeta)$ in SO-HL^{2}TS m and a state predicate sp, written $\sigma(\zeta) \models sp$, where $\sigma \in \Sigma$ and $0 \leq \zeta \leq r_\sigma$ is defined as follows:*

- $\sigma(\zeta) \models ap$ *iff* $ap \in L(\sigma(\zeta))$;
- $\sigma(\zeta) \models T_1 = T_2$ *iff* $[\![T_1]\!]_{\sigma(\zeta)} = [\![T_2]\!]_{\sigma(\zeta)}$
- $\sigma(\zeta) \models T_1 < T_2$ *iff* $[\![T_1]\!]_{\sigma(\zeta)} = [\![T_2]\!]_{\sigma(\zeta)}$
- $\sigma(\zeta) \models \phi \to \phi'$ *iff* $\sigma(\zeta) \models \phi \to \sigma(\zeta) \models \phi'$

Take Figure 4.3 for example, $receivePos🔔.T < L \to receivePos🔔.Pos < M$ is a state predicate. And in the SO-HL^{2}TS m, there is $\sigma_3(0) \models receivePos🔔.T < L \to receivePos🔔.Pos < M$.

4.4.2 Hybrid programs and dTL formulas

Hybrid programs are extensions of the Dynamic Logic programs. They generalize real-time programs [40] to hybrid changes and can be used to describe the hybrid behaviors of SO-HL^{2}TSs. Hybrid programs are defined over language of effects (LE, see Definition 4.3.9) and are interpreted over paths of SO-HL^{2}TSs.

Definition 4.4.5 (Hybrid Program). The set *HP* of hybrid programs is inductively defined as follows:

- *If $\beta \in LE$, then $\beta \in HP$;*
- *$\chi \in HP$ is an empty hybrid program;*
- *If $\beta, \gamma \in$ HP, then $\beta \cup \gamma \in$ HP;*
- *If $\beta, \gamma \in$ HP, then $\beta; \gamma \in$ HP;*
- *If $\beta \in$ HP, then $\beta^* \in$ HP.*

In definition 4.4.5, $;$, $\cup$ and $*$ are Dynamic Logic propositional operators which have the meanings composition, choice and iteration respectively. $(C_{dot} = v_0)^*$ and $C_{dot} = v_1; C_{dot} = V \wedge V_{dot} = b; end\boxtimes.currPos = C - C_0 \wedge end\boxtimes.currTime = t' - t)$ are examples of hybrid programs that are constructed from business role *ETCSC*.

A hybrid program is interpreted as a set of paths of a SO-HL^{2}TS, each transition in which satisfies the corresponding LE formula that composites the hybrid program. In particular, the composition and iteration operators in the hybrid program are interpreted as the concatenation of paths of the SO-HL^{2}TS, and the choice operator is interpreted as the union of paths of the SO-HL^{2}TS. The interpretation of hybrid programs is defined as follows:

Definition 4.4.6 (Interpretation of hybrid programs)**.** The interpretation of a hybrid program β which is a set of paths of the SO-HL^{2}TS m, written $[\![\beta]\!]$, is defined as follows,:

- If $\beta \in LE$, then $[\![\beta]\!] = \{\sigma(0)\dots\sigma(r_\sigma) | \sigma \in \Sigma$ and $\exists \sigma' \in \Sigma$ such that $\sigma'(r_{\sigma'}) \to \sigma(0) \models \beta\}$ (the satisfaction of LE over transitions, $\models$, can be found in Definition 4.3.10);
- $[\![\chi]\!] = \{\lambda\}$;
- $[\![\beta \cup \gamma]\!] = [\![\beta]\!] \cup [\![\gamma]\!]$;
- $[\![\beta;\gamma]\!] = \{\sigma \circ \varsigma : \sigma \in [\![\beta]\!], \varsigma \in [\![\gamma]\!]$ when $\sigma \circ \varsigma$ is defined $\}$;
- $[\![\beta^*]\!] = \bigcup_{n\in\mathbb{N}} [\![\beta^n]\!]$, where $[\![\beta^0]\!] = \{\lambda\}$ and $[\![\beta^n]\!] = [\![\beta^{n-1};\beta]\!]$.

$[\![\beta]\!]$ is also called the trace semantics of hybrid program β.

dTL formulas are defined on the basis of state predicates and hybrid programs. The definition of dTL formulas is adapted from [41]. The difference from that in [41] is that, in our definition the atomic predicate is replaced by state predicate and the hybrid programs are defined differently (see Definition 4.4.5 and Definition 4.4.6). The definition of dTL formulas is shown as follows:

Definition 4.4.7 (dTL formulas)**.** *the dTL state formula ϕ and dTL trace formula ϕ are inductively defined as follows:*

$$\begin{aligned}\phi &::= true \mid sp \mid \neg\phi, \phi \wedge \phi', \phi \vee \phi', \phi \to \phi', \forall t'\phi, \exists t'\phi \mid [\beta]\pi, \langle\beta\rangle\pi \\ \pi &::= \phi \mid \Box\phi, \Diamond\phi\end{aligned}$$

with $sp \in SP$ and $\beta \in HP$.

The t' in Definition 4.4.7 is the effect term that is defined in Definition 4.3.7. In Definition 4.4.7, t' works as a free variable.

In dynamic logic as introduced in Chapter 2, modality "[]" in the dTL formula $[\beta]\pi$ represents all possible behaviors of a system β, modality "$\langle\rangle$" in the dTL formula $\langle\beta\rangle\pi$ represents the existence of some behavior of β, both of which satisfy condition π. And in our definition, β is a hybrid program and π is a dTL trace formula which is allowed to refer to all states along a path of a SO-HL^{2}TS using temporal operators $\Box$ and $\Diamond$. The temporal trace formula $\Box\phi$ expresses that the formula ϕ holds at all the states along a path selected by $[\beta]$ or $\langle\beta\rangle$. Dually, the trace formula $\Diamond\phi$ expresses that ϕ holds at some state in such a path. e.g. state formula $\langle\beta\rangle\Box\phi$ expresses that the state formula ϕ holds in every state along at least one path in $[\![\beta]\!]$, and state formula $[\beta]\Diamond\phi$ expresses that the state formula ϕ holds in at least one state along every path in $[\![\beta]\!]$. Similarly are $[\beta]\Box\phi$ and $\langle\beta\rangle\Diamond\phi$. Formulas without $\Box$ and $\Diamond$ operators are called non-temporal dynamic logic formulas [42, 29, 43]. If they follow some paths of a SO-HL^{2}TS selected by a hybrid program, then they should hold for the last states of these paths. e.g. $[\beta]\phi$ expresses that ϕ is true at the end of each path in $[\![\beta]\!]$. In contrast, $[\beta]\Box\phi$ expresses that ϕ is true along all states of every path in $[\![\beta]\!]$.

dTL state formulas are satisfied by states of SO-HL^{2}TSs and dTL trace formulas are satisfied by paths of SO-HL^{2}TSs. The satisfaction of dTL formulas is adapted from [41]. In our definition, the states and paths that satisfy the formulas are that of the SO-HL^{2}TSs *m*. The satisfaction of dTL formulas is defined as follows:

Definition 4.4.8 (Satisfaction of dTL formulas)**.** *The satisfaction relation between a state $\sigma(\zeta)$ in SO-HL^{2}TS m and a dTL state formula ϕ, written $\sigma(\zeta) \models \phi$, where $\sigma \in \Sigma$ and $0 \leq \zeta \leq r_\sigma$ is defined as follows:*

- $\sigma(\zeta) \models$ *true;*
- $\sigma(\zeta) \models$ *sp as defined in Def. 4.4.4;*
- $\sigma(\zeta) \models \neg\phi$ iff not $\sigma(\zeta) \models \phi$;
- $\sigma(\zeta) \models \phi \wedge \phi'$ *iff* $\sigma(\zeta) \models \phi$ *and* $\sigma(\zeta) \models \phi'$*;*
- $\sigma(\zeta) \models \phi \vee \phi'$ *iff* $\sigma(\zeta) \models \phi$ *or* $\sigma(\zeta) \models \phi'$*;*
- $\sigma(\zeta) \models \phi \rightarrow \phi'$ *iff* $\sigma(\zeta) \models \phi \rightarrow \sigma(\zeta) \models \phi'$*;*
- $\sigma(\zeta) \models \forall t'\phi$ *iff* $\sigma(\zeta) \models \phi$ *for all* $[\![t']\!]_{\sigma'(\xi)}$ *where* σ' *is an arbitrary function in* Σ *and* ξ *is an arbitrary real number in* $[0, r_{\sigma'}]$ *;*
- $\sigma(\zeta) \models \exists t'\phi$ *iff* $\sigma(\zeta) \models \phi$ *for some* $[\![t']\!]_{\sigma'(\xi)}$ *where* σ' *is some function in* Σ *and* ξ *is some real number in* $[0, r_{\sigma'}]$ *;*
- $\sigma(\zeta) \models [\beta]\pi$ iff for each trace $\rho \in \beta$ with $first\rho = \sigma(\zeta)$, if the satisfaction relation between ρ and π is defined then $\rho \models \pi$;
- $\sigma(\zeta) \models \langle\beta\rangle\pi$ iff there is a trace $\rho \in \beta$ with $first\rho = \sigma(\zeta)$, if the satisfaction relation between ρ and π is defined then $\rho \models \pi$.

The satisfaction relation between a path ρ of the SO-HL^{2}TS m and a dTL trace formula π, written $\rho \models \pi$, is defined as follows:

- $\rho \models \phi$ iff ρ terminates and $last\rho \models \phi$, whereas the satisfaction relation between ρ and ϕ is not defined if ρ does not terminate;

- $\rho \models \Box\phi$ iff $\sigma_i(\zeta) \models \phi$ for all positions (i, ζ) of ρ;
- $\rho \models \Diamond\phi$ iff $\sigma_i(\zeta) \models \phi$ for some position (i, ζ) of ρ.

4.4.3 Behavior constraints

In order to be able to specify the behaviors in business protocols, four behavior constraints are defined in [17]. Theses behavior constraints "capture the conditions under which events are executed or discarded and the conditions under which events are published" [17]. They are named with ***initiallyEnabled***, ***enables***, ***enables...until*** and ***ensures***. In Table 4.1 we list the informal and formal interpretations of the behavior constraints.

Behavior constraints	**Informal interpretation**	**Formal interpretation**
initiallyEnabled *e?*	The event e will never be discarded.	$A[true_{\{\neg e¿\}} W_{e?} true]$
s ***enables*** *e?*	Once *s* becomes true, *e* can not be discarded ever again. And before *s* becomes true, *e* can not be executed.	$A[\neg s_{\{\neg e?\}}$ $W(s \wedge \neg EF < e¿ > true)]$
s ***enables*** *e?* ***until*** *w*	*e* can only be executed and can not be discarded, after *s* becomes true but only while *w* has never been true. Once *w* becomes true, *e* can not be executed anymore.	$(\neg E[\neg w_{\{true\}}$ $U(s \wedge E[\neg w_{\{true\}} U_{\{e¿\}} true])])$ $\wedge(AG(w \Rightarrow \neg EF < e? > true))$ $\wedge(A[true_{\{\neg e?\}} W s])$
s ***ensures*** *e!*	After *s* becomes true *e* will be published, but not before.	$A[\neg s_{\{\neg e!\}} W(s \wedge AF[e!]true)]$

Table 4.1: Informal and formal interpretation of behavior constraints: $e?, e!, e¿ \in Act$ and $s, w \in SP$ where Act is the set of actions in the SO-HL^{2}TS m and SP is the set of state predicates.

Since the evolutions of hybrid systems involve both discrete changes and time-continuous changes, it is needed to specify such hybrid changes in business protocols that are interpreted over SO-HL^{2}TSs. In order to do this, we define a new behavior constraint ***always*** *sp*, which can specify the behaviors that are caused by hybrid changes of services that are abstracted by SO-HL^{2}TSs. In the behavior constraint, *sp* is a state predicate. ***always*** *sp* is called the *hybrid behavior constraint*, and it is informally interpreted as "*sp* holds at each state of all the paths of a certain hybrid program β. β is defined in the interpretation of hybrid behavior constraint that is shown as follows:

Definition 4.4.9 (Interpretation of hybrid behavior constraint)**.**

For the service module M in which *"**always** sp"* is declared, let β be a hybrid program which represents a hybrid behavior of M, i.e. $[\![\beta]\!]$ is the set of all the paths of SO-HL^{2}TS m starting from $\sigma_0(0)\ldots\sigma_0(r_{\sigma_0})$ selected by M. Then *"**always** sp"* is interpreted as:

$$[\beta]\Box sp$$

where $sp \in SP$

In Definition 4.4.9, the hybrid program β for a certain service module only exists under some conditions, i.e. there exists an SO-HL^{2}TS that satisfies all the business roles of that service module. In Chapter 5 we will show how to construct such a hybrid program β.

The set of behavior constraints declared in a business protocol extends Definition 5.3.7 from [17] with the hybrid behavior constraint *"**always** sp"* and is shown as follows:

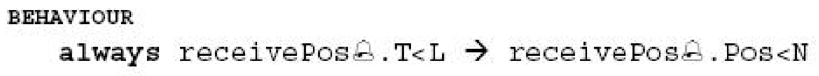
Definition 4.4.10 (Behavior constraints)**.** *The set ABRV of behavior constraints that can be specified for sig in a business protocol is defined as follows:*

$$ABRV ::= \textbf{\textit{initiallyEnabled}}\ e? \mid s\ \textbf{\textit{enables}}\ e? \mid s\ \textbf{\textit{enables}}\ e?\ \textbf{\textit{until}}\ w \mid$$
$$s\ \textbf{\textit{ensures}}\ e! \mid \textbf{\textit{always}}\ s$$

where $e?, e! \in Act$ of m and $s, w \in SP$.

As shown in Table 4.1 and Definition 4.4.9, *"**always** sp"* is interpreted over all the states along some paths of the SO-HL^{2}TS *m*, which include both the continuous parts and the discrete parts of the paths; and other behavior constraints are only interpreted over certain states in some paths of *m*, which include only the discrete parts of paths. So we call ***always*** *s* hybrid behavior constraint and the rest of the behavior constraints normal behavior constraints.

The normal behavior constraints are interpreted as UCTL formulas that can be found in [17], and the hybrid behavior constraint is interpreted in Definition 4.4.9. Figure 4.4 shows the behavior specified with the hybrid behavior constraint of business protocol *MOC*. In Figure 4.4, *L* and *N* are constants. *L* is the maximum time and *N* is the maximum displacement for the movement of a train.



```
BEHAVIOUR
   always receivePos🔔.T<L → receivePos🔔.Pos<N
```

Figure 4.4: The behavior constraint of business protocol *MOC*

4.4.4 Formalization of business protocol

As introduced in the beginning of this section, a business protocol is a formal specification of a service provides-interface or requires-interface. In the business protocol, a set of interactions in which the service interface involves and a set of behaviors that the service module in which the service interface is included can expect to hold are declared. According to this, we give the formal definition of business protocol which is the same as Definition 5.3.9 from [17] and is shown as follows:

Definition 4.4.11 (Business Protocol)**.** A business protocol *is a pair*

$$\langle sig, BEHAVIOR \rangle$$

where:

- *sig is an interaction signature;*
- *BEHAVIOR $\subset$ ABRV is a set of behavior constraints defined over sig.*

Take the business protocol *MonitoringCenter* in Figure 4.3 for example. Suppose it is formalized as $MonitoringCenter = \langle sig, BEHAVIOR \rangle$. Then *sig* includes the name and parameters of interaction *receivePos*, and $BEHAVIOR = \{$***always*** *receivePos*🔔$.T < L \rightarrow receivePos\Phi.Pos < N\}$

As introduced in the beginning of this section, the behaviors declared in a business protocol of a service module specify the features that the outside parties can expect the service which is modeled by the service module to hold. Thus the satisfaction of a business protocol can be seen equal to the satisfaction of the dTL formulas that are declared as behaviors of that business protocol. Since dTL formulas are satisfied by states and paths of SO-HL^{2}TSs, we define that an SO-HL^{2}TS satisfies a business protocol if and only if it satisfies the behaviors specified in the business protocol. The formal definition of satisfaction of business protocol is adapted from Definition 5.3.10 from [17]. In our definition, we simplify the definition by discarding provides-interfaces and requires-interfaces which are not distinguished in this thesis. The definition is shown as follows:

Definition 4.4.12 (Satisfaction of Business Protocol)**.** *The SO-HL^{2}TS m satisfies a business protocol $\langle sig, BEHAVIOR \rangle$ iff:*

- *For each normal behavior constraint $n \in BEHAVIOR$, $s_0 \models n$ where the satisfaction relation between a state and n which is interpreted as a UCTL formula is defined in [36, 37]*;
- *For each hybrid behavior constraint $h \in BEHAVIOR$, $s_0 \models h$ where the satisfaction relation between a state and h which is interpreted as a dTL formula is defined in Definition 4.4.8*;

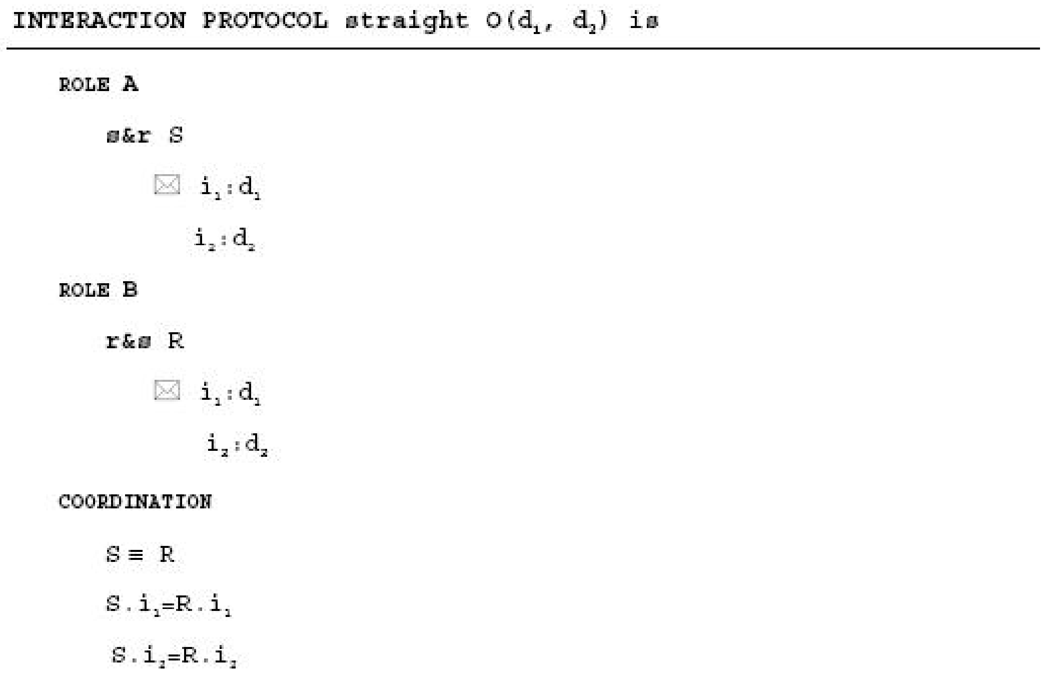
4.5 Extension of Interaction Protocols

As introduced in Chapter 3, wires in service modules are modeled by interaction protocols. An interaction protocol correlates a pair of interactions, each of which is declared in a business role or a business protocol. Figure 4.5 shows an example of a business protocol.

Since in SRML, interaction protocols are designed as separate, reusable entities, they are labeled with *connectors* that coordinates the interactions in which service components or service interfaces are involved. A business protocol can be labeled with different *connectors*, and thus can model different wires in a service module.

In this section, we first introduce coordinations over which interaction protocols are defined; and then introduce interaction protocols and connectors, by which the wires of service modules can be modeled. Throughout the remaining of this section we consider the following fixed structures:

- $\Xi = \langle N, W, PLL, \Psi, 2WAY, 1WAY \rangle$ be a configuration;



INTERACTION PROTOCOL straight $O(d_1, d_2)$ is

ROLE A

  s&r S

    ✉ $i_1 : d_1$

     $i_2 : d_2$

ROLE B

  r&s R

    ✉ $i_1 : d_1$

     $i_2 : d_2$

COORDINATION

  $S \equiv R$

  $S.i_1 = R.i_1$

  $S.i_2 = R.i_2$

Figure 4.5: An interaction protocol of service module *Train-Control*

- partyA and partyB be two interaction signatures;
- II_1 and II_2 be an interaction interpretation over $2WAY \cup 1WAY$ for partyA and partyB, respectively. We will use II to denote $II_1 \cup II_2$;
- $m = \langle S, s_0, \Sigma, Act, R, AP, L, TIME, \Pi \rangle$ be the SO-HL^{2}TS for Ξ;

over which all the definitions will be given.

4.5.1 Coordinations

As introduced in the beginning of this Chapter, an interaction protocol correlates two interactions. Since interaction protocols are reusable, they are defined independently of the names of the interactions, and a wire of a service module can be modeled by one interaction protocol or several interaction protocols. Therefore, when interaction protocols are used to model a wire, we need functions that map form the interactions declared in the interaction protocols to the interactions declared in the business roles or business protocols that model the service components or service interfaces connected by the wire. Such functions are called *signature morphisms*.

A signature morphism maps each interaction name and each parameter that associate with the interaction in one interaction signature with an interaction name and a parameter that associate with the interaction in another interaction signature, and not every interaction in the target signature needs to be in the mapping of the signature morphism. The formal definition of signature morphism is the same as Definition 5.4.4 from [17], and is shown as follows:

Definition 4.5.1 (Signature morphism)**.** A signature morphism, *map, from an interaction signature $\langle NAME, PARAM \rangle$ to another interaction signature $\langle NAME', PARAM' \rangle$ is a function that:*

- *assigns to each interaction name $a \in NAME_t$ with $t \in TYPE$ an interaction name $a' \in NMAE'_t$;*
- *assigns to each parameter $p \in PARAM_{\#}(a)_d$, where $\# \in \{$🔔$, ✉, ✓, ✟, ✗\}$, a parameter $p' \in PARAM'_{\#}(a')_d$ such that $map(a) = a'$.*

A *coordination* correlates the interaction names and parameters of two interactions. In order to be able to specify the interaction names and parameters, we have to first define *coordination terms*, over which coordinations are defined.

A coordination term can be a constant, the result of some operations and a parameter that associated with some interaction. Since coordinations only correlate interaction names and parameters, variables are not included in coordination terms. The formal definition of terms is the same as Definition 5.4.1 from [17], and is shown as follows:

Definition 4.5.2 (Coordination term)**.** The D-indexed family of sets CTERM of Coordination Terms is defined as follows:

- *If $c \in F_d$ then*

$$c \in CTERM_d$$

for every $d \in D$

- *If $f \in F_{<d_1,\ldots,d_n,d_{n+1}>}$ and $\overrightarrow{p} \in CTERM_{<d_1,\ldots,d_n>}$ then*

$$f(\overrightarrow{p}) \in CTERM_{d_{n+1}}$$

for every $d_1,\ldots,d_n,d_{n+1} \in D$

- *If $a \in NAME$ and $p \in PARAM(a)_d$, then*

$$a.p \in CTERM_d$$

for every $d \in D$

For example, in figure 4.5 $S.i_1$ and $R.i_1$ are coordination terms.

The interpretation of coordination terms is adapted from Definition 5.4.6 from [17]. In our definition, coordination terms are interpreted over the states of SO-HL^{2}TSs, thus they are denoted in the form $\sigma(\zeta)$ ($\sigma \in \Sigma$ in m and $\zeta \in [0, r_\sigma]$). The interpretation of terms is shown as follows:

Definition 4.5.3 (Interpretation of coordination terms)**.** *The interpretation of a coordination term $T \in CTERM$ in a state $\sigma(\zeta)$ with $\sigma \in \Sigma$ and $0 \le \zeta \le r_\sigma$, written $[\![T]\!]_{\sigma(\zeta)}$,* is defined as follows, where $II(param) = \langle param', view\rangle$:

- $[\![c]\!]_{\sigma(\zeta)} = c_{\mathscr{U}}$
- $[\![f(T_1,\ldots,T_n)]\!]_{\sigma(\zeta)} = f_{\mathscr{U}}([\![T_1]\!]_{\sigma(\zeta)},\ldots,[\![T_n]\!]_{\sigma(\zeta)})$
- $[\![a.p]\!]_{\sigma(\zeta)} = view(II(map(a)).p'^{\Pi^{\sigma(\zeta)}})$

A coordination is defined as the equivalence of two interaction names, and a set of formulas of parameters that associate with the interactions. In particular, the operator $\equiv$ is used for specifying the equivalence of two interaction names, this is to say that a coordination defines a one-to-one relationship between two interaction names. The formal definition of terms is the same as Definition 5.4.2 from [17], and is shown as follows:

Definition 4.5.4 (Coordination)**.** A coordination COORD *is a set of elements of ϕ, where:*

- $\phi ::= t_1 = t_2 \mid a \equiv b$

with $t_1, t_2 \in CTERM_d$ for some $d \in D$ and $a \in NAME$ and $b \in NAME'$, such that:

- *For every $a \in NAME$ and $b, c \in NAME'$ if $a \equiv b \in COORD$ then $a \equiv c \notin COORD$;*
- *For every $a \in NAME'$ and $b, c \in NAME$ if $a \equiv b \in COORD$ then $a \equiv c \notin COORD$.*

For example, in figure 4.5 $S.i_1 = R.i_1$ is a formula of the coordination of interaction protocol *straight* $O(d_1, d_2)$.

A coordination need to be satisfied by a SO-HL^{2}TS, since it correlates two interactions, each of which can be mapped to an interaction declared in a role (a role can be a business role or a business protocol) by a signature morphism, and these two roles should be interpreted over the same SO-HL^{2}TS; and because of the correlation of two interactions, the coordination also needs to be satisfied by the interaction interpretations of the two interactions.

The satisfaction of coordinations is adapted from Definition 5.4.7 from [17]. In our definition, coordinations are satisfied by SO-HL^{2}TSs. The definition is shown as follows:

Definition 4.5.5 (Satisfaction of Coordinations)**.** *The satisfaction relation for coordinations by the pair* $\langle m, II \rangle$ *is defined as follows, where* Σ *is defined in m:*

- $\langle m, II \rangle \models t_1 = t_2$ *iff* $[\![t_1]\!]_{\sigma(\zeta)} = [\![t_2]\!]_{\sigma(\zeta)}$ *for every* $\sigma \in \Sigma$ *and* $\zeta \in [0, r_\sigma]$*;*
- $\langle m, II \rangle \models a \equiv b$ *iff* $II(map(a)) = II(map(b))$*.*

4.5.2 Interaction protocols and connectors

An interaction protocol consists of a a pair of interactions and an coordination that correlates the names of the interactions and the parameters that associate with the interactions. The pair of interactions in a business protocol are labeled with *Role A and Role B*. The formal definition of interaction protocols is the same as Definition 5.4.3 from [17], and is shown as follows:

Definition 4.5.6 (Interaction Protocol)**.** An Interaction Protocol *is a triple*

$$\langle sig_1, sig_2, coord \rangle$$

where coord is a coordination for the interaction signatures sig_1 *and* sig_2*. We refer to* sig_1 *and* sig_2 *as the roles of the interaction protocol* — Role A *and* Role B*, respectively.*

In Figure 4.5, the interaction protocol *straight* $O(d_1, d_2)$ correlates an interaction of type *snd* with an interactions of type *rcv*, and the associating event ⊠of each interactions has two parameter. S and R are the names of the interactions, and d_1 and d_2 are the data types of the parameters. They are left undefined until the interaction protocol is applied. The equivalence of the two interactions is specified in *coord*. An interaction protocol is satisfied by the tuple $\langle m, II \rangle$ if and only if the coordination of it is satisfied by $\langle m, II \rangle$.

Interaction protocols and signature morphisms are used to compose *connectors*. In SRML, connectors model wires. A connector links each pair of the interactions specified in a set of interaction protocols to the interaction signatures of a pair of roles with a pair of signature morphisms. The pair of roles model two components (a component can be a service component or a service interface) that are connected by the wire modeled by the connector in a service module. In this way, interaction protocols are linked to the actual interactions. A connector can contain one or several interaction protocols, depending on the number of the interactions it connects. The The formal definition of connectors is shown as follows:

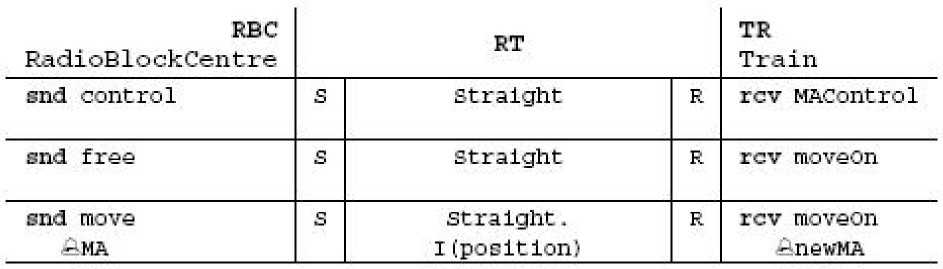
Definition 4.5.7 (Connector)**.** *A connector for two interaction signatures* $\langle NAME, PARAM\rangle$ *and* $\langle NAME', PARAM'\rangle$ *is a triple*

$$\langle map_1, ip, map_2\rangle$$

where:

- *ip is a set of interaction protocols;*
- map_1 *is a signature morphism that maps the interactions labeled with Role A in ip to* $\langle NAME, PARAM\rangle$*;*
- map_2 *is a signature morphism that maps the interactions labeled with Role B in ip to* $\langle NAME', PARAM'\rangle$*.*

Figure 4.6 shows the connector from Appendix A, that binds interactions in business protocol *RadioBlockCenter* to interactions in business role *Train* using three interaction protocols. Among these interaction protocols, we take *Straight.I (position)* for example. Interaction name *S* and variable $S.i_1$ of *Straight.I (position)* are mapped to interaction name *move* and parameter *move🔔.MA* respectively by the interaction morphism from Role A of *Straight.I (position)* to the interaction signature of *RadioBlockCenter*; interaction name *R* and variable $R.i_1$ of *Straight.I (position)* are mapped to interaction name *moveOn* and parameter *moveOn🔔.newMA* respectively by the interaction morphism from Role B of *Straight.I (position)* to the interaction signature of *Train*.



RBC RadioBlockCentre	RT			TR Train
snd control	S	Straight	R	rcv MAControl
snd free	S	Straight	R	rcv moveOn
snd move 🔔MA	S	Straight. I(position)	R	rcv moveOn 🔔newMA

Figure 4.6: The connectors that bind the business protocol *RBC* to the business role*TR*

Let partyA and partyB be two components in a service module, Figure 4.7 shows the structure of a connector binding the interaction signature of the SRML specification of *partyA* to the interaction signature of the SRML specification of *partyB*. The SRML specification of *PartyA* and the SRML specification of *partyB* are denoted by *spec(partyA)* and *spec(partyB)*.

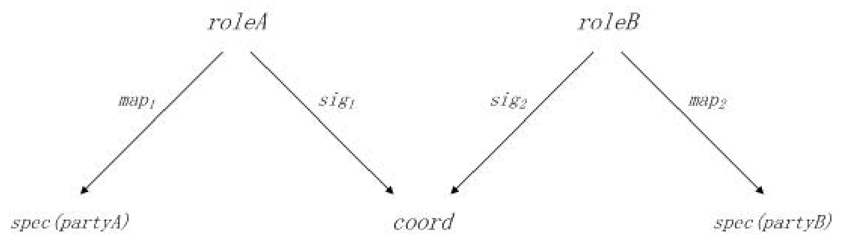


Figure 4.7: The composition of a connector

4.6 Formalization of service module

Based on the formal definition of business roles, business protocols, interaction protocols and connectors, service modules can be defined. In SRML, a service module models a service by formally specifying the compositions of a service. In this section, we formalize service modules and define the satisfaction of service modules.

As introduced in Chpater 3, a service module defines the service components, service interfaces and the wires connecting them in a service, it also defines how these compositions of the service are connected. In Figure 3.2 we showed the service module *Train-Control* from Appendix A.

A service module includes a graph that consists of the compositions of the service modeled by that service module, and a set of functions that assign these compositions with SRML specification. i.e. each service component is assigned with a business role, each service interface is assigned with a business protocol and each wire is assigned with an interaction protocol that is labeled by a connector. The formalization of service module is the same as Definition 5.5.1 from [17], and is shown as follows:

Definition 4.6.1 (Service module)**.** A service module is *a tuple*

$$\langle N, W, C, client, spec, prov \rangle$$

where:

- $\langle N, W \rangle$ *is a simple graph (undirected, without self-loops or multiple edges), where* N *is a set of nodes (the parties that compose the service and the client of the service) and the symmetric relation* $W \subseteq N \times N$ *is the set of edges (the wires that interconnect the parties and the client);*

- *We distinguish between different types of nodes:*

  – $client \in N$ *is the client of the service;*

  – $P = N \backslash \{client\}$ *are the nodes whose composition provides the service — called parties — and consists of:*

    ∗ $C \subseteq P$*, the components; and*

  - ∗ *$R = P\backslash C$, the (external) required services;*

- *spec assigns:*
  - *a business role to each component $c \in C$;*
  - *a business protocol to each required service $r \in R$; We use sign(p) to refer to the interaction signature (indirectly) assigned by spec to a party $p \in P$;*
  - *a connector for sign(p) and sign(p') to each internal wire $\langle p, p' \rangle \in WW$ such that every interaction name coordinated by that connector is not coordinated by any other connector in $spec(w)$;*

- *$prov = \langle\langle NAME, PARAM\rangle, BEHAVIOR\rangle$ is a business protocol — the provides-interface of the module — such that:*
  - *$NAME \subseteq \bigcup NAME^{sign(p)}$ with $p \in P$ and $\langle client, p\rangle \in W$;*
  - *$PARAM_{\#}(a) = PARAM_{\#}^{sign(p)}(a)$ for each $p \in P$, $a \in NAME \cap NAME^{sign(p)}$ and $\# \in \{$🔔, ✉, ✓, 💣, ✞$\}$;*

We use wire(a) to denote a wire $w \in W$ such that a is in the coordination of $spec(w)$.

Take service module *Train-Control*= $\langle N, W, C, client, spec, prov\rangle$ for example, $N = \langle ETCSC, MOC, RBC\rangle$, $C = \langle ETCSC\rangle$, $W = \langle\langle MOC, ETCSC\rangle, \langle ETCSC, RBC\rangle\rangle$, $client = MOC$, and $spec(MOC)$, $spec(RBC)$, $spec(ETCSC)$, $spec(\langle MOC, ETCSC\rangle)$ and $spec(\langle ETCSC, RBC\rangle)$ are specified as that in Appendix A.

Since the SRML specifications assigned by a service module are satisfied by different structures (such as interaction interpretations and SO-HL^{2}TSs), a service module is satisfied by a tuple of those structures. Such a tuple is called an *interpretation structure*. An interpretation structure of a service module consists of a configuration and an SO-HL^{2}TS for the whole SRML specifications of the service module, and an interaction interpretation and an attribute interpretation for the SRML specifications for each party (a service component or a service interface) of that service module.

The formal definition of interpretation structures is adapted from Definition 5.5.2 from [17]. In our definition, we use SO-HL^{2}TS instead of SO-TS. The definition is shown as follows:

Definition 4.6.2 (Interpretation structure)**.** An interpretation structure for a service module $M = \langle N, W, C, client, spec, prov\rangle$ is a tuple

$$\langle \Xi, II, m, \Delta\rangle$$

such that:

- $\Xi = \langle N', W', PLL, 2WAY, 1WAY\rangle$ is a configuration;
- m is a SO-HL^{2}TS for Ξ;

- II is a P-indexed family of interaction interpretations over $2WAY \cup 1WAY$ such that II_p, with $p \in P$, interprets the signature in spec(p), i.e. it interprets the interactions of party p;
- Δ is a C-indexed family of attribute interpretations over m, such that Δ_c interprets the attribute declaration in spec(c), i.e. it interprets the attributes of component c.

Let $prov = \langle\langle NAME, PARAM\rangle, BEHAVIOR\rangle$. We use II^{prov} to denote $II|_{NAME \cup PARAM}$, i.e. the interpretation of the interactions in the provides-interface.

A service module can be satisfied by an interpretation structure. An interpretation structure satisfies a service module if and only if it satisfies the configuration of the service module and the SRML specification of each party of the service module. The formal definition of satisfaction of a service module is adapted from Definition 5.5.3 from [17]. In our definition, we use the interaction structure defined in Definition 4.6.2. The definition is shown as follows:

Definition 4.6.3 (Satisfaction of a service module)**.** *Given a service module specification* $M = \langle N, W, C, client, spec, prov\rangle$ *and a configuration* $\Xi = \langle N', W', PLL, 2WAY, 1WAY\rangle$*, an interpretation structure* $sem = \langle \Xi, II, m, \Delta\rangle$ *is said to satisfy the composition of M, written* $sem \models M$ *iff:*

- $N = N'$*,* $W = W'$ *and* $PLL = R \cup \{client\}$*;*
- *For each party* $p \in P$*,* II_p *is local to p;*
- *For each party* $p \in P$ *and interaction name* $a \in NAME^{sign(p)} \cap NAME^{sign(prov)}$*,* $II(a) \in INT_{\langle p, client\rangle} \cup INT_{\langle client, p\rangle}$*;*
- $\langle m, \Delta\rangle$ *satisfies spec(p), for each party* $p \in P$*;*
- $\langle m, II\rangle$ *satisfies spec(w), for each wire* $w \in W$*.*

e.g. given a service module M, Ξ is the configuration of M, sig is an interaction signature of M, II is an interaction interpretation for sig, m is a SO-HL^{2}TS for Ξ and m satisfies all the business roles and business protocols specified by M, Δ is an attribute interpretation over m; then there is $\langle \Xi, II, m, \Delta\rangle \models M$.

Chapter 5

A method for verifying the extended SRML behavior constraints

In this chapter, we provide a method for verifying hybrid behaviors that are business protocols (as that defined in Section 4.4.3). This method includes the transformation from SRML hybrid behavior constraints to dTL formulas and the verification of those dTL formulas. The transformation is realized by translating SRML based finite automata of a given service module to the corresponding regular expression for transitions using Brzozowski's method [44], and mapping the resulted regular expression to the hybrid program which is a composition of the dTL formulas to be verified. This approach and the related definitions are illustrated in Section 5.1. The verification of SRML hybrid behaviors is done by verifying the corresponding dTL formulas with a set of sequent calculus adopted from [41]. The sequent calculus and explanation for operators and terms of the calculus are presented in Section 5.2. In Section 5.3, we make a case study of a small part of the European Train Control System by modeling it with a SRML service module and verifying a hybrid behavior constraint of the service module using the method introduced in Section 5.1 and Section 5.2. Part of the case study and verification can also be found in [28].

5.1 SRML based finite automata and regular expressions for transitions

Given a SRML service module M with the hybrid behavior *"**always** sp"* as that defined in Section 4.4.3, in order to be able to verify $[\beta]\Box sp$ formally (here β is an unknown hybrid program), we need to obtain the hybrid program β which represents the behaviors of M. This approach can be done in the following steps:

1. Construct a SRML based finite automaton for the SRML service module M (in this step, when given a SO-HL^{2}TS m and a SRML service module M which is interpreted over m, the SRML based finite automaton for M could be constructed according to Definition 5.1.1 over paths of m and the SRML specifications of M);

2. Transform the SRML based finite automaton to regular expressions for transition using Brzozowski's method;

3. Map the regular expression for transitions to hybrid program.

The hybrid program obtained from the steps above is the hybrid program β in the hybrid behavior constraint $[\beta]\Box s$. In order to show the validity of this approach, we give some definitions, mapping rules and theorems which are essential to the approach.

Definition 5.1.1 (SRML based automata)**.** Given a service module M, a set of transition names TR and an interpretation structure $sem = \langle \Xi, II, m, \Delta, TR\rangle$ such that $sem \models M$, where $m = \langle S, s_0, \Sigma, \sigma_0, Act, R, AP, L, TIME, \Pi\rangle$. A SRML based automaton of M, written D_M, is a tuple

$$\langle Q, \overline{Q}, \sigma_0(0)\ldots\sigma_0(r_{\sigma_0}), TR, \delta, F\rangle$$

where:

- Q is the set of all the traces of states in the paths of m, such that for every $\sigma \in \Sigma$, $\sigma(0)\ldots\sigma(r_\sigma) \in Q$;
- $\overline{Q} = \{\sigma(0)\ldots\sigma(r_\sigma) | \sigma(0)\ldots\sigma(r_\sigma) \in Q$, and $\forall \sigma'(0)\ldots\sigma'(r_{\sigma'}) \in [\sigma(0)\ldots\sigma(r_\sigma)]$ there is $TIME^{\sigma(0)} \leq TIME^{\sigma'(0)}, \ldots, TIME^{\sigma(r_\sigma)} \leq TIME^{\sigma'(r_{\sigma'})}\}$;
- $\sigma_0(0)\ldots\sigma_0(r_{\sigma_0})$ is the initial sequence of states;
- TR is the set of transition names;
- δ is a transition function that takes any $\sigma(0)\ldots\sigma(r_\sigma) \in \overline{Q}$ and a $tr \in TR$ as arguments, and returns a $\sigma'(0)\ldots\sigma'(r_{\sigma'}) \in \overline{Q}$ such that: $\delta(\sigma(0)\ldots\sigma(r_\sigma), tr) = \sigma'(0)\ldots\sigma'(r_{\sigma'})$ *iff* there exists an $\alpha \in 2^{Act}$ and a $\sigma''(0)\ldots\sigma''(r_{\sigma''}) \in Q$ such that $(\sigma(r_\sigma), \alpha, \sigma''(0)) \in R$, $\sigma''(0)\ldots\sigma''(r_{\sigma''}) \in [\sigma'(0)\ldots\sigma'(r_\sigma)]$, $II(trigger(tr)) \in \alpha$ and $\sigma(r_\sigma) \models guard(tr)$;
- $F \subseteq \overline{Q}$ is the set of accepting traces of states such that:
  - If there exist a trace of states $\sigma(0)\ldots\sigma(r_\sigma) \in \overline{Q}$ and an $\alpha \in 2^{Act}$ such that $(\sigma(r_\sigma), \alpha, \sigma_0(0)) \in R$, then $\sigma_0(0)\ldots\sigma_0(r_{\sigma_0}) \in F$;
  - For every $\sigma(0)\ldots\sigma(r_\sigma) \in \overline{Q}$, if there doesn't exist an $\alpha \in 2^{Act}$ and a $\sigma'(0)\ldots\sigma'(r_{\sigma'}) \in \overline{Q}$ such that $(\sigma(r_\sigma), \alpha, \sigma'(0)) \in R$, then $\sigma(0)\ldots\sigma(r_{\sigma_0}) \in F$.

If $\overline{Q}$ is finite, the SRML based automaton is called SRML based finite automaton.

In Definition 5.1.1, Q is a different set from Σ. Σ is the set of functions that are specified in the HL^{2}TS m, while Q is the set of traces of states that are obtained by applying the functions in Σ. Definition 5.1.1 is adapted from the definition of Deterministic Finite Automata (DFA) [45, 46, 47], this is because in our example a transition from a source state can only result in one target state. For a more general case, the definition can be adapted from Nondeterministic Finite Automata (NFA) [48].

Definition 5.1.2 (Transition Diagrams for SRML based finite automata)**.** A transition diagram for a SRML based finite automaton $D_M = \langle Q, \overline{Q}, \sigma_0(0)\dots\sigma_0(r_{\sigma_0}), TR, \delta, F\rangle$, which is defined over the SO-HL^{2}TS m is a graph defined as follows:

- There is a node for each $\sigma(0)\dots\sigma(r_\sigma) \in \overline{Q}$;
- For every $tr \in TR$, if $\delta(\sigma(0)\dots\sigma(r_\sigma), tr) = \sigma'(0)\dots\sigma'(r_\sigma)$, then there is an arc from node $\sigma(0)\dots\sigma(r_\sigma)$ to node $\sigma'(0)\dots\sigma'(r_\sigma)$, labeled tr;
- Nodes corresponding to the elements in set F are marked with double circles, the rest of the nodes are marked with single circles.

For example, the SRML based finite automaton $D_{Train-Control}$ for service module *Train-Control* (specified in Appendix A) is defined as follows:

$$D_{Train-Control} = \langle Q, \overline{Q}\sigma_0(0)\dots\sigma_0(3), TR, \delta, F\rangle \tag{5.1}$$

where:

- $\overline{Q} = \{\sigma_0(0)\dots\sigma_0(3), \sigma_1(0)\dots\sigma_1(5), \sigma_2(0)\dots\sigma_2(4), \sigma_3(0)\dots\sigma_3(2)\}$;
- $TR = \{Cont, Nego, Corr, Stop\}$
- $\delta(\sigma_0(0)\dots\sigma_0(3), Cont) = \sigma_0(0)\dots\sigma_0(3)$;
- $\delta(\sigma_0(0)\dots\sigma_0(3), Nego) = \sigma_1(0)\dots\sigma_1(5)$;
- $\delta(\sigma_1(0)\dots\sigma_1(5), Corr) = \sigma_2(0)\dots\sigma_2(4)$;
- $\delta(\sigma_2(0)\dots\sigma_2(4), Stop) = \sigma_3(0)\dots\sigma_3(2)$;
- $F = \{\sigma_0(0)\dots\sigma_0(3), \sigma_3(0)\dots\sigma_3(2)\}$.

Service module *Train-Control* is defined over the SO-HL^{2}TS $m = \langle S, s_0, \Sigma, Act, R, AP, L, TIME, \Pi\rangle$,where $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_0', \sigma_1', \sigma_2', \sigma_3', \sigma_0'', \sigma_1'', \sigma_2'', \sigma_3'', \dots\}$ with $\sigma_0 : [0,3] \to S$, $\sigma_1 : [0,5] \to S$, $\sigma_2 : [0,4] \to S$, $\sigma_3 : [0,2] \to S$. The number 3, 5, 4, and 2 are chosen arbitrarily for the possible durations of σ_0, σ_1, σ_2 and σ_3. The transition diagram of $D_{Train-Control}$ is shown in Figure 5.1. Notice that in Figure 5.1, each node represents a trace of states but not a single state, this is because we wish to use finite automata to represent the evolutions of hybrid systems. In such transition diagrams for SRML based finite automata, if there is an arch from a source node to a target node, it means that a transition occurs from the last state of the source node to the first state of the target node.

Next we define regular expressions for transitions which is a bridge for transforming SRML based finite automata to hybrid programs.

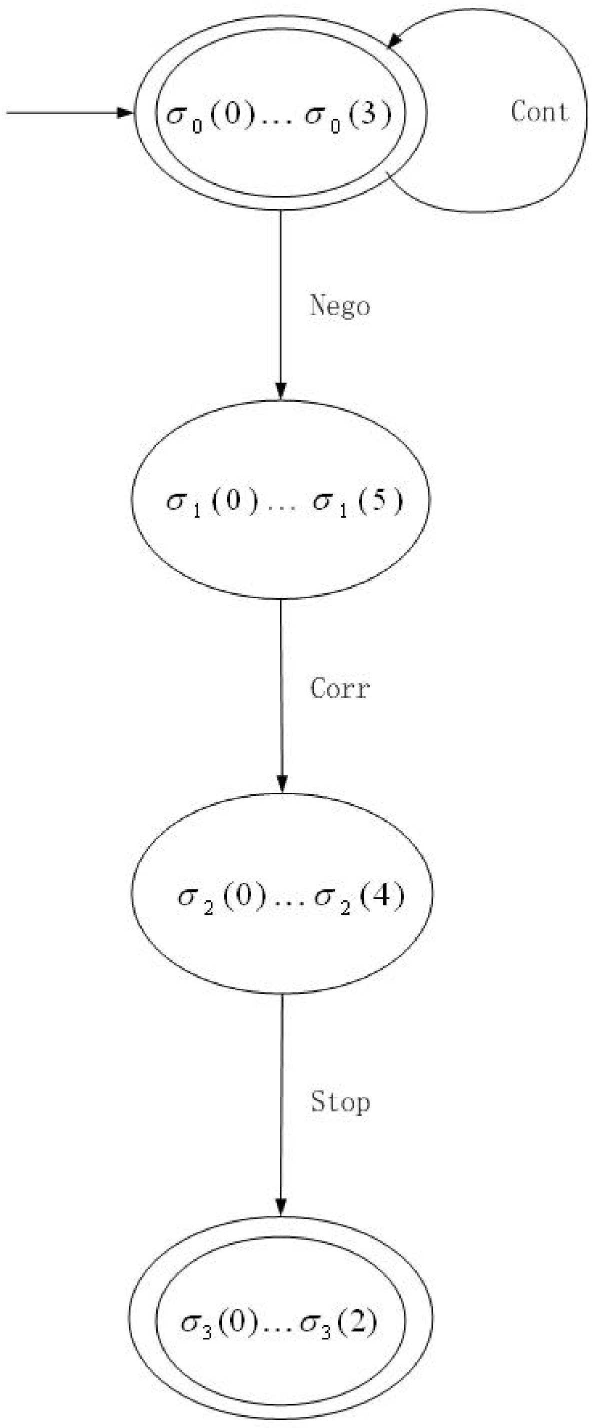

$\sigma_0(0)\ldots\sigma_0(3)$
Cont
Nego
$\sigma_1(0)\ldots\sigma_1(5)$
Corr
$\sigma_2(0)\ldots\sigma_2(4)$
Stop
$\sigma_3(0)\ldots\sigma_3(2)$

Figure 5.1: Transition diagrams for $D_{Train-Control}$

Definition 5.1.3 (Sequence of transition names)**.** If TR is a set of transition names, $n \in \mathbb{N}$ and $tr_i \in TR$ for every $1 \leq i \leq n$, then $w = tr_1; \ldots ; tr_n$ is a sequence of transition names. The length of w is denoted by $|w|$, and $|w| = n$. An empty sequence of transition names is denoted by ε, and $|\varepsilon| = 0$.

For example, given the set of transition names $TR = \{Cont, Nego, Corr, Stop\}$, $w = Nego; Corr; Stop$ is a sequence of transition names and $|w| = 3$.

Definition 5.1.4 (Regular expressions for transitions)**.** Given a set of transition names TR, the set of regular expressions for transitions of TR, denoted by RE_{TR}, is defined as follows:

- If $tr \in TR$, then RE_{TR};
- $\varepsilon \in RE_{TR}$;
- $\emptyset \in RE_{TR}$, where $\emptyset$ is the empty set;
- If $E \in RE_{TR}$ and $F \in RE_{TR}$, then $E + F \in RE_{TR}$;
- If $E \in RE_{TR}$ and $F \in RE_{TR}$, then $E; F \in RE_{TR}$;
- If $E \in RE_{TR}$, then $E^* \in RE_{TR}$.

For example, given the set of transition names $TR = \{Cont, Nego, Corr, Stop\}$, $Cont^*; (Cont + Nego; Corr; Stop)$ is a regular expression for transitions of TR.

In order to show the equivalence of SRML based finite automata and regular expressions for transitions on the input symbols, we need to define the language for SRML based finite automata and the language for regular expressions for transitions:

Definition 5.1.5 (The extension of transition functions)**.** For a SRML based finite automaton $D_M = \langle Q, \overline{Q}, \sigma_0(0) \ldots \sigma_0(r_{\sigma_0}), TR, \delta, F \rangle$, the extension of transition function δ, denoted by $\hat{\delta}$, is defined inductively as follows:

- $\hat{\delta}(\sigma(0) \ldots \sigma(r_\sigma), \varepsilon) = \{\sigma(0) \ldots \sigma(r_\sigma)\}$ for every $\sigma(0) \ldots \sigma(r_\sigma) \in \overline{Q}$;
- If $w = tr_1; \ldots ; tr_n$ is a sequence of transition names where $n \in \mathbb{N}$ and $tr_i \in TR$ for every $1 \leq i \leq n$, then for every $\sigma(0) \ldots \sigma(r_{\sigma_0}) \in \overline{Q}$, there is:

$$\begin{aligned}
&\hat{\delta}(\sigma(0) \ldots \sigma(r_\sigma), w) = \delta(\sigma'(0) \ldots \sigma'(r_{\sigma'}), tr_n) \\
&\text{ where } \sigma'(0) \ldots \sigma'(r_{\sigma'}) = \hat{\delta}(\sigma(0) \ldots \sigma(r_\sigma), w'), w' = tr_1; \ldots ; tr_{n-1}, \\
&\delta(\sigma'(0) \ldots \sigma'(r_{\sigma'}), tr_n) \text{ exists and } \hat{\delta}(\sigma(0) \ldots \sigma(r_\sigma), w') \text{ exists .}
\end{aligned}$$

Take the SRML based finite automaton $D_{Train-Control}$ defined in Formula 5.1 for example, $\hat{\delta}$ is the extension for transition function δ and $\hat{\delta}(\sigma_0(0) \ldots \sigma_0(3), Nego; Corr; Stop) = \sigma_3(0) \ldots \sigma_3(2)$.

Definition 5.1.6 (The language of SRML based finite automata)**.** If $D_M = \langle Q, \overline{Q}, \sigma_0(0)\ldots\sigma_0(r_{\sigma_0}), TR, \delta, F\rangle$, then the language of D_M, denoted by $L(D_M)$, is defined as:

$$L(D_M) = \{w|\hat{\delta}(\sigma_0(0)\ldots\sigma_0(r_{\sigma_0}), w) \in F\}$$

For example, $L(D_{Train-Control}) = \{Cont, Cont;Cont, Cont;Cont;Cont, \ldots, Nego;Corr;Stop, Cont;Nego;Corr;Stop, Cont;Cont;Nego;Corr;Stop, \ldots\}$.

Definition 5.1.7 (The language of regular expressions for transitions)**.** Given a set of regular expressions for transitions RE, the language of a regular expression $E \in RE$, denoted by $L(E)$, is defined inductively defined as follows:

- If tr is a transition name, $L(tr) = \{tr\}$;
- $L(\varepsilon) = \{\varepsilon\}$;
- $L(\emptyset) = \emptyset$;
- $L(E+F) = L(E) \cup L(F)$;
- $L(E;F) = \{tr_1;\ldots;tr_m;tr'_1;\ldots;tr'_n | tr_1;\ldots;tr_m \in L(E)$, $tr'_1;\ldots;tr'_n \in L(F)$ and $m,n \in \mathbb{N}\}$;
- $L(E^*) = (L(E))^* = \bigcup_{n\in\mathbb{N}} L(E)^n$, where $L(E)^0 = \{\varepsilon\}$ and $L(E)^n = L(E^{n-1};E)$.

For example, $L(Cont^*;(Cont+Nego;Corr;Stop)) = \{Cont, Cont;Cont, Cont;Cont;Cont, \ldots, Nego;Corr;Stop, Cont;Nego;Corr;Stop, Cont;Cont;Nego;Corr;Stop, \ldots\}$.

Then we show the equivalence between the language of SRML based finite automata and the language of regular expression for transitions:

Theorem 5.1.1. If D_M is a SRML based finite automaton , then there exists a regular expression for transitions E such that $L(D_M) = L(E)$.

Proof. The equivalence between the language of DFA and the language of regular expressions is proved in [49] by constructing regular expression for some given DFA inductively. Since SRML based finite automata can be seen as a type of DFA and regular expression for transitions can be seen as a type of regular expression, Theorem 5.1.1 can be proved in the same way as that in [49]. □

Moreover, E can be obtained using Brzozowski's method.

Given a SRML based finite automaton D_M, we denote the regular expression for transitions with E_M if $L(D_M)$ equals to the language of that regular expression for transitions. In order to obtain the hybrid program which then composites the hybrid behavior constraint of SRML service module M, we need to convert E_M to hybrid programs. The function for converting is defined as follows:

Definition 5.1.8 (Hybrid programs for regular expressions for transitions)**.** Given a set of transition names TR and a SRML service module M which is defined over TR, the hybrid program for a regular expression for transitions $E \in RE_{TR}$ is defined inductively as follows:

- If $E \in TR$, then $hp(E) = eff(E)$ (*eff* is the function of effect specified in M);
- If $E = \varepsilon$, then $hp(E) = \chi$;
- If $E = \emptyset$, then $hp(E) = \beta$ where β is an arbitrary hybrid program whose trace semantics $[\![\beta]\!] = \emptyset$;
- If $E, F \in RE_{TR}$, then $hp(E+F) = hp(E) \cup hp(F)$;
- If $E, F \in RE_{TR}$, then $hp(E;F) = hp(E);hp(F)$;
- If $E \in RE_{TR}$, then $hp(E^*) = (hp(E))^*$.

Note that a hybrid program with empty trace semantics can arise if M is inconsistent and thus not satisfied by any transition system.

For example, given the set of transition names $TR = \{Cont, Nego, Corr, Stop\}$ and SRML service module $Train-Control$ whose specification is in Appendix A, $Cont^*;(Cont+Nego;Corr;Stop)$ is a regular expression for transitions of TR, the hybrid program for $Cont^*;(Cont+Nego;Corr;Stop)$, obtained by $hp(Cont^*;(Cont+Nego;Corr;Stop))$, is:

$$\begin{aligned}
&(C_{dot} = v_0)^*;(C_{dot} = v_0 \cup \\
&C_{dot} = v_1; C_{dot} = V \wedge V_{dot} = b; \\
&end\boxtimes.currPos = C - C_0 \wedge end\boxtimes.currTime = t' - t).
\end{aligned} \tag{5.2}$$

Given a SRML service module M defined over a set of transition names TR, and a SO-HL^{2}TS m over which the business roles of M are interpreted, D_M is the SRML based finite automaton for M. We then show that $[\![hp(E_M)]\!]$ is the set of all the paths of m selected by M (E_M is the regular expression for transitions such that $L(D_M) = L(E_M)$):

Theorem 5.1.2. If M is a SRML service module which is defined over a set of transition names TR, and the business roles of M are interpreted over a SO-HL^{2}TS m, D_M is the SRML based finite automaton for M and E_M is the regular expression for transitions such that $L(D_M) = L(E_M)$. Then $[\![hp(E_M)]\!]$ is the set of all the paths of m selected by M.

Proof. From Definition 4.4.6 and Definition 5.1.7 we can obtain $[\![hp(E_M)]\!] = \{hp(E) | E \in L(E_M)\}$. Since $L(D_M) = L(E_M)$ (the existence of E_M is proved in Theorem 5.1.1), we have $[\![hp(E_M)]\!] = \{hp(E) | E \in L(D_M)\}$. From Definition 5.1.6, it is obvious that $\{hp(E) | E \in L(D_M)\}$ is the set of all the paths of m selected by M. Thus $[\![hp(E_M)]\!]$ is the set of all the paths of m selected by M.

□

5.2 Verification of behavior constraints

Given a service module $M = \langle N, W, C, client, spec, prov\rangle$ where $prov = \langle\langle NAME, PARAM\rangle,$ $BEHAVIOR\rangle$, for every behavior $x \in BEHAVIOR$, the verification can be divided into two parts:

- If x is a behavior specified by a normal behavior constraint, it can be verified by UMC model-checker [50] following the procedure provided in [17];
- If x is a behavior specified by the hybrid behavior constraint, it can be verified manually with a set of rule schemata of dTL verification calculus adapted from [41].

Since in this thesis we mainly focus on the continuous behavior of a transition system, only the rule schemata of dTL verification calculus shown in table 5.1 is introduced here.

A sequent calculus is of the form $\Gamma \vdash \Delta$, where Γ and Δ are finite sets of formulas. Its semantics is $\bigwedge_{\phi \in \Gamma} \phi \rightarrow \bigvee_{\psi \in \Delta} \psi$. In these rules, ϕ and ψ are state formulas and π is a trace formula; β and γ are hybrid programs; $T, T_1, \ldots, T_n, T', T'_1, \ldots, T'_n$ are state terms, $v, \dot{v} \in VAR$ are also state terms and $t \in ETERM_{time}$. Particularly rules P1–P9 are standard rules for propositional logic. P10 is a special case of formula $Cl_\forall(F_0 \rightarrow G_0) \rightarrow Cl_\forall(F \rightarrow G)$, which is an instance of a first-order tautology of real arithmetic. In this formula, F and G are first order formulas, F_0 is the result of substituting some variables in F with certain constants and G_0 is the result of substituting some variables in G with certain constants, and $Cl_\forall$ is the universal closure. Rules D1–D8 deals with dTL state formulas, which have no temporal operator and are similar to dynamic logic formulas in [38]. In rule D3, F is a first-order formula and the semantics of $F_{T_1,\ldots,T_n}^{T'_1,\ldots,T'_n}$ is to replace $T_1, \ldots, T_n$ with $T'_1, \ldots, T'_n$ in F. In rules D7 and D8, $v = f_{v_0}(t)$ is the solution of differential equation $\dot{v} = T$ with initial condition $f(t_0) = v_0$. Rules T1–T8 deal with dTL trace formulas, which have the temporal operators $\Box$ and $\Diamond$.

5.3 Case study of European Train Control System

Our case study is a small part of the European Train Control System (ETCS), whose scenario is based on a model of the ETCS presented in [51, 41]. In this section, we show the structure of our ETCS model, assign a SRML module to it and verify a behavior specified with the hybrid behavior constraint with rule schemata of dTL calculus introduced in Section 5.2.

5.3.1 An overview of the Train Control System

Figure 5.2 shows a network architecture of a control system which is proposed in [52]. The regulatory control area contains many sensors which send real-time data of certain objects to operator workstations in the supervisory control area. Operator workstations control the movement of the objects according to the messages that are received by them, and also send feedbacks. Corporation workplace in the scheduling control area decides the order of the objects' movement according to the messages received from the operator workstations. For example it issues moving authority to the objects by sending messages to operator workstations.

(P1) $\dfrac{\vdash\ \phi}{\neg\phi\ \vdash}$	(P2) $\dfrac{\phi\ \vdash}{\vdash\ \neg\phi}$	(P3) $\dfrac{\phi\ \vdash\ \psi}{\vdash\ \phi \rightarrow \psi}$
(P4) $\dfrac{\phi, \psi\ \vdash}{\phi \wedge \psi\ \vdash}$	(P5) $\dfrac{\vdash\ \phi \quad \vdash\ \psi}{\vdash\ \phi \wedge \psi}$	(P6) $\dfrac{\vdash\ \phi \quad \psi\ \vdash}{\phi \rightarrow \psi\ \vdash}$
(P7) $\dfrac{\phi\ \vdash \quad \psi\ \vdash}{\phi \vee \psi \vdash}$	(P8) $\dfrac{\vdash\ \phi, \psi}{\vdash\ \phi \vee \psi}$	(P9) $\dfrac{}{\phi\ \vdash\ \phi}$
(P10) $\dfrac{F_0\ \vdash\ G_0}{F\ \vdash\ G}$		
(D1) $\dfrac{[\beta]\pi \wedge [\gamma]\pi}{[\beta \cup \gamma]\pi}$	(D2) $\dfrac{\langle\beta\rangle\pi \vee \langle\gamma\rangle\pi}{\langle\beta \cup \gamma\rangle\pi}$	
(D3) $\dfrac{F^{T_1',\ldots,T_n'}_{T_1,\ldots,T_n}}{[\!\langle T_1 = T_1' \wedge \ldots \wedge T_n = T_n' \rangle\!]F}$	(D4) $\dfrac{[\!\langle\beta\rangle\!][\!\langle\gamma\rangle\!]\phi}{[\!\langle\beta;\gamma\rangle\!]\phi}$	
(D5) $\dfrac{\phi \wedge [\beta;\beta^*]\phi}{[\beta^*]\phi}$	(D6) $\dfrac{\phi \vee \langle\beta;\beta^*\rangle\phi}{\langle\beta^*\rangle\phi}$	
(D7) $\dfrac{\forall t \geq t_0[v = f_{v_0}(t)]}{[\dot{v} = T]\phi}$	(D8) $\dfrac{\exists t \geq t_0\langle v = f_{v_0}(t)\rangle\phi}{\langle\dot{v} = T\rangle\phi}$	
(T1) $\dfrac{[\beta]\Box\phi \wedge [\beta][\gamma]\Box\phi}{[\beta;\gamma]\Box\phi}$	(T2) $\dfrac{\langle\beta\rangle\Diamond\phi \vee \langle\beta\rangle\langle\gamma\rangle\Diamond\phi}{\langle\beta;\gamma\rangle\Diamond\phi}$	
(T3) $\dfrac{\phi \vee [T = T']\phi}{[T = T']\Box\phi}$	(T4) $\dfrac{\phi \wedge \langle T = T'\rangle\phi}{\langle T = T'\rangle\Diamond\phi}$	
(T5) $\dfrac{[\dot{v} = T]\phi}{[\dot{v} = T]\Box\phi}$	(T6) $\dfrac{\langle\dot{v} = T\rangle\phi}{\langle\dot{v} = T\rangle\Diamond\phi}$	
(T7) $\dfrac{[\beta]\Box\phi}{[\beta^*]\Box\phi}$	(T8) $\dfrac{\langle\beta\rangle\Diamond\phi}{\langle\beta^*\rangle\Diamond\phi}$	

Table 5.1: Rule schemata of the temporal dynamic dTL verification calculus

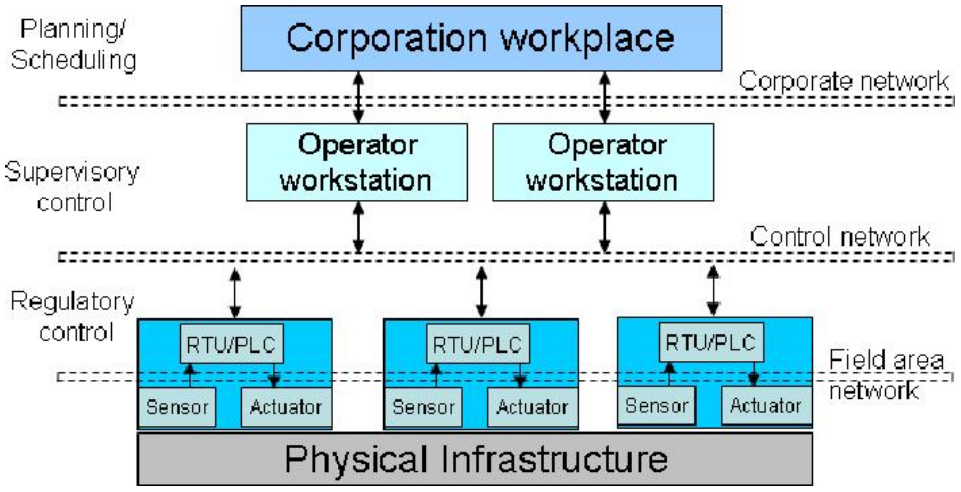


Figure 5.2: Network architecture of a control system

ETCS is part of the European Rail Traffic Management System (ERTMS) that controls each train within the network. It is an instance of the control system shown in Figure 5.2. Our model is of ETCS level 3. The object being observed and controlled is a moving train. The Balises on the railway work as sensors; the operator workstation which include a ETCS computer and a Monitoring Center, is installed inside the train; the corporation workplace which include a Radio Block Center(RBC), is in a remote control center and communicates with the train via radio networks; see Figure 5.3. The Global System for Mobile Communications-Railway (GSM-R) is a closed mobile phone network for the railways. It is the bearer system for messages between the ETCS computer and the Radio Block Center(RBC). The RBC is a centralized safety unit that receives train position information and sends movement authorities to trains via GSM-R.

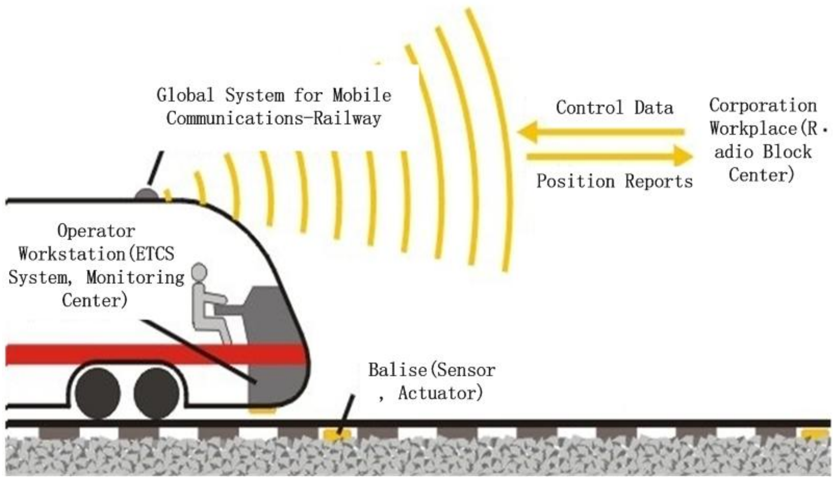

Global System for Mobile Communications-Railway
Control Data
Position Reports
Corporation Workplace(Radio Block Center)
Operator Workstation(ETCS System, Monitoring Center)
Balise(Sensor, Actuator)

Figure 5.3: Model of ETCS Level 3

5.3.2 The specification and verification of service module *Train-Control*

Figure 5.3 shows a simple model of ETCS Level 3 that is adapted form the models of ETCS in Wiki. As that shown in Figure 5.3, when observing the movement of the train from the operator workstation inside the cabin, the integration of *ETCSCenter*, *RadioBlockCenter* and *MonitoringCenter* can be seen as a hybrid system, in which *RadioBlockCenter* and *MonitoringCenter* only perform discrete changes, and *ETCSCenter* performs both discrete changes and continuous processes. In the SRML service module *Train-Control* which abstracts this hybrid system in a service-oriented style (shown in Figure 5.4), *ETCSCenter* is abstracted as a service component, *RadioBlockCenter* and *MonitoringCenter* are abstracted as service interfaces.

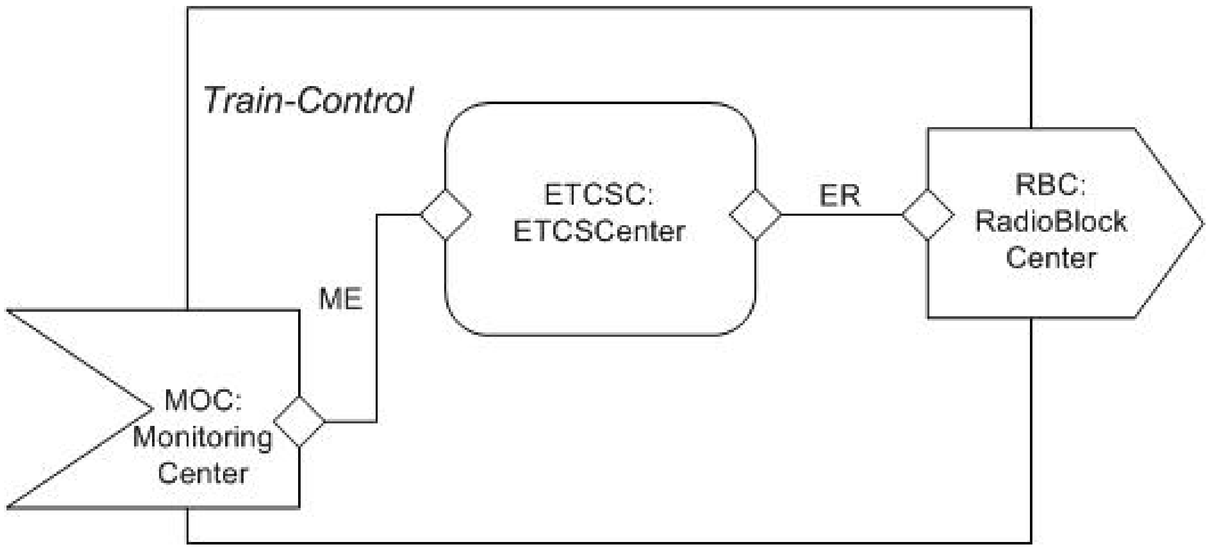


Figure 5.4: SRML service module *Train-Control*

Trains controlled by *ETCSCenters* are coordinated by decentralized *RadioBlockCenter*, which grants or denies moving authorities (MA) to each individual train by the GSM-R. In emergencies, trains always have to stop within the MA issued by the *RadioBlockCenter*. Each train negotiates with the *RadioBlockCenter* to extend its MA when approaching to the end of the MA. This process can be divided into several stages: a) a *Far* stage in which the train is far enough form the end of its current MA and is moving with a normal constant speed; b) a *Negotiation* stage in which the train has a certain distance to the end of its current MA, and is moving with a constant speed lower than the normal speed; c) a *Correction* stage in which the train has a certain distance which is near enough to the end of its current MA, and decelerates with a certain rates; d) a *Stop* stage in which the train breaks to stillness and have no further movement. The *ETCSCenter* changes the movement of the train in different stages on receiving different messages from the *RadioBlockCenter*. When the train enters the Stop stage, the *ETCSCenter* send the time duration of the train's movement and its current displacement to the *MonitoringCenter*, where one can check whether the displacement of the train is within its current MA.

According to the scenario introduced above, we assign a SRML specification to the service module Train-Control that is shown in Figure 5.4 such that Train-Control= $\langle N, W, client, spec, prov\rangle$, where:

- $N = \{MOC, ETCSC, RBC\}$;
- $W = \{\langle MOC, ETCSC\rangle, \langle ETCSC, RBC\rangle\}$;
- Node *ETCSC* is the service component that coordinates the movement process of the train, assigned with business role *ETCSCenter*;
- Node *RBC* is the service interface that requires service provided for knowing the current position of the train and issuing movement authority, assigned with business protocol *RadioBlockCentre*;
- Node *MOC* is the service interface that provides service to client of the service. The client gets the current positioning signal from the service, assigned with business protocol *MonitoringCernter*;
- ME, ER represent wires $\langle MOC, ETCSC\rangle$ and $\langle ETCSC, RBC\rangle$ that make the partner relationship between *MOC* and *ETCSC*, *ETCSC* and *RBC* explicitly.
- $prov(MOC), spec(RBC), spec(ETCSC), spec(\langle MOC, ETCSC\rangle)$ and $spec(\langle ETCSC, RBC\rangle)$ are specified as that in Appendix A.

Suppose $\Xi = \langle N, W, PLL, \Psi, 2WAY, 1WAY\rangle$ to be a configuration, $TR = \{Nego, Corr, Cont\}$ to be a set of transition names defined over Ξ, $sig = \langle NAME, PARAM\rangle$ to be an interaction signature, II to be an interaction interpretation for sig over $2WAY \cup 1WAY$ local to some node $n \in N$, Δ to be an attribute interpretation over m, $m = \langle S, s_0, \Sigma, Act, R, AP, L, TIME, \Pi\rangle$ to be a SO-HL^{2}TS for Ξ. $sem = \langle \Xi, II, m, \Delta\rangle$ is an interpretation structure for service module *Train-Control* such that $sem \models$*Train-Control* and:

- $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma'_0, \sigma'_1, \sigma'_2, \sigma'_3, \sigma''_0, \sigma''_1, \sigma''_2, \sigma''_3, \ldots\}$ with
  - $\sigma_0 : [0,3] \to S$, $\sigma'_0 : [0,3] \to S$, ..., $\sigma_1 : [0,5] \to S$, $\sigma'_1 : [0,5] \to S$, ..., $\sigma_2 : [0,4] \to S$, $\sigma'_2 : [0,4] \to S$, ..., $\sigma_3 : [0,2] \to S$, $\sigma'_3 : [0,2] \to S$, ...;
  - $[\sigma_0(0)\ldots\sigma_0(3)] = \{\sigma_0(0)\ldots\sigma_0(3), \sigma'_0(0)\ldots\sigma'_0(3), \sigma''_0(0)\ldots\sigma''_0(3), \ldots\}$;
  - $[\sigma_1(0)\ldots\sigma_1(5)] = \{\sigma_1(0)\ldots\sigma_1(5), \sigma'_1(0)\ldots\sigma'_1(5), \sigma''_1(0)\ldots\sigma''_1(5), \ldots\}$;
  - $[\sigma_2(0)\ldots\sigma_2(4)] = \{\sigma_2(0)\ldots\sigma_2(4), \sigma'_2(0)\ldots\sigma'_2(4), \sigma''_2(0)\ldots\sigma''_2(4), \ldots\}$;
  - $[\sigma_3(0)\ldots\sigma_3(2)] = \{\sigma_3(0)\ldots\sigma_3(2), \sigma'_3(0)\ldots\sigma'_3(2), \sigma''_3(0)\ldots\sigma''_3(2), \ldots\}$;
- $(\sigma_0(3), \alpha_1, \sigma_1(0)) \in R$ and $\alpha_1 \in Act$;
- $(\sigma_0(3), \alpha_2, \sigma'_0(0)) \in R$ and $\alpha_2 \in Act$;

- $(\sigma_1(5), \alpha_3, \sigma_2(0)) \in R$ and $\alpha_3 \in Act$;
- $(\sigma_2(4), \alpha_4, \sigma_3(0)) \in R$ and $\alpha_4 \in Act$;
- $MAControl\text{🔔}? \in Act, II(MAControl\text{🔔}?) \in PRC^{\sigma_0(3) \rightarrow \sigma_1(0)}$; we assume the initial speed of the train is v_0, thus we have the guard condition $\sigma_0(3) \models C_{dot} = v_0$;
- $Dec\text{🔔}? \in Act, II(Dec\text{🔔}?) \in PRC^{\sigma_1(5) \rightarrow \sigma_2(0)}$; since $MAControl\text{🔔}? \in Act, II(MAControl\text{🔔}?) \in PRC^{\sigma_0(3) \rightarrow \sigma_1(0)}$, $\sigma_0(3) \models C_{dot} = v_0$ and $sem \models$*Train-Control*, according to Definition 4.3.12 there is $\sigma_1(0) \ldots \sigma_1(5) \models C_{dot} = v_1$, thus we have the guard condition $\sigma_1(5) \models C_{dot} = v_1$;
- $end\text{🔔}? \in Act, II(end\text{🔔}?) \in PRC^{\sigma_2(4) \rightarrow \sigma_3(0)}$; since $Dec\text{🔔}? \in Act, II(Dec\text{🔔}?) \in PRC^{\sigma_1(5) \rightarrow \sigma_2(0)}$, $\sigma_1(5) \models C_{dot} = v_1$ and $sem \models$*Train-Control*, according to Definition 4.3.12 there is $\sigma_2(0) \ldots \sigma_2(4) \models V_{dot} = b$,thus we have the guard condition $\sigma_2(4) \models V_{dot} = b$;
- $moveOn\text{🔔}? \in Act, II(moveOn\text{🔔}?) \in PRC^{\sigma_0(3) \rightarrow \sigma_0'(0)}$; since the initial speed of the train is v_0, thus we have the guard condition $\sigma_0(3) \models C_{dot} = v_0$;

The SRML based finite automaton for *Train-Control*, $D_{Train-Control}$, is presented in Formula 5.1 and the corresponding transition diagram for $D_{Train-Control}$ is shown in Figure 5.1. To make it more explicate, we provide all the paths of m selected by *Train-Control* shown in Figure 5.5. In this figure, the part in the dashed box can be seen to be repeated several times (might be infinite) if not considering the time instance in each state.

With the Brzozowski's method that is introduced in Section 5.1 , we construct the characteristic equations of each node in Figure 5.1 and show these equations in Formula 5.3. We also show the solutions for these characteristic equations in Formula 5.4.

$$\begin{aligned} E_{\sigma_0} &= Cont; (E_{\sigma_0} + \varepsilon) + Nego; E_{\sigma_1} \\ E_{\sigma_1} &= Corr; E_{\sigma_2} \\ E_{\sigma_2} &= Stop; (E_{\sigma_3} + \varepsilon) \\ E_{\sigma_3} &= \varepsilon \end{aligned} \tag{5.3}$$

$$\begin{aligned} E_{\sigma_0} &= Cont^*; (Cont + Nego; Corr; Stop) \\ E_{\sigma_1} &= Corr; Stop \\ E_{\sigma_2} &= Stop \\ E_{\sigma_3} &= \varepsilon \end{aligned} \tag{5.4}$$

Thus $E_{Train-Control} = E_{\sigma_0} = Cont^*; (Cont + Nego; Corr; Stop)$. According to the SRML specification for service module *Train-Control* in Appendix A and Definition 5.1.8, the hybrid program for *Train-Control* is:

$$hp(E_{Train-Control}) = hp(Cont^*; (Cont + Nego; Corr; Stop)) = \text{Formula 5.2} \tag{5.5}$$

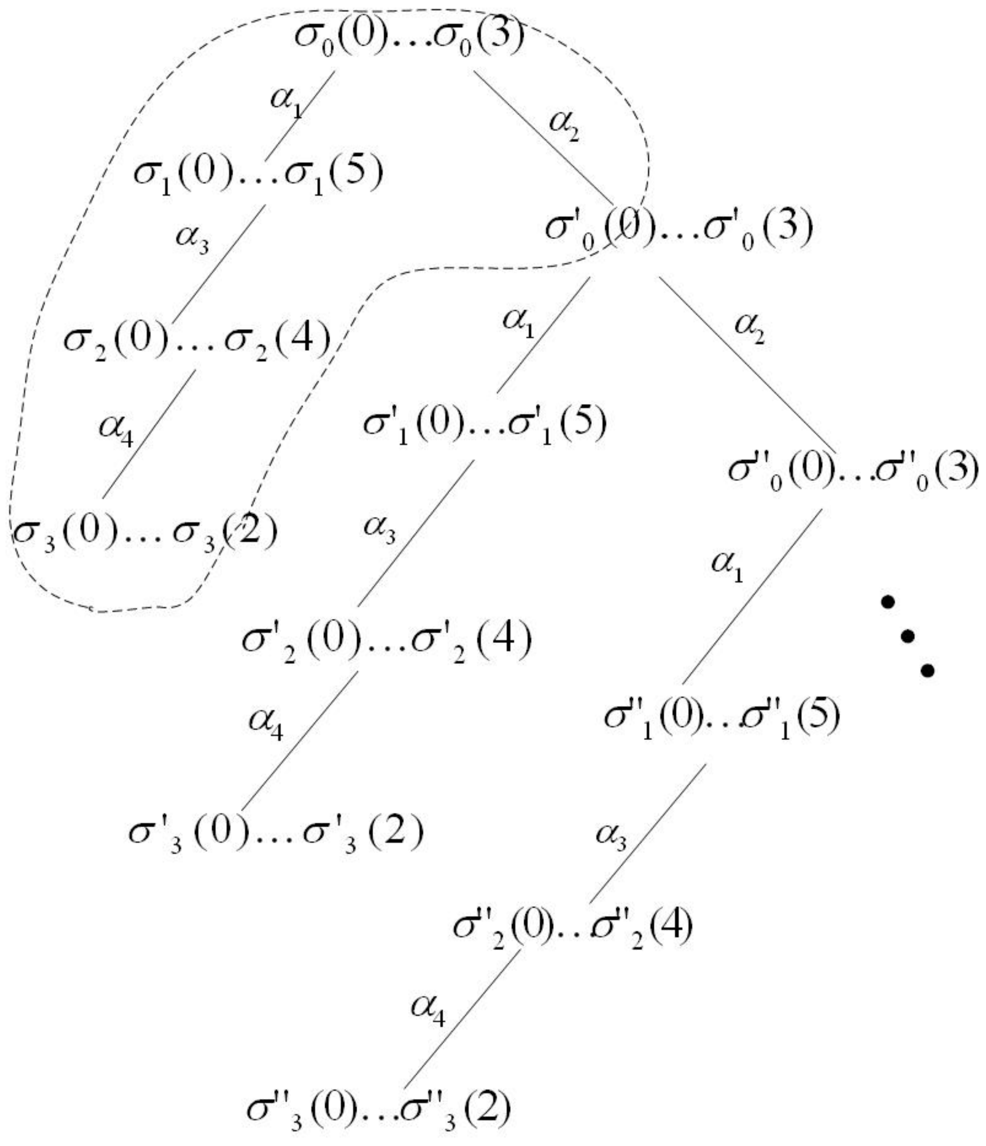

$\sigma_0(0)\ldots\sigma_0(3)$
α_1
α_2
$\sigma_1(0)\ldots\sigma_1(5)$
$\sigma'_0(0)\ldots\sigma'_0(3)$
α_3
α_1
α_2
$\sigma_2(0)\ldots\sigma_2(4)$
α_4
$\sigma'_1(0)\ldots\sigma'_1(5)$
$\sigma''_0(0)\ldots\sigma''_0(3)$
$\sigma_3(0)\ldots\sigma_3(2)$
α_3
α_1
$\sigma'_2(0)\ldots\sigma'_2(4)$
α_4
$\sigma''_1(0)\ldots\sigma''_1(5)$
$\sigma'_3(0)\ldots\sigma'_3(2)$
α_3
$\sigma''_2(0)\ldots\sigma''_2(4)$
α_4
$\sigma''_3(0)\ldots\sigma''_3(2)$

Figure 5.5: All the paths of SO-HL^{2}TS *m* that are selected by service module *Train-Control*

Thus the behavior constraint that needs to be verified in business protocol *MOC* can be written as:

$$[\text{Formula 5.2}]\mathit{receivePos}\boxtimes.T < L \rightarrow \mathit{receivePos}\boxtimes.\mathit{Pos} < N \tag{5.6}$$

and it expresses that within a certain time duration L, the position of the train will not exceed the position of movement authority N.

In order to make the verification more easily to be understood, we list the types and physical meanings of the terms (variables and constants) that are declared in service module *Train-Control* in Table 5.2:

Name	**Type**	**Data type**	**Physical meaning**
C	local variable	position	displacement of the train
C_0	constant	position	initial displacement of the train
M	local variable	position	Moving authority of the train
C_{dot}	local variable	speed	the derivative of the displacement of the train to real numbers
v_0	constant	speed	speed of the train at the free-moving stage
v_1	constant	speed	speed of the train at the negotiation stage
V	local variable	speed	speed of the train at the correction stage
V_{dot}	local variable	acc	the derivative of the displacement of the speed of the train at the correction stage to real numbers
b	constant	acc	acceleration of the train at the correction stage

Table 5.2: Terms declared in service module *Train-Control*

From the wire TB, we have $\mathit{receivePos}\boxtimes.T \equiv \mathit{end}\boxtimes.\mathit{currTime}$ and $\mathit{receivePos}\boxtimes.\mathit{Pos} \equiv \mathit{end}\boxtimes.\mathit{currPos}$, thus instead of verifying Formula 5.6, we only need to verify Formula 5.7:

$$[\beta_1^* \cup \beta_1;\beta_2;\beta_3;\beta_4]\mathit{end}\boxtimes.\mathit{currTime} < L \rightarrow \mathit{end}\boxtimes.\mathit{currPos} < N \tag{5.7}$$

where:

$$\begin{aligned}
&\beta_1 \equiv C_{dot} = v_0\\
&\beta_2 \equiv C_{dot} = v_1\\
&\beta_3 \equiv C_{dot} = V \wedge V_{dot} = b\\
&\beta_4 \equiv \mathit{end}\boxtimes.\mathit{currPos} = C - C_0 \wedge \mathit{end}\boxtimes.\mathit{currTime} = t' - t_0
\end{aligned}$$

We suppose that the default values of $\mathit{end}\boxtimes.\mathit{currPos}$ and $\mathit{end}\boxtimes.\mathit{currTime}$ are all zero. Thus it is obvious that if $[\beta_1^* \cup \beta_1;\beta_2;\beta_3]t' - t_0 < L \rightarrow C - C_0 < N$ holds, then Formula 5.7 holds. So we only have to verify:

$$[\beta_1^* \cup \beta_1;\beta_2;\beta_3]t' - t_0 < L \rightarrow C - C_0 < N \tag{5.8}$$

To simplify the verification of Formula 5.8, besides β_1, β_2, β_3 and β_4 that are given above, we also use the following abbreviations:

$$\psi \equiv C_0 < M \wedge v_0 > 0 \wedge L \geq 0$$
$$\phi \equiv t' - t_0 < L \rightarrow C - C_0 < N$$

where ψ is a sanity condition for Formula 5.8, and Formula 5.8 is true only under some certain invariant conditions. Since v_0 is the speed of the train at free moving stage and v_1 is the speed of the train at the negotiation stage in which normally the speed of the train must slow down, we can assume that $v_0 > v_1$. Thus we take $Lv_0 + C_0 < N$ as the invariant condition for $[\beta_1]\Box\phi$ to be true and take $\psi, t' \geq t \vdash v_1^2 < 2b(m - Lv_0 - C_0) \wedge Lv_0 + C_0 < N$ as the invariant condition for $[\beta_1][\beta_2][\beta_3]\Box\phi$ and $[\beta_1][\beta_2]\Box\phi$ to be true. Then we have to prove that

$$\begin{array}{l} \text{If } \psi \vdash Lv_0 + C_0 < N \text{ holds ,} \\ \text{then } \psi \vdash [\beta_1]\Box\phi \text{ holds;} \end{array} \tag{5.9}$$

and

$$\begin{array}{l} \text{If } \psi, t' \geq t \vdash Lv_0 + C_0 < N \text{ holds ,} \\ \text{then } \psi \vdash [\beta_1][\beta_2]\Box\phi \text{ holds;} \end{array} \tag{5.10}$$

and

$$\begin{array}{l} \text{If } \psi, t' \geq t \vdash v_1^2 < 2b(m - Lv_0 - C_0) \wedge Lv_0 + C_0 < N \text{ holds ,} \\ \text{then } \psi \vdash [\beta_1][\beta_2][\beta_3]\Box\phi \text{ holds.} \end{array} \tag{5.11}$$

The proof of 5.9 is shown as:

$$\begin{array}{ll}
 & \psi \vdash Lv_0 + C_0 < N \\ \hline
\text{P10} & \psi \vdash \forall t' \geq t(t' - t \leq L \rightarrow (t' - t)v_0 + C_0 < N) \\ \hline
\text{D3} & \psi \vdash \forall t' \geq t\{\!|C = (t' - t)v_0 + C_0|\!\}\phi \\ \hline
\text{D7} & \psi \vdash [\beta_1]\phi \\ \hline
\text{T5} & \psi \vdash [\beta_1]\Box\phi
\end{array}$$

The proof of 5.10 is shown as:

$$\begin{array}{ll}
 & \psi, t' \geq t \vdash Lv_0 + C_0 < N \\ \hline
\text{D3} & \psi, t' \geq t \vdash \{\!|C = C_0 + v_0(t' - t)|\!\}v_1(L - t_1) + C < N \\ \hline
\text{P10} & \psi, t' \geq t \vdash \{\!|C = C_0 + v_0(t' - t)|\!\}\forall t' \geq t(t' - t < L - t_1 \rightarrow C_1 + v_0(t' - t) < N) \\ \hline
\text{D3} & \psi, t' \geq t \vdash \{\!|C = C_0 + v_0(t' - t)|\!\}\forall t' \geq t\{\!|C = C_1 + v_1(t' - t)|\!\}\phi \\ \hline
\text{D7} & \psi, t' \geq t \vdash \{\!|C = C_0 + v_0(t' - t)|\!\}[\beta_2]\phi \\ \hline
\text{T5} & \psi, t' \geq t \vdash \{\!|C = C_0 + v_0(t' - t)|\!\}[\beta_2]\Box\phi \\ \hline
\text{P3} & \psi \vdash t' \geq t \rightarrow \{\!|C = C_0 + v_0(t' - t)|\!\}[\beta_2]\Box\phi \\ \hline
\text{P10} & \psi \vdash \forall t' \geq t\{\!|C = C_0 + v_0(t' - t)|\!\}[\beta_2]\Box\phi \\ \hline
\text{D7} & \psi \vdash [\beta_1][\beta_2]\Box\phi
\end{array}$$

The proof of 5.11 is shown as:

$$\begin{array}{ll}
 & \psi, t' \geq t \vdash v_1^2 < 2b(N - Lv_0 - C_0) \land Lv_0 + C_0 < N \\ \cline{2-2}
\text{D3} & \psi, t' \geq t \vdash \{\!| C = C_0 + v_0(t'-t) |\!\} v_1^2 < 2b(N - v_0(L - t_1) - C) \land (L - t_1)v_0 + C < N \\ \cline{2-2}
\text{D3} & \psi, t' \geq t \vdash \{\!| C = C_0 + v_0(t'-t) |\!\} \{\!| C = C_1 + v_1(t'-t) |\!\} \\
 & v_1^2 < 2b(N - v_0(L - t_1 - t_2) - C) \land (L - t_1 - t_2)v_0 + C < N \\ \cline{2-2}
\text{P10} & \psi, t' \geq t \vdash \{\!| C = C_0 + v_0(t'-t) |\!\} \{\!| C = C_1 + v_1(t'-t) |\!\} \\
 & \forall t' \geq t (t' - t < L - t_1 - t_2 \to \frac{b}{2}(t'-t)^2 + v_1(t'-t) + C_2 < N) \\ \cline{2-2}
\text{D3} & \psi, t' \geq t \vdash \{\!| C = C_0 + v_0(t'-t) |\!\} \{\!| C = C_1 + v_1(t'-t) |\!\} \\
 & \forall t' \geq t \{\!| C = \frac{b}{2}(t'-t)^2 + v_1(t'-t) + C_2 |\!\} \phi \\ \cline{2-2}
\text{D7} & \psi, t' \geq t \vdash \{\!| C = C_0 + v_0(t'-t) |\!\} \{\!| C = C_1 + v_1(t'-t) |\!\} [\beta_3] \phi \\ \cline{2-2}
\text{T5} & \psi, t' \geq t \vdash \{\!| C = C_0 + v_0(t'-t) |\!\} \{\!| C = C_1 + v_1(t'-t) |\!\} [\beta_3] \Box \phi \\ \cline{2-2}
\text{p3} & \psi \vdash t' \geq t \to \{\!| C = C_0 + v_0(t'-t) |\!\} \{\!| C = C_1 + v_1(t'-t) |\!\} [\beta_3] \Box \phi \\ \cline{2-2}
\text{p10} & \psi \vdash \forall t' \geq t \{\!| C = C_0 + v_0(t'-t) |\!\} \forall t' \geq t \{\!| C = C_1 + v_1(t'-t) |\!\} [\beta_3] \Box \phi \\ \cline{2-2}
\text{D7} & \psi \vdash [\beta_1][\beta_2][\beta_3] \Box \phi
\end{array}$$

From the proof of Formula 5.11 we have:

$$\text{T7} \; \frac{\psi \vdash [\beta_1] \Box \phi}{\psi \vdash [\beta_1^*] \Box \phi}$$

and from the proof of Formula 5.9, Formula 5.10, and Formula 5.11 we have

$$\begin{array}{ll}
 & \psi \vdash [\beta_1][\beta_2][\beta_3] \Box \phi \\ \cline{2-2}
\text{T1} & \psi \vdash [\beta_1][\beta_2; \beta_3] \Box \phi \\ \cline{2-2}
\text{T1} & \psi \vdash [\beta_1; \beta_2; \beta_3] \Box \phi
\end{array}$$

Thus we finally obtain:

$$\text{D1} \; \frac{\psi \vdash [\beta_1^*] \Box \phi \land \psi \vdash [\beta_1; \beta_2; \beta_3] \Box \phi}{\psi \vdash [\beta_1^* \cup \beta_1; \beta_2; \beta_3] \Box \phi}$$

Chapter 6

Mapping from SRML to implementation environment

In this chapter, we provide a method for mapping SRML service modules to service modules supported by IBM WebSphere Process Server (WPS), with which SRML service modules could be modeled and tested. Section 6.1 gives a general introduction to WebSphere products that support the development and deployment of SCA models. Section 6.2 shows the correspondence between declarations of SRML service modules and WID service modules. In particular, a set of mapping rules that represent the correspondence are provided in this section. Section 6.3 shows the result of implementation and testing of *Train-Control* module using WebSphere products.

6.1 Introduction to WebSphere Process Server

WebSphere Process Server (WPS) is one of the key products in the IBM WebSphere Business Process Management suite. It is a comprehensive SOA integration platform, which is based on WebSphere Application Server, and provides support for SCA programming model. As a process engine, WPS provides a hosting environment for business processes (a business process is a series of tasks executed in a specific order that is followed by an organization to achieve a larger business goal). It also provides several Web-based applications such as Business Process Choreographer Explorer and Business Process Choreographer Observer, which manage the various aspects of business processes.

SCA describes all integration artifacts as service components with well defined interfaces, and organizes business application code based on service components that implement business logic. These business applications provide their capabilities as services through *interfaces*, and require services offered by other components through *references*. To support such architecture, the construction of service-oriented applications for WebSphere Process Server includes the following aspects:

- Implementing service components that provide services to outside parties and require services provided by outside parties;

- Assembling service components to build service module by wiring service references;
- Generating bindings that use transport and protocol to connect to external clients and services;
- Assigning quality to attributes of service components.

In this section, we introduce the implementation of service components; the assembly of service components and the generation of bindings are introduced in the next section; the assignment of quality of service attributes is not discussed in this thesis.

In Websphere, a service component consists of an implementation, some interfaces that defines the inputs, outputs and faults of the service component, and zero or some references that identifies the interface of another service or component. The implementation is associated with the component and performs the logic of it. Based on WebSphere Application Server, WPS offers a variety types of service components implementation on top of a SOA core. The implementation types available for service components are listed as follows:

- **Interface maps**
  An *interface map* is a bridge component linking two SCA components that have different implementations. Since having different implementations, the interfaces of the components are labeled with different method signatures. The interface map enables them to communicate by mapping the operations and parameters of these interfaces. In this way the differences of the implementations are resolved and the two components can interact. An interface map maps two interfaces in two levels:
  - *Operation mappings*, by which operations of one interface are mapped to operations of another interface.
  - *Parameter mappings*, by which parameters of the business object in one method signature are mapped to parameters of the business object in another method signature. In this way, parameter mappings correlate the parameters associate with different operations. There are four different types of parameter mappings: *map*, which correlate parameters of two business objects that have different fields; *extract*, which correlate a complex parameter of one business object to the output parameters of another business object by extracting pertinent information of the complex parameter; *custom*, which performs the mappings by calling Java codes; and *assign*, which assigns a value directly to an output parameter. Parameter mappings are one level deeper than operation mappings and operation mappings can include parameter mappings.
- **Business state machines**
  A *business state machine* is an event-driven service component in which external operations trigger the changes from one discrete state to another. Each state is implemented with a mode, which determines which activities and operations can happen in that state. Business state machines enable the representation of business processes using states and events, which are useful for modeling real-time and event-driven systems. Business state

machines are normally used during business modeling and analysis design. In the first case, they are used to model use case scenarios; in the second case, they are used to model event-driven objects or model the different aspects of the same state machine.

- **Java objects**
  A Java object is a service component that is implemented with Java codes, and it is also called a plain old Java object (POJO). A java object consists of the Java codes that represent the business logic of the component, and a WSDL interface or a Java interface that is bounded to the component. Java objects are common implementations of service components.

- **Processes**
  A WPS *process* component, also called a *business process* component, implements a WS-BPEL compliant process engine. A business process consists of a series of activities, rules, conditions or individual tasks, which are executed sequentially or in parallel to achieve an integral business goal. These compositions of a business process are connected with *connectors*, which specify the logical order according to which the business process is executed. A business process can be declared as the transaction behaviors for invoke, human task and snippet activities, and can run in *microflows* mode or *long-running business process* mode. Microflows run in a single transaction which lasts for a short period of time, and long-running business processes run in a series of chained transitions which last for days or months.

- **Human tasks**
  A *human task* in WPS is a stand-alone component that can assign work to human users or to invoke other service components. It performs an activity that needs the interaction of a human user, and it provides a common interface for humans to deal with human centric and automatic tasks in a uniform way. Human task components include built-in support for role-based task assignment, scheduling and escalation policies in case a task is not processed within a predefined time limit. The properties of a task type represent the meta information specific for that task. There are four main types of human tasks:

  – *To-do task*, which is invoked by a service component that assigns a task to a human to do something.
  – *Invocation task*, which is triggered by a human who assigns a task to a service component.
  – *Collaboration task*, which is triggered by a human who assigns a task to another human.
  – *Administration task*, which is created by components to offer an interface for a human administrator.

- **Selectors**
  A *selector* is a routing component that selects the service component to invoke dynamically

at run time. It consists of a set of date range entries, a set of selection criteria and a default destination. A selector selects the component to invoke using the date range entries and selection criteria that are declared in it. In detail, the selector changes the component chosen to be invoked according to certain selection criteria, and this criteria evaluates to a specific date range entries. The component chosen to be invoked can be any SCA service component.

- **Rule groups (business rules)**
  Business rules are service components which declare the policies or conditions that must hold in a business process. These policies or conditions specify how business policies or practices are executed in a business activity. In business rules, the business policies, conditions and values that changes over time are not included. Thus, business process applications are more flexible by using business rules. Business rules can be implemented in two types: a *rule set* that consists of a set of if-then rules, and a *decision table* that formalizes simple rule logics in the format of a table. These two types are not used separately but always combined to form what are called *rule groups*.

Figure 6.1 from [53] shows the implementation of a WPS component and its interface and reference.

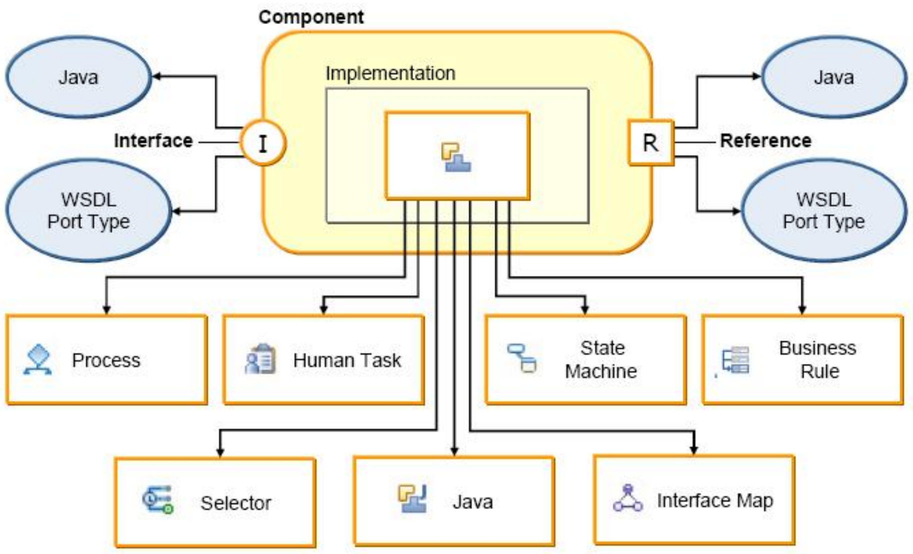


Figure 6.1: A WPS component with all types of implementation

6.2 Mapping from SRML module to WPS module

When having a service module well specified in SRML, we can transform it to WPS service module. In order to realize the transformation, a set of functions that map from SRML specification domain to WPS implementation domain are needed. We first give a brief introduction to the WPS domain, which includes building blocks of WPS service module that are used for implementing SRML service module, then show the definitions of the functions.

In WPS, a service module of SRML can be represented in a standard block diagram referred to as a *module assembly*, or an *assembly diagram*. As introduced in Section 6.1, in WPS there are seven different implementation types of SCA components. Among these types, we choose state machine as the implementation of WPS component as it could be easily obtained form SRML based finite automata defined in Chapter 5. In the assembly diagram the components are wired together and each component has its own interface to publish and receive messages. In the implementation of a SRML service module, we map variables and transitions of all the business roles into a single WPS component which is implemented by state machine (in such a WPS state machine, differential equations specified in the SRML service module are implemented as their respective solutions, since WPS state machine requires the exact value of each variable in the computation approach, and these solutions are written in Java codes), map interactions into WPS interface and reference, map business protocol which links to a service requirer into WPS export, map business protocol which links to a service provider into WPS import and map interaction protocols into WPS wires. Figure 6.2 shows the correspondence between a SRML service module and a WPS service module in an assembly diagram.

We introduce each part of the WPS service module shown in Figure 6.2 as follows:

Interface: An *interface* declares a set of operations that are accepted and responded by the state machine. It provides the input and output of a service component by applying the operations and is independent of the implementation type of that component. One service component can only have one interface.

E.g.,create an interface named TrainControlInterface *for the module, with five operations as shown in Table 6.1. Each operation triggers the corresponding transition from source state to target state.*

Operation type	Operation name	Input name	Input type	Output name	Output type
One Way	Start	id MA	string float	-	-
One Way	Continue	id MA	string float	-	-
One Way	MAControl	id	string	-	-
One Way	MAControlConfirm	id	string	-	-
Request Response	End	id	string	dis time	float float

Table 6.1: Operations of *TrainControlInterface*

Reference:A *reference* specifies the operations that can be invoked by the state machine by pointing to the interface that includes these operations in another service component. A service component can include zero or more references to other service components or imports that are included in the current module.

Variables: *Variables* store the data used in a state machine. There are two types of variables that

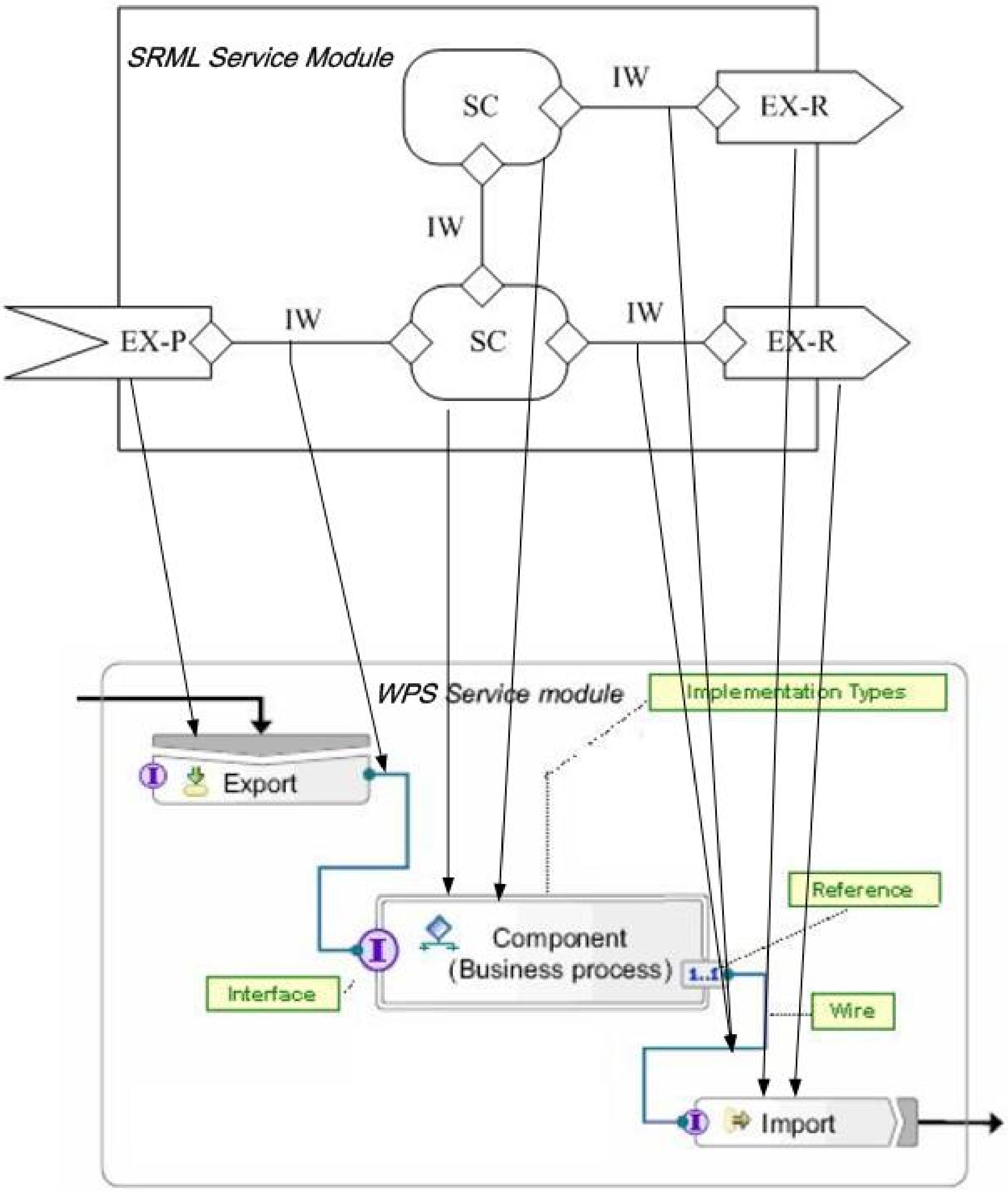


Figure 6.2: The correspondence between SRML service module and WPS service module

can be used in a state machine:

- *Data type variable*: Can be either a business object or a simple type, such as string or integer.
  E.g., the variable names and types for WPS module TrainControl shown in Table 6.2.

- *Interface variables*: Uses either an input or output parameter as defined within an interface.
  E.g., inputs and outputs of the operations shown in Table 6.1.

A variable can be either global (created for the global state machine scope) or local (created for nested scopes and only visible to objects within the scopes).

Variable name	C	M	t	V	dotC	dotV	v0	v1	b
Variable type	float	float	int	double	double	int	double	double	int

Table 6.2: Data type variables of *TrainControl* module

Import An *import* identifies service provider outside a service module and can be called from the inside of the module. An import component has an associated binding that specifies the way of transporting the data to and from the external service.
Export An *export* exposes the interface of the state machine to external service require. It has an associated binding that describes the physical communication mechanism to be used.
Wires *Wires* are used to link pairs of components in an assembly diagram.
Correlation properties: *Correlation properties* are specific variables that are used to distinguish different instances of a state machine at runtime.
E.g., for each operation (event) that the state machine responds to, the interface variable "id" with data type "string" is used as the correlation property.
State: A *state* is a discrete stage in which a business transaction can take place. In this thesis, we use three types of states to construct state machines:

- *Initial state*: Is the first state in which a state machine is started by an operation of the outbound transition of it.

- *Simple state*: Is a general state that can have an entry action and an exit action. The entry action occurs when the state is entered and the exit action occurs when the state is exited. When the entry action occurs, all the outbound transitions of the state are triggered.

- *Final state*: Is the last state in which a state machine terminates in a normal or expected completion. Since in a state machine, the finial state is the last state, it has not exit action.

E.g., the state machine ETCSC in Figure 6.4 includes five states: InitialState1 is the initial state; State1, State2, State3 are simple states, FinalState1 is a finial state.

Transition: A *transition* represents the movement from one state to the next by recognizing an appropriate triggering event. The execution of the movement is controlled by the evaluation of the conditions that associate with it. When a transition is enabled, the set of actions that associate with it are executed. A transition can include an event, a condition and a set of actions and they are introduced respectively as follows:

- **Event**: An *event* triggers a transition with which it associates when it is evoked. There are three types of events that can trigger a transition:
  - *Call event*: The transition is triggered when the correct operation is called.
  - *Timer event*: The transition is triggered when the timer expires.
  - *Completion event*: The transition is triggered when the source state is entered and its entry action, if any, is executed.

  In the state machine *ETCSC*, only call events are included. The call events are actually operations defined in the interface, and the attributes of the events are the input parameters of the operation. If the transition has an action, these attributes are available to the action.

- **Condition**: A *condition* is a guard of the transition with which it associates. When the evaluation of the condition is true, the state machine can move to the next state, otherwise the state machine must remain in the current state.
  E.g., in Figure 6.4, Condition0 is a condition on the self transition to state State1.

- **Action**: *Actions* associate with a transition are the activities that are executed when the state in which the transition takes place is entered or exited, or when the transition is triggered.
  E.g., in Figure 6.4, ActCon is an action which assigns some variables with certain values and is executed when event Continue is triggered; Entry1 is an action which prints out a message when some condition is satisfied and is executed when state State1 is entered.

The correspondence between elements in SRML service module domain and elements in WPS service module domain with state machine being the implementation type for service components is shown in Table 6.3

In the implementation of SRML service module, some elements in SRML service module domain such as wire, variable and service requirer/provider are directly realized by elements in WPS service module domain; and others such as trigger, guard and effect need functions that map formally from SRML expressions to WPS codings. We provide functions for mapping SRML interactions and transitions to WPS interface and transitions as follows:

Definition 6.2.1 (Parameter sequence)**.** Given an interaction signature $\langle NAME, PARAM\rangle$ and an interaction name $opName \in NAME$, the input parameter sequence $inParSeq_{opName}$ and output parameter sequence $outParSeq_{opName}$ of $opName$ is defined as follows:

- $inParSeq_{opName} = empty|\text{🔔}inPar_1 : inParType_1, \dots, inPar_n : inParType_n$, where $inPar_i \in PARAM_{\text{🔔}}(opName)_{inParType_i}$ and $inParType_i \in D$ for every $1 \leq i \leq n$;

SRML service module	**WPS service module implementation type for components: state machine**
transition	transition
trigger	call event(operation)
guard	condition
effect	entry action
variable	data type variable
parameter	interface variable
wire	wire
interaction	interface-reference
service provider interface	import
service requirer interface	export
node of the SRML based finite automaton	state

Table 6.3: Correspondence between elements in SRML service module domain and elements in WPS service module domain (implementation type of WPS service components: state machine)

- $outParSeq_{opName} = empty|\text{✉}outPar_1 : outParType_1, \dots, outPar_m : outParType_m$, with $outPar_i \in PARAM_{\text{✉}}(opName)_{outParType_i}$ and $outParType_i \in D$ for every $1 \leq i \leq m$.

Definition 6.2.2 (Interaction sequence)**.** Given a interaction signature $\langle NAME, PARAM\rangle$, a SRML interaction OP is defined as $OP = opType\ opName\ inParSeq_{opName}\ outParSeq_{opName}$ where $opType \in \{rcv, r\&s\}$ and $opName \in NAME$. Thus a interaction and the concatenation of interaction sequences are defined as follows:

- $interSeq = empty|OP_1\ OP_2 \dots OP_n$ is an interaction sequence iff OP_i is a SRML interaction for every $1 \leq i \leq n$;
- If $interSeq = OP_1\ OP_2 \dots OP_n$ and $interSeq' = OP'_1\ OP'_2 \dots OP'_m$, then $interSeq \circ OP'_1\ OP'_2 \dots OP'_m = OP_1\ OP_2 \dots OP_n\ OP'_1\ OP'_2 \dots OP'_m$;
- If $interSeq$ is an interaction sequence, then $empty \circ interSeq = interSeq \circ empty = interSeq$.

Definition 6.2.3 (Mappings of interaction types and parameters from SRML to WPS)**.** $f_{interType}$: SRML-TYPE $\rightarrow$ WID-operation type, $f_{inputPar}$: SRML-input parameter sequence $\rightarrow$WID-input name and input type, $f_{outputPar}$: SRML-output parameter sequence $\rightarrow$ WID-output name and output type are three functions which map from SRML domain to WPS domain, where:

- $f_{interType}(r\&s) =$ Request Response;
- $f_{interType}(snd) =$ One Way;
- $f_{inputPar}(empty) = -, -$;
- $f_{inputPar}(\text{🔔}inPar_1 : inParType_1, \dots, inPar_n : inParType_n) = inPar_1 \dots inpar_n, inParType_1 \dots inParType_n$;

- $f_{outputPar}(empty) = -,-;$
- $f_{outputPar}(✉outPar_1 : outParType_1, \dots, outPar_n : outParType_n) =$ $outPar_1 \dots outpar_n, outParType_1 \dots outParType_n.$

Definition 6.2.4 (Mapping from SRML interaction to WPS interface). $f_{interface}$: SRML-interaction sequence $\rightarrow$ WPS-interface table is a function which maps form SRML interaction domain to WPS interface domain, and is inductively defined as follows:

$$\begin{aligned}&f_{interface}(opType\ opName\ inParSeq_{opName}\ outParSeq_{opName} \circ interSeq) =\\&f_{interType}(opType), opName, f_{inputPar}(inParSeq_{opName}),\\&f_{outputPar}(outParSeq_{opName})\ f_{interface}(interSeq)\end{aligned}$$

where $opType\ opName\ inParSeq_{opName}\ outParSeq_{opName}$ is a SRML interaction and $interSeq$ is a SRML interaction sequence. In particular, when $interSeq = \emptyset$, i.e. there is no interaction included in the SRML interaction sequence $interSeq$, there is:

$$\begin{aligned}&f_{interface}(opType\ opName\ inParSeq_{opName}\ outParSeq_{opName} \circ interSeq) =\\&f_{interType}(opType), opName, f_{inputPar}(inParSeq_{opName}), f_{outputPar}(outParSeq_{opName})\end{aligned}$$

Definition 6.2.5 (Function of WPS expressions). f_{exp} is a function which assign SRML terms and formulas with WPS expressions, where f_{exp} : SRML-terms and expressions $\rightarrow$ WID-codes. The WPS expressions are written in Java. Function f_{exp} is defined as follows:

- $f_{exp}(opName\#) = opName$, where $opName \in NAME$ and $\# \in \{🕭, ✉, ✓, ✞, ✗\}$;
- $f_{exp}(T) = T^{WID}$ where, $T \in STERM \cup ETERM$ and T^{WID} is the WID expression of T;
- $f_{exp}(true) = true$;
- $f_{exp}(T = T') = f_{exp}(T) == f_{exp}(T')$, where $T, T' \in STERM$ or $T, T' \in ETERM$;
- $f_{exp}(\neg\phi) = f_{exp}(\phi)!$, where $\phi \in LS \cup LE$;
- $f_{exp}(\phi \wedge \phi') = f_{exp}(\phi) \&\& f_{exp}(\phi')$, where $\phi, \phi' \in LS$ or $\phi, \phi' \in LE$.

Based on function of WID expressions, we can define three functions: $f_{trigger}$, f_{guard} and f_{effect} which map *trigger*, *guard* and *effect* in SRML into WID.

Definition 6.2.6 (Functions of trigger, guard and effect). $f_{trigger}$, f_{guard} and f_{effect} are functions which map *trigger*, *guard* and *effect* of SRML transitions into WID, and are defined as follows:

- $f_{trigger}(a\#) = f_{exp}(a\#)$, where $a \in NAME$, $\# \in \{🕭, ✉, ✓, ✞, ✗\}$;
- $f_{guard}(\phi) =$ if $f_{exp}(\phi)$ *return true; return false;*, where $\phi \in LS$;
- $f_{effect}(\phi) =$ if $f_{exp}(\phi)$ *system.out.println "true"; system.out.println "false";*, where $\phi \in LE$;

6.3 Transformation and implementation of SRML module *Train-Control*

The transformation from SRML service module *Train-Control* to WPS module *TrainControl* includes constructing an automaton of SRML service module and implementing WPS state machine and interface. The automaton of SRML service module *M* is constructed based on the SRML based finite automaton of *M*. The implementation of WPS state machine and interface is based on the mapping functions defined in Section 6.2. As a whole, the transformation can be done in the following steps:

1. Draw an automaton of *Train-Control* module

   The automaton of *Train-Control* module is obtained from the diagram of the SRML based finite automaton $D_{Train-Control}$ shown in Figure 5.1 and the business role *ETCSC* shown in Appendix A:

   a. Adapt all the nodes and edges from Figure 5.1 for a new diagram;
   b. Label all the nodes in the new diagram as that in Figure 5.1;
   c. If an edge in Figure 5.1 is labeled with *tr* ($tr \in TRANSName$), then label the same edge in the new diagram with *tri(tr)* and *gua(tr)*, and add *eff(tr)* to the label of target node of the edge.

   we have the automaton of SRML service module *Train-Control* shown in Figure 6.3.

   where:

   - *tri(Nego)≡MAControl🔔?*, *gua(Nego)≡* $\dot{C} = v_0$, *eff(Nego)≡* $\dot{C} = v_1$;
   - *tri(Corr)≡Dec🔔?*, *gua(Corr)≡* $\dot{C} = v_1$, *eff(Corr)≡* $\dot{C} = V \wedge \dot{V} = b$;
   - *tri(Cont)≡moveOn🔔?*, *gua(Cont)≡* $\dot{C} = v_0$, *eff(Cont)≡* $\dot{C} = v_0$;
   - *tri(Stop)≡end🔔?*, *gua(Stop)≡* $\dot{V} = b$,
     eff(Stop)≡ reportPos🔔!.currPos=C+M∧reportPos🔔!.time=t.

2. Implement the automaton of *Train-Control* module with WPS business logic — state machine

   Figure 6.4 shows the state machine of WPS business logic which implements the *Train-Control* module.

   The mappings from SRML specifications of *Train-Control* module to expressions in WPS, according to the mapping rules presented in Section 6.2, are listed as follows:

   - $f_{exp}(MAControl$🔔$?) = MAControl$
   - $f_{exp}(Dec$🔔$?) = MAControlConfirm$
   - $f_{exp}(moveOn$🔔$?) = Continue$
   - $f_{exp}(end$🔔$?) = End$

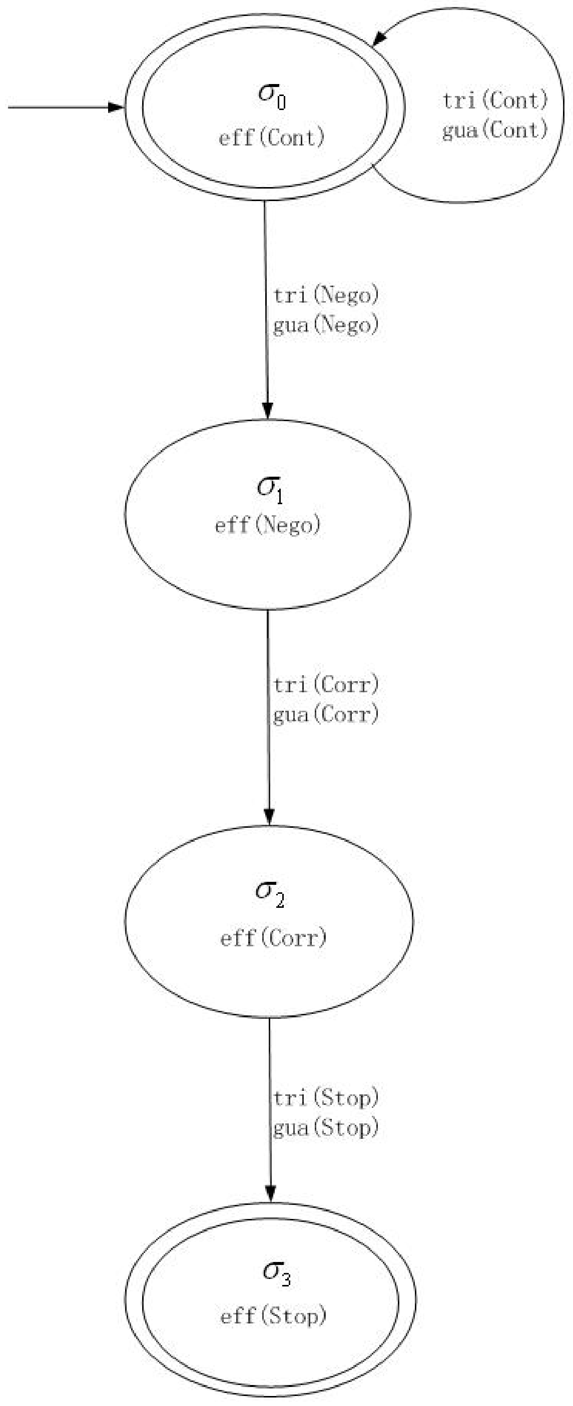


Figure 6.3: Automaton for SRML service module *Train-Control*

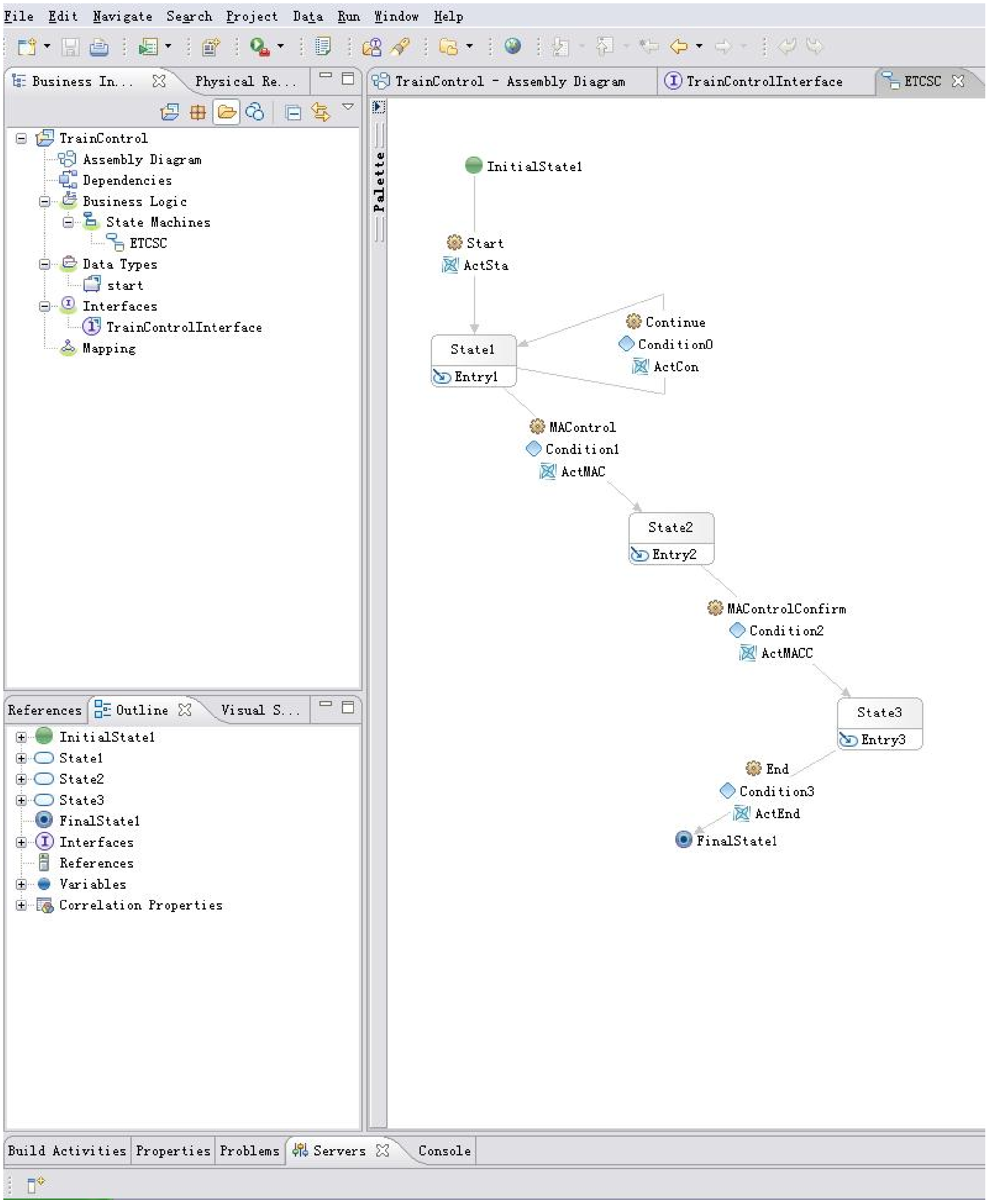


Figure 6.4: WPS state machine for module *Train-Control*

- $f_{exp}(\dot{C} = v_0) = dotC == v0$
- $f_{exp}(\dot{C} = v_1) = dotC == v1$
- $f_{exp}(\dot{V} = b) = dotV == b$
- $f_{exp}(\dot{C} = V \wedge \dot{V} = b) = dotC == V \&\& dotV == b$
- $f_{exp}(end\boxtimes!.currPos = C + M \wedge end\boxtimes!.time{=}t) =$ $Pos == C + MA \&\& time == t$

In Figure 6.4, node labeled with *State1* corresponds to node labeled with σ_0 in Figure 6.3; respectively, node labeled with *State2* corresponds to node labeled with σ_1; node labeled with *State3* corresponds to node labeled with σ_2; node labeled with *FinalState1* corresponds to node labeled with σ_3. To initialize the SRML automaton in Figure 6.3, we add node *InitialState1* in the WPS implementation as the initial state (see Figure 6.4).

In Figure 6.4, *Continue*, *MAControl*, *MAControlConfirm* and *End* match to *tri(Cont)*, *tri(Nego)*, *tri(Corr)* and *tri(Stop)* in Figure 6.3, and are implemented as:

- $Continue : f_{trigger}(tri(Cont))$
- $MAControl : f_{trigger}(tri(Nego))$
- $MAControlConfirm : f_{trigger}(tri(Corr))$
- $End : f_{trigger}(tri(Stop))$

Condition1, *Condition2* and *Condition3* mach to *gua(Cont)*, *gua(Nego)*, *gua(Corr)* and *gua(Stop)*, and are implemented as:

- $Condition0 : f_{guard}(gua(Cont))$
- $Condition1 : f_{guard}(gua(Nego))$
- $Condition2 : f_{guard}(gua(Corr))$
- $Condition3 : f_{guard}(gua(Stop))$

Entry1, *Entry2*, *Entry3* and *Entry4* mach to *eff(Cont)*, *eff(Nego)*, *eff(Corr)* and *eff(Stop)*, and are implemented as:

- $Entry1 : f_{effect}(eff(Cont))$
- $Entry2 : f_{effect}(eff(Nego))$
- $Entry3 : f_{effect}(eff(Corr))$
- $Entry4 : f_{effect}(eff(Stop))$

ActSta, *ActCon*, *ActMAC* and *ActMACC* in Figure 6.4 are assignments of variables, *ActEnd* returns the displacement and the moving time of the train. They are implemented as (in JAVA code):

- *ActSta*: v0=(float)100; C=(float)0; t=(float)0; M=Start_Input_MA; dotC=(float)100;

- *ActCon*: t=t+(float)3; C=C+3*v0; M=C+M+Continue_Input_MA;
- *ActMAC*: v1=(float)50; t=t+(float)5; C=C+5*v0; dotC=v1;
- *ActMACC*: b=(float)10; C=C+v1*(float)4-(float)0.5*b*(float)16; V=v1-b*(float)4; dotC=v1-b*(float)4; dotV=(float)10;
- *ActEnd*: End_Output_dis=C; End_Output_time=t;

Notice that in an automaton of SRML service module, if a node marked with double circle is linked to an edge with outgoing arrow, this node in the corresponding WPS state machine is implemented with a normal state, otherwise this node would be implemented with a finial state. This is because in WPS state machine, a finial state has no outgoing transition to other states. And for such normal states, we should control it manually at run time when they act as finial state. That's why in Figure 6.4 *State1* is a normal state but not a finial state.

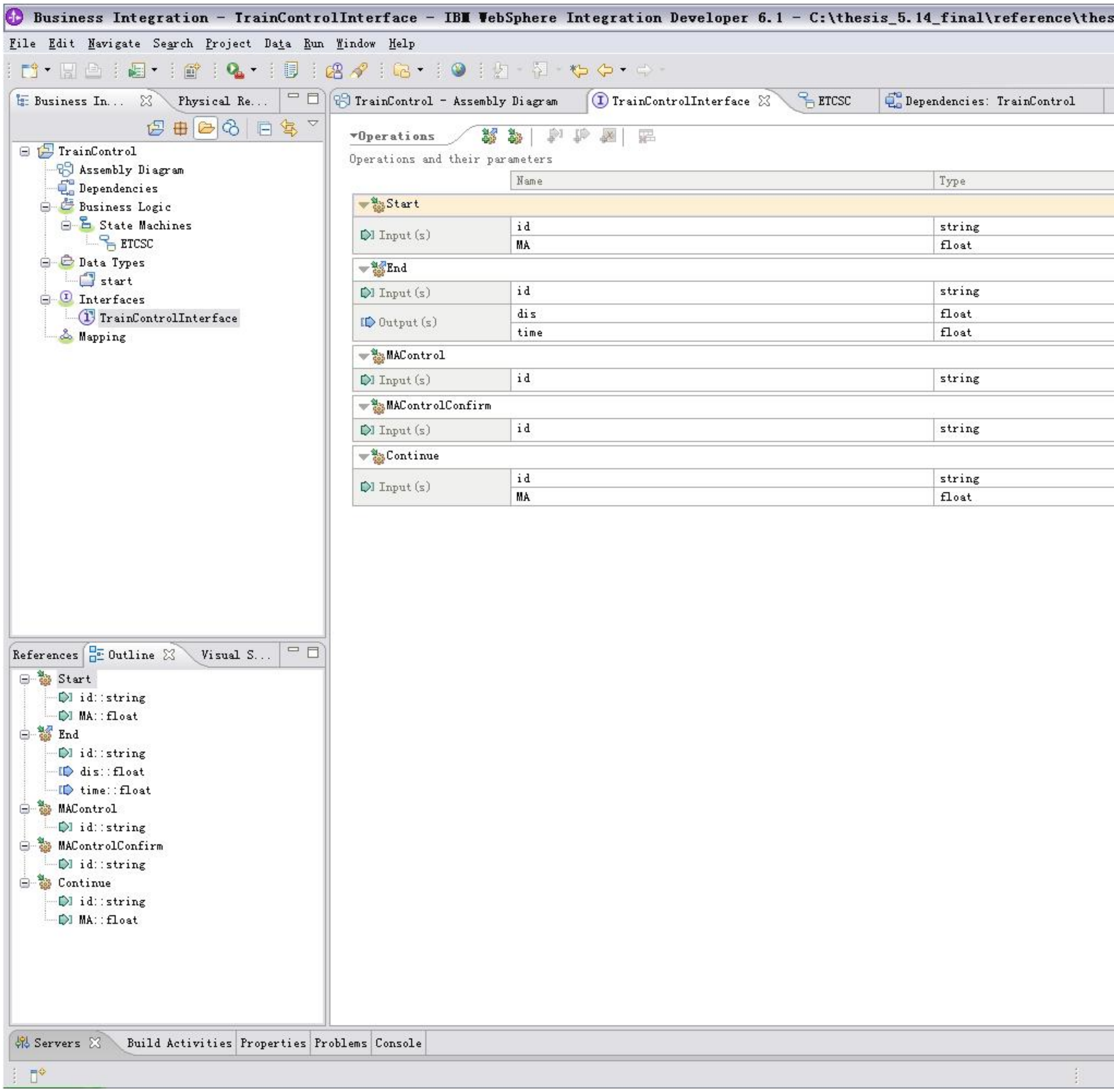
3. Implement interactions with WPS interfaces

   The interactions of business role *ETCSC* (specified in Appendix A) are mapped to WPS interface using the mapping rules in Section 6.2. The mappings are presented as follows:

   - The parameter sequences of *ETCSC* are obtained according to Definition 6.2.1:
     - $inParSeq_{MAControl} = empty$, $outParSeq_{MAControl} = empty$;
     - $inParSeq_{Dec} = empty$, $outParSeq_{Dec} = empty$;
     - $inParSeq_{moveOn} =$ 🔔$newMA : position$, $outParSeq_{moveOn} = empty$;
     - $inParSeq_{end} = empty$, $outParSeq_{end} =$ ✉$currPos : position, currTime : time$.
   - The interaction sequence $interSeq_{ETCSC}$ of business role *ETCSC* is obtained according to Definition 6.2.2:
     - $OP_1 = rcv\ MAControl\ inParSeq_{MAControl}\ outParSeq_{MAControl}$;
     - $OP_2 = rcv\ Dec\ inParSeq_{Dec}\ outParSeq_{Dec}$;
     - $OP_3 = rcv\ moveOn\ inParSeq_{moveOn}\ outParSeq_{moveOn}$;
     - $OP_4 = r\&s\ end\ inParSeq_{end}\ outParSeq_{end}$;
     - $interSeq_{ETCSC} = OP_1, OP_2, OP_3, OP_4$
   - The mapping from interactions of *Train-Control* module to the interface of WPS module *TrainControl* is obtained inductively according to Definition 6.2.4:

     $f_{interface}(rcv\ MAControl\ inParSeq_{MAControl}\ outParSeq_{MAControl} \circ OP_2, OP_3, OP_4) =$
     $f_{interType}(rcv), f_{exp}(MAControl), f_{inputPar}(inParSeq_{MAControl}),$
     $f_{outputPar}(outParSeq_{MAControl}), f_{interface}(OP_2, OP_3, OP_4)$

     The result of the mapping can be seen in Table 6.1, the input variable "*id*" is a correlation property used to distinguish one instance of the state machine of *TrainControl* module from another within a runtime environment, thus is not specified in SRML. And the WPS implementation can be seen in Figure 6.5


Business Integration - TrainControlInterface - IBM WebSphere Integration Developer 6.1 - C:\thesis_5.14_final\reference\thes
File Edit Navigate Search Project Data Run Window Help
Business In...
Physical Re...
TrainControl
Assembly Diagram
Dependencies
Business Logic
State Machines
ETCSC
Data Types
start
Interfaces
TrainControlInterface
Mapping
TrainControl - Assembly Diagram
TrainControlInterface
ETCSC
Dependencies: TrainControl
Operations
Operations and their parameters
References
Outline
Visual S...
Servers
Build Activities
Properties
Problems
Console

	Name	Type
Start		
Input(s)	id	string
	MA	float
End		
Input(s)	id	string
Output(s)	dis	float
	time	float
MAControl		
Input(s)	id	string
MAControlConfirm		
Input(s)	id	string
Continue		
Input(s)	id	string
	MA	float

```
Start
    id::string
    MA::float
End
    id::string
    dis::float
    time::float
MAControl
    id::string
MAControlConfirm
    id::string
Continue
    id::string
    MA::float
```

Figure 6.5: WPS interface for module *Train-Control* interaction

We adopt WebSphere Integration Developer (WID) to build WPS service module. WID provides a development environment for building integrated applications. It enables integration developers to create, manage and test services for IBM WPS and WESB. The features in WID separate business logic from implementation details.

WID is built on the Rational Software Development V7.0.0.5 Platform, which is based on Eclipse 3.2 technology. Each IBM product that is built on the Rational Software Development Platform coexists and shares plug-ins and features with other products that are based upon this platform. There are two primary user roles that are associated with WID:

- *Integration developer*: Integrates existing and new services and users into the business process defining the service composition components. The specialist typically uses visual composition tools and service-bus configuration tools to wire abstract service components that comprise the business processes.

- *Application developer*: Implements the design for services that are provided by the software architect. This includes using an appropriate language and technology in which to implement the services, and following the design for those components provided by the software architect.

A typical development flow for the development process of business integration and mediation modules in WID is as follows:

1. Start WID and open a workspace.

2. Switch to the Business Integration perspective for development.

3. Create a library to store artifacts, such as business objects and interfaces, that are shared among multiple modules.

4. Create a new module or mediation module.

5. Create the business objects to contain the application data.

6. Create the interface and define the interface operations for each component.

7. Create and implement the service components.

8. Build the module assembly by adding the service components, imports and exports to the assembly diagram. Bind the imports and exports to a protocol.

9. Test the module in the integrated test environment.

10. Deploy the module to WPS.

11. Share the tested module with others on the team by putting it in a repository.

WID provides an assembly editor where the developer groups service components into modules and specifies which service interfaces are exposed by the module to a service requirer. It also provides a runtime environment including WPS and WESB. When deploying a module well developed in WID to WPS, it is connected to services such as Java beans, Web services or service components that WPS and WESB provide at runtime to form complete integration solutions. Figure 6.6 from [53] shows different levels in which the WPS components are developed in WID.

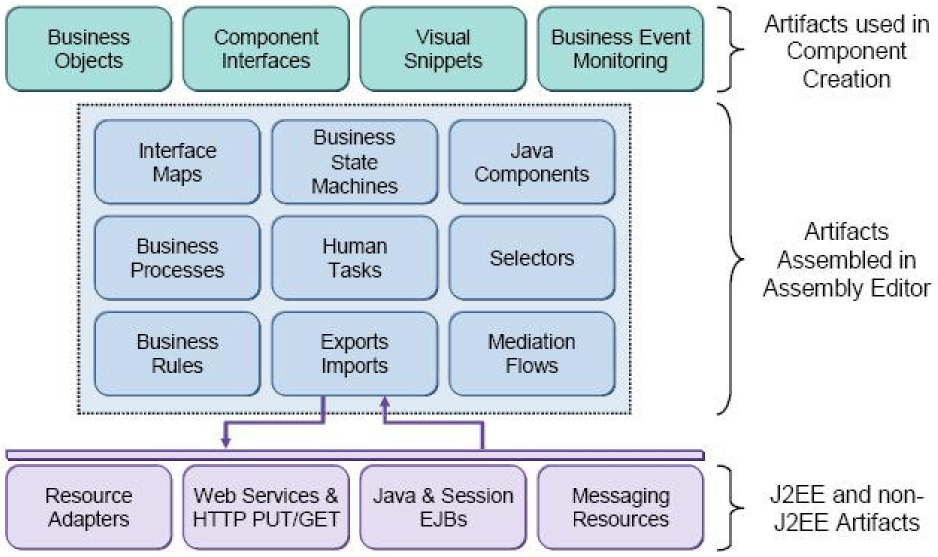


Figure 6.6: Key features of WebSphere Integration Developer

A work flow for implementing WPS module *TrainControl* in WID is provided in Appendix B.

Chapter 7

Conclusion and future prospects

In this thesis, our main goal is to give a formal specification for hybrid systems in a Service-Oriented point of view and to develop a method of formal verification. In order to achieve the first part of this goal, we make a hybrid extension of the SO-L^{2}TSs, named as SO-HL^{2}TSs, make an extension of the modeling language SRML and interpret it over SO-HL^{2}TSs. To achieve the second part of this goal, we adopt dTL formulas and a set of sequent calculus for verifying the formulas, and develop a method for transforming the SRML specification of a certain service module into the respective dTL formulas that could be verified. In the end, we provide a case study of a small part of the European Train Control System, show the SRML specification and the procedure of the verification for some property of the system, and provide a set of rules which map the SRML specification into a WID service module, by which part of the features of the hybrid system specified by SRML can be implemented.

In this chapter, a summary and some conclusions of the whole thesis are presented and the future prospects are provided by comparing the achievement of this thesis with other related works.

7.1 Summary and conclusions

This thesis succeeded in providing a SOC-based formal specification and a method of formal verification for hybrid systems, it also found out a way of implementing and testing part of the features of hybrid systems with SCA based integrated platforms – WID and WPS. The main contribution of this thesis is stated as follows:

- The definitions of SO-HL^{2}TS and its paths.
  SO-HL^{2}TS is a hybrid extension of SO-L^{2}TS. It defines a set of functions which map from certain real number intervals to sequences of states, and defines a set of transitions relations between these sequences of states. This makes the specification and interpretation of continuous time execution in a hybrid transition system possible. Paths of a SO-HL^{2}TS differs from paths of an SO-L^{2}TS in that a path of a SO-HL^{2}TS consists of finite or infinite traces of states, which are mapped form real numbers with functions declared in that SO-HL^{2}TS. These traces of states can represent the continuous behavior of the SO-HL^{2}TS.

- The extension of SRML.
  SRML provides a formal framework for modeling business services and activities. In detail, it defines syntax and semantics of components, interfaces and wires for service modules. The syntax of SRML are extended by enriching notions defined in business roles and notions defined in business protocols. The semantics of SRML are extended by interpreting SRML over SO-HL^{2}TS.

- The verification of SRML constraints.
  One of the most important SRML extension is the hybrid behavior constraint of business protocols. The behaviors specified with the hybrid behavior constraint are dTL formulas that are constructed over hybrid programs and can be verified with dTL sequent calculus adapted from [41]. To verify SRML behaviors specified with such behavior constraint, a method is provided for transforming the SRML behaviors into dTL formulas that could be verified. When given a path of the SO-HL^{2}TS over which the SRML behaviors to be verified are interpreted, this method finds out the SRML based finite automaton for the SO-HL^{2}TS, adopts Brzozowiski's method to construct its regular expression, transforms this regular expression into the respective hybrid programs and substitute them into the original behaviors to obtain the dTL formulas that could be verified.

- The mapping from SRML service module to WPS service module.
  IBM WebSphere Process Management suite provides a SOA integration platform for developing, deploying and testing the functional behaviors of services and the communications among services, over which business processes are constructed. Among the products in WebSphere Process Management suit, WebSphere Integration Developer and WebSphere Process Server are adopted for implementing and testing SRML service modules. SRML service modules are implemented with WPS state machines and the respective WPS componenet interfaces and references. In our approach, a set of formal mapping rules from SRML domain to WPS domain which include state machines, interfaces and references are provided. Over these rules, part of the activities of a business process could be implemented with WID, sequences of actions and values of variables of the business process could be checked with WPS.

In this thesis, we extend SRML not only by adding new terms to its syntax and semantics, but more important, we extended its semantic domain by defining SO-HL^{2}TSs and interpreted SRML terms over SO-HL^{2}TSs paths which are continuous in different real number intervals, while in [17] terms are only interpreted over single states. Such extension enables us to be able to specify continuous evolutions which are controlled by differential equations with SRML extension. Moreover, SO-HL^{2}TSs extend L^{2}TSs, over which UCTL formulas are interpreted [36, 37]; and in our approach, dTL formulas are interpreted over SO-HL^{2}TSs. Thus, SO-HL^{2}TSs work as the semantic domain for both UCTL and dTL formulas, and can support both the verification of UCTL formulas with UMC model-checker [36] and the verification of dTL formulas with dTL sequent calculus rules [41].

In the approach of transforming from SRML module to SRML based finite automata, we adopt the basic form of DFA [45, 46, 47] such that SRML based finite automata can be seen as an extension of DFA. But SRML based finite automata are more similar to hybrid automata [25] in that they include values of continuous variables in each state and over each transitions. In particular, a SRML based finite automaton is defined over the corresponding SRML service module and the SO-HL^{2}TS over which that service module is interpreted, and each node of the SRML based finite automaton denotes a trace of state, but not a single state as that defined in [25].

7.2 Future prospects

Our work mainly relates to two research domains: service-oriented architecture and hybrid systems. In the former we focus on the formal specification of hybrid transition systems, and in the latter we focus on the method for verifying behaviors of hybrid transition systems. Thus the future prospects of our work lie in the following sections:

7.2.1 Formal specifications in SOA

In SOC, besides the functional behaviors and interfaces of a single service, the discovery and binding of services is another important content, which includes communications between service requirer and service provider, composition of services, the timing problem of asynchronous systems and so on. When coupling hybrid transition systems with these aspects, some interesting ideas are brought about.

In [18], the definitions of requires-interfaces and provides-interfaces are refined and the "*reduct*" operations over service provider and service requirer are defined, and in [54] a formal model *SRMLight* for reconfiguration between service and client is presented. In order to adapt the new framework, the languages in SRML (such as *LS* and *LE*) and the set of behavior constraints of business protocol would need to be enlarged.

In [55, 56], an algebra of discrete timed input/output automata that execute in the context of different clock granularities is introduced. This allows services and clients in the same network reconfigure at run-time. If some elements in such a network act as hybrid transition systems, new problems would possibly arise in the consistency and feasibility check. Moreover, a set of LTL-based orchestration schemes for formalizing the discovery and binding of services is proposed in [57, 58], while in this thesis, the static behaviors of services are expressed by dTL, which is a combination of CTL and DL. When designing services under such schemes, how to fit features based on LTL and features based on dTL together would be another problem.

7.2.2 Methods and techniques for validation

In our work, we adopt dTL as the logic basis for validating hybrid behaviors of SRML service modules. dTL was first brought forward in [41], but different from that defined in [41], we interpret dTL formulas over paths of SO-HL^{2}TSs, over which UCTL formulas can also be interpreted. In such a way, the hybrid behaviors and normal behaviors of SRML service modules are addressed into the same semantic domain.

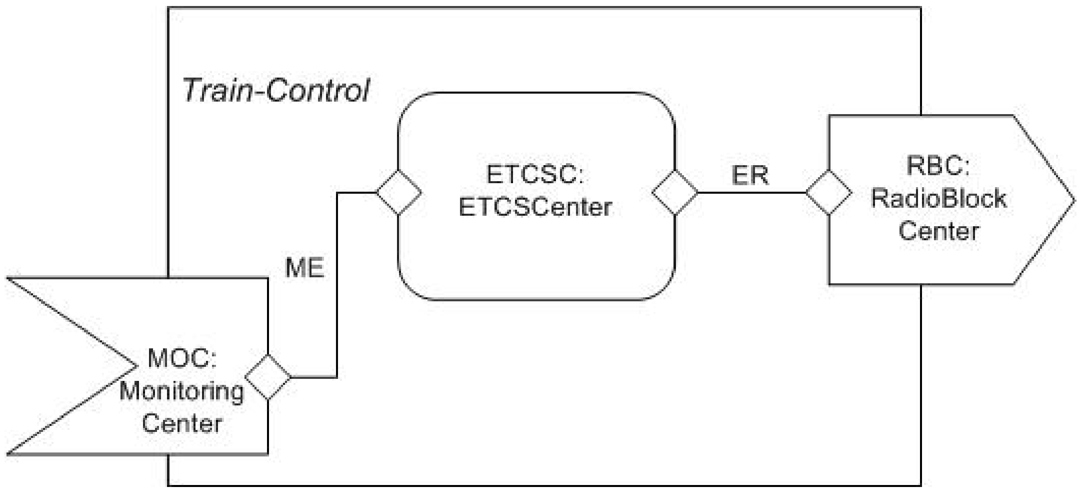
In [59], a new type of logic for validating hybrid systems, named dTL^2, is brought forward. It enlarges the dTL formulas by allowing nested temporalities and can be used for verifying nontrivial temporal properties of hybrid systems. In our future work, to interpret dTL^2 over paths of SO-HL^{2}TSs would be a possible aspect. This will enable us to check if some events happen during the intermediate states along certain traces of states.

Another prospect of our future work would be enriching case studies on different types of complex systems, such as the example of human body exposure to extreme heat studied in [60] and cyber-physical systems studied in [61]. Especially in [62] , an approach of refactoring, refinement and reasoning for cyber-physical systems is provided. When taking similar systems for our case studies, this approach would likely to be adopted.

In this thesis we adopt formal verification to validate the behaviors specified with the hybrid behavior constraint of SRML service modules. However model checking is a more realistic and efficient way for validation. The model checking tools for verifying hybrid systems includes: Kronos [63] and UPPAAL [64] which are used to verify CTL properties of timed automata [65], PhaVer [66] and SpaceEX [67] which compute approximations of the reachable set of hybrid automata with linear dynamics, HSOLVER [68] and Ariadne[69] that manage hybrid systems with nonlinear dynamics, KeYmaera [70] which uses automated theorem proving techniques to verify nonlinear hybrid systems symbolically. Moreover, in [71, 72, 73] differential dynamic logic is brought forward to solve the model checking problems with KeYmaeral, and in [74] the logic HyLTL is brought forward to solve the model checking problems with PhaVer. In our future work, we would be interested in developing a model checking method that can use some of these tools introduced above. Possibly differential dyanmic logic or HyLTL would be adopted in this approach.

Appendix A

SRML specification of *Train-control*



MODULE TRAIN-CONTROL **is**

DATATYPES

sorts: speed, acc
position, time

PROVIDES

RBC: RadioBlockCentre

REQUIRES

MOC: MonitoringCenter

COMPONENTS

ETCSC: ETCSControl

WIRES

MOC MonitoringCenter	**ME**			**ETCSC** ETCSControl
s&r receivePos ✉Pos T	R i_1 i_2	Straight. O(position,time)	S i_1 i_2	**r&s** end ✉currPos currTime

ETCSC ETCSControl	**ER**			**RBC** RadioBlockCentre
rcv MAControl	S	Straight	R	**snd** control
rcv Dec	S	Straight	R	**snd** reduce
rcv moveOn 🔔newMA	S	Straight. I(position)	R	**snd** move 🔔MA

END MODULE

SPECIFICATIONS

BUSINESS ROLE ETCSC **is**

INTERACTIONS

rcv MAControl

rcv Dec

rcv moveOn

🔔 newMA:position

r&s end

✉ currPos:position

currTime:time

ORCHESTRATION

var C:position, V:speed, M:position, C_{dot}:speed, v_0:speed, V_{dot}:acc

initial v_0=100, C=0, t=0, M=500

transition Nego

> **trigger** MAControl🔔?
> **guard** $C_{dot}=v_0$
>
> **effect** $C_{dot}=v_1$

transition Corr

> **trigger** Dec🔔?
> **guard** $C_{dot}=v_1$
>
> **effect** $C_{dot}=V$
> $\wedge$ $V_{dot}=b$

transition Cont

> **trigger** moveOn🔔?
> **guard** $C_{dot}=v_0$
>
> **effect** $C_{dot}=v_0$

transition Stop

> **trigger** end🔔?
> **guard** $V_{dot}=b$
>
> **effect** end✉.currPos=C+M
> $\wedge$ end✉.currTime=t

BUSINESS PROTOCOL MonitoringCenter **is**

INTERACTIONS

s&r receivePos

✉ Pos:position

T:time

BEHAVIOUR

always receivePos🔔.T<L → receivePos🔔.Pos<N

BUSINESS PROTOCOL RadioBlockCentre **is**

INTERACTIONS

snd control

```
    snd reduce
    snd move
       🔔 MA:position
```

INTERACTION PROTOCOL straight is

```
 ROLE A
    snd S
 ROLE B
    rcv R
 COORDINATION
    S ≡ R
```

INTERACTION PROTOCOL straight I(d_1) is

```
 ROLE A
    snd S
       🔔 $i_1$:$d_1$
 ROLE B
    rcv R
       🔔 $i_1$:$d_1$
 COORDINATION
    S ≡ R
    S.$i_1$=R.$i_1$
```

INTERACTION PROTOCOL straight O(d_1, d_2) is

```
 ROLE A
    s&r S
       ✉ $i_1$:$d_1$
         $i_2$:$d_2$
 ROLE B
    r&s R
       ✉ $i_1$:$d_1$
          $i_2$:$d_2$
 COORDINATION
    S ≡ R
    S.$i_1$=R.$i_1$
    S.$i_2$=R.$i_2$
```

END SPECIFICATIONS

Appendix B

Implementation of *Train-Control* module

The implementation of WPS module *TrainControl* with WebSphere Integration Developer (WID) is done following the steps below:

Starting with WID

Start WebSphere Integration Developer and create a new workspace in the intended directory. We will see the *TrainControl* module appears in the Business Integration perspective as showed in Figure . The Business Integration view provides a logical view of the key resources in each module, mediation module and library. Within each project, the resources are categorized by type.

Creating the interface

Create an interface named *TrainControlInterface* for the module, with five operations. Each operation has input or output parameters as shown in Table 6.1 and represents the event that will cause the transition from one state to another in the state machine that is created in the next step.

Creating the state machine

1. Create a new state machine as a component of the *TrainControl* module and name it *ETCSC*. Appoint *TrainControlInterface* as its interface and *Start* as its first operation.

2. Add the input parameter *id* to the correlation blank. As introduced in Section 6.2 correlation defines properties that are used to distinguish one instance of a state machine from another within a runtime environment. Since *id* is a input parameter of type *string* for every operation in *TrainControlInterface*, it can work as a correlation by assigning it with the same value for each operation of the same instance of state machine *ETCSC* at runtime.

3. In the variable blank, create variables as that listed in Table 6.2.

Configuring the state *State1*

1. Rename the automatically generated state to *State1*.

2. Add action *ActSta* to the transition that goes from *InitialState1* to *State1*. In the **property** tab of *ActSta*, choose **details** and add the following Java codes:
   ```
   v0=(float)100;
   ```

C=(float)0;
t=(float)0;
M=Start_Input_MA;
dotC=(float)100;

3. Add entry action *Entry1* (An entry action is an activity which is executed when entering a state) to *State1*. In the **properties** tab of *Entry1*, choose **details** and add the following Java codes:
   if(dotC==v0)System.out.println("true");
   System.out.println("false");

4. Create a self transition (the transition in which the source state and the target state are the same) on *State1*. Choose operation *Continue* form interface *TrainControlInterface* and add to the self transition as an event.

5. Add condition *Condition0* to the self transition. In the **properties** tab of *Condition0*, choose **details** and add the following Java codes:
   if(v1==100)return true;
   return false;

6. Add action *ActCon* to the self transition. In the **properties** tab of *ActCon*, choose **details** and add the following Java codes:
   t=t+(float)3;
   *C=C+3*v0;*
   M=C+M+Continue_Input_MA;

Configuring the state *State2*

1. Create a new state and name it with *State2*.

2. Create a new transition from *State1* to *State2* state. Choose operation *MAControl* form interface *TrainControlInterface* and add to the transition as an event.

3. Add condition *Condition1* to the transition. In the **properties** tab of *Condition1*, choose **details** and add the following Java codes:
   if(v1==100)return true;
   return false;

4. Add action *ActMAC* to the self transition. In the **properties** tab of *ActMAC*, choose **details** and add the following Java codes:
   v1=(float)50;
   t=t+(float)5;
   *C=C+5*v0;*
   dotC=v1;

5. Add entry action *Entry2* to *State2*. In the **properties** tab of *Entry2*, choose **details** and add the following Java codes:

if(dotC==v1)System.out.println("true");
System.out.println("false");

Configuring the state *State3*

1. Create a new state and name it with *State3*.

2. Create a new transition from *State2* to *State3* state. Choose operation *MAControlConfirm* form interface *TrainControlInterface* and add to the transition as an event.

3. Add condition *Condition2* to the transition. In the **properties** tab of *Condition2*, choose **details** and add the following Java codes:
   if(v1==50)return true;
   return false;

4. Add action *ActMACC* to the self transition. In the **properties** tab of *ActMACC*, choose **details** and add the following Java codes:
   b=(float)10;
   C=C+v1(float)4-(float)0.5*b*(float)16;*
   V=v1-b(float)4;*
   dotC=v1-b(float)4;*
   dotV=(float)10;

5. Add entry action *Entry3* to *State3*. In the **properties** tab of *Entry3*, choose **details** and add the following Java codes:
   if(dotC==V&&dotV==b)System.out.println("true");
   System.out.println("false");

Configuring the state *FinalState1*

1. Rename the state *FinalState* with *FinalState1*.

2. Create a new transition from *State3* to *FinalState1* state. Choose operation *End* form interface *TrainControlInterface* and add to the transition as an event.

3. Add condition *Condition3* to the transition. In the **properties** tab of *Condition3*, choose **details** and add the following Java codes:
   if(b==10)return true;
   return false;

4. Add action *ActEnd* to the self transition. In the **properties** tab of *ActEnd*, choose **details** and add the following Java codes:
   End_Output_dis=C;
   End_Output_time=t;

5. Add entry action *Entry4* to *FinalState1*. In the **properties** tab of *Entry4*, choose **details** and add the following Java codes:
   if(End_Output_dis==C&&End_Output_time==t)System.out.println("true");
   System.out.println("false");

   By finishing the steps above we obtain the state machine ETCSC as shown in Figure 6.3. B.1

   Assembling the WPS module

1. Open the *TrainControl* assembly diagram and drag the state machine *ETCSC* to the canvas of the assembly diagram.
2. Add import component *RBC* and export component *MOC* to the canvas of the assembly diagram.
3. Link *RBC* with *ETCSC* and *ETCSC* with *MOC*. the result can be seen in Figure B.1

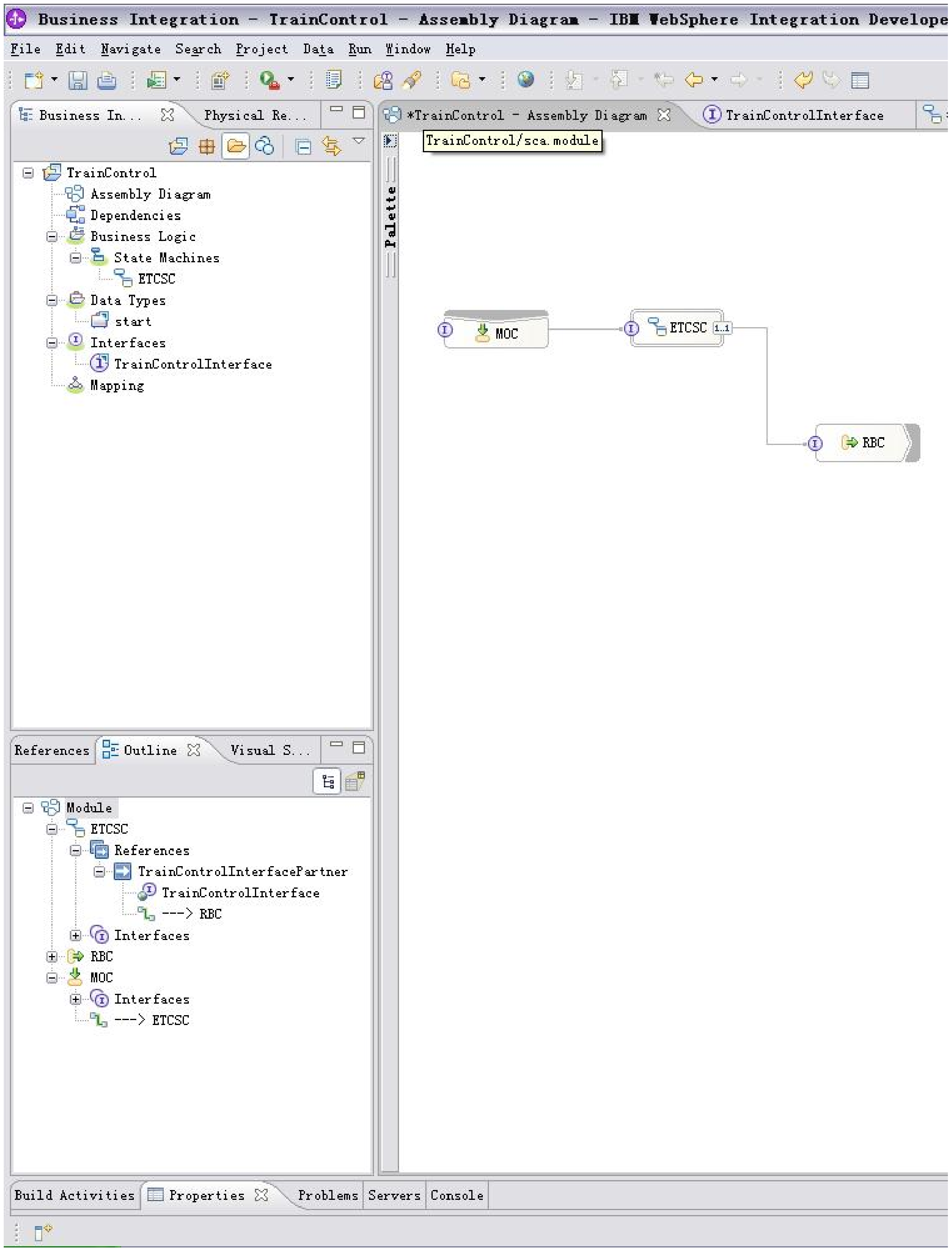


Figure B.1: Assembly Diagram of WPS service module *TrainControl*

Appendix C

Publication of Ning Yu

Ning Yu and Martin Wirsing. A SOC-Based Formal Specification and Verification of Hybrid Systems. Lecture Notes in Computer Science (LNCS), 9463:151-169, 2016. (Own contribution: main author. Supported by co-author mainly on notions in some of the definitions and some explanations.) Cited as [28] in the bibliography. This paper provides the basis for the results of Chapter 4 and Chapter 5.

Bibliography

[1] D. Georgakopoulos and M. Papazoglou. *Service-Oriented Computing*. The MIT Press, Cambridge, Massachusetts, 2009.

[2] D. Krafzig, K. Banke, and D. Slarma. *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall, Professional Technical References, Indianapolis, Ind., 2005.

[3] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D.F. Ferguson. *Web Services Platform Architecture*. Prentice Hall PTR, U.S.A., 2005.

[4] I. Foster and C. Kesselman. The Grid: Blueprint for a New Computing Infrastructure. *Elsevier Series in Grid Computing*, 2004.

[5] http://www.osgi.org. The OSGI Aliance.

[6] http://www.microsoft.com. Microsoft COM (Component Object Model) Technology.

[7] Object Management Group. Common Object Request Broker Architecture (CORBA). http://www.omg.org.

[8] J. Yang. Web Service Componentization. *Communications of the ACM*, 46(10):35–40, 2003.

[9] A. Dhesiaseelan and V. Ragunathan. Web Services Container Reference Architecture (WSCRA). *IEEE*, Proceedings of the International Conference on Web Services:805–806, 2004.

[10] D. Box et al. Simple Object Access Protocol (SOAP)1.1. *W3C note*, 2000.

[11] M.P. Papazoglou and D. Georgakopoulos. Introduction to a Special Issue on Service-Oriented Computing. *Communications of the ACM*, 46(10):24–28, 2003.

[12] T. Andrews et al. Business Process Execution Language (BPEL), Version 1.1. *BEA Systems, IBM, Microsoft, SAP and Siebel Systems*, Technical report, 2003.

[13] A. Arkin. Business Process Modeling Language (BPML). *BPMI.org*, Last call draft report, 2002.

[14] M. Colan. Service-Oriented Architecture Expands the Vision of Web Services, Part 2. *IBM DeveloperWorks*, 2004.

[15] Universal Description, Discovery, and Integration (UDDI). *http://www.uddi.org*, Technical report, 2000.

[16] H. Ossher and P. Tarr. Multi-dimensional Separation of Concerns and the Hyperspace Approach. *Proceedings of the Symposium on Software Architectures and Component Technology*, The State of the Art in Software Development, 2000.

[17] J. A. Abreu. *Modelling Business Conversations in Service Component Architectures*. PhD thesis, University of Leicester, 2009.

[18] J. Fiadeiro, A. Lopes, and J. Abreu. A Formal Model for Service-Oriented Interactions. *Science of Computer Programming*, 77(5):577–608, 5 2012. Copyright 2012 Elsevier B.V., All rights reserved.

[19] Service Component Architecture Assembly Model Specification Version 1.2, July 2011. [online] `http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec/v1.2/csd01/sca-assembly-spec-v1.2-csd01.html`.

[20] Michael S. Branicky. *Introduction to Hybrid Dynamical Systems*. Birkhäuser, Boston, 2005.

[21] P.P. Varaiya. Smart Cars on Smart Roads: Problems of Control. *IEEE Trans. Automatic Control*, 38(2):195–207, 1993.

[22] G. Meyer. Design of Flight Vehicle Management Systems. *IEEE Conf. Decision and Control*, page Plenary Lecture, 1994.

[23] C. Tomlin, G.J. Pappas, and S. Sastry. Conflict Resolution for Air Traffic Management: A Study in Multi-Agent Hybrid Systems. *IEEE Trans. Automatic Control*, 43(4):509–521, 1998.

[24] A. Back, . Guckenheimer, and M. Myers. A Dynamical Simulation Facility for Hybrid Systems. *Hybrid Systems*, pages 255–267, 1993.

[25] Thomas A. Henzinger. The Theory of Hybrid Automata. *LNCS*, 170:256–292, 2000. LICS '96 Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science.

[26] Arjan van der Schaft and Hans Schumacher. *An Introduction to Hybrid Dynamical Systems*. Springer-Verlag, Amsterdam, 2000.

[27] H. Elmqvist, F. Boudaud, J. Broenink, D. Bruck, T. Ernst, P. Fritzon, A. Jeandel, K. Juslin, M. Klose, S.E. Mattsson, M. Otter, P. Sahlin, H. Tummescheit, and H. Vangheluwe. Modelica$^{\mathrm{TM}}$ Version 1, September 1997. `http://www.Dynasim.se/Modelica/Modelica1.html`.

[28] N. Yu and M. Wirsing. A SOC-Based Formal Specification and Verification of Hybrid Systems. *LNCS*, 9463:151–169, 2015.

[29] A. Platzer. Differential Logic For Reasoning About Hybrid Systems. *LNCS*, 4416:746–749, 2007.

[30] K.A. Stroud and Dexter Booth. *Differential Equations*. Industrial Press, Inc., New York, 2005.

[31] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. *LNCS*, 131:52–71, 1981. Proceedings of Workshop on Logics of Programs.

[32] E.A. Emerson. Temporal and Modal Logic. *Handbook of Theoretical Computer Science*, pages 997–1067, 1990.

[33] A. Pnueli M. Ben-Ari and Z. Manna. The Temporal Logic of Branching Time. *Principles of Programming Languages*, 20:207–226, 1983. Proceedings of the 8th Ann. Symp.

[34] E.A. Emerson and E.M. Clarke. Characterizing Correctness Properties of Parallel Programs as Fixpoints. *LNCS*, 85:169–181, 1981. Proceedings of the 7th Internat. Coll. on Automata, Languages and Programming.

[35] L. Lamport. Sometimes is Sometimes "Not never" —on the Temporal Logic of Programs. *Principles of Programming Languages*, pages 174–185, 1980. Proceedings of the 7th Ann. ACM Symp.

[36] Maurice H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An Action/State-Based Model-Checking Approach for the Analysis of an Asynchronous Protocol for Service-Oriented Applications. *FMICS 2007, LNCS*, 4916:133–148, 2008.

[37] Maurice H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. A State/Event-Based Model-Checking Approach for the Analysis of Abstract System Properties. *Science of Computer Programming*, 76 Issue 2:119–135, 2011.

[38] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Massachusetts Institute of Technology, London, 2000.

[39] R. M. Keller. Formal Verification of Parallel Programs. *Communications of the ACM*, 19:371–384, 1976.

[40] T. Henzinger, X. Nicollin, and J. Sifakis. Symbolic Model Checking for Real-Time Sysems. *LICS*, IEEE Computer Society:394–406, 1992.

[41] A. Platzer. A Temporal Dynamic Logic for Verifying Hybrid System Invariants. *Technical report*, (12, AVACS), 2007.

[42] A. Platzer. Towards a Hybrid Dynamic Logic for Hybrid Dynamic Systems. *LICS International Workshop on Hybrid Logic*, (Seattle, USA.), 2006.

[43] A. Platzer. Differential Dynamic Logic for Verifying Parametric Hybrid System. *LNAI*, 4548:216–232, 2007.

[44] Janusz A. Brzozowski. Derivatives of Regular Expressions. *Journal of the Association for Computing Machinery*, 11:481–494, 1964.

[45] D. A. Huffman. The Synthesis of Sequential Switching Circuits. *J. Franklin Inst.*, 257:3-4:161–190 and 257–303, 1954.

[46] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34:5:1045–1079, 1955.

[47] E. F. Moore. Gedanken experiments on sequential machines. *Automata Studies*, Ann. of Math. Studies No.34:129–153, 1956.

[48] M. O. Rabin and D. Scott. Finite Automata and Their Decision Problems. *IBM J. Research and Development*, 3:2:115–125, 1959.

[49] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, U.S.A., 2001.

[50] F. Mazzanti. Umc user guide v3.3. *Technical Report 2006-TR-33, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR.*, 2006.

[51] Johannes Faber. Verifying Real-Time Aspects of the European Train Control System. *Proceedings of the 17th Nordic Workshop on Programming Theory*, pages 67–70, 2005.

[52] Alvaro A. Cardenas, Saurabh Amin, and Shankar Sastry. Research Challenges for the Security of Control Systems. 2008.

[53] Carla Sadtler, Srinivasa Rao Borusu, Sergly Fastovets, Thalia Hooker, Ernese Norelus, Fabio Paone, and Dong Yu. *Getting Started with IBM WebSphere Process Server and IBM WebSphere Enterprise Service Bus, Part1: Development*. IBM Corp., 2008.

[54] J. Fiadeiro and A. Lopes. A Model for Dynamic Reconfiguration in Service-Oriented Architecture. *Software and Systems Modeling*, 12(2):349–367, 5 2013. Copyright 2012 Elsevier B.V., All rights reserved.

[55] B. Delahaye, J. Fiadeiro, A. Legay, and A. Lopes. *A Timed Component Algebra for Service*, pages 242–257. Lecture Notes in Computer Science. Springer, 2013.

[56] B. Delahaye, J. Fiadeiro, Axel Legay, and A. Lopes. *Heterogeneous Timed Machines*, volume 8687 of *Lecture Notes in Computer Science*, pages 115–132. Springer, 2014.

[57] I. Tutu and J. Fiadeiro. A Logic-Programming Semantics of Services. pages 299–313, 2013.

[58] I. Tutu and J. Fiadeiro. Service-Oriented Logic Programming. *Logical Methods in Computer Science*, 11(3):1–38, 8 2015.

[59] J. Jeannin and A. Platzer. dTL^2: Differential Temporal Dynamic Logic with Nested Temporalities for Hybrid Systems. *LNAI*, 8562:292–306, 2014.

[60] M. Fadlisyah, P. Olveczdy, and E. Abraham. Formal Modeling and Analysis of Human Body Exposure to Extreme Heat in HI-Maude. *LNCS*, 7571:139–161, 2012.

[61] A. Rajhans, S.W. Cheng, B. Schmerl, B.H. Krogh, C. Aghi, and A. Bhave. An Architectural Approach to the Design and Analysis of Cyber-Physical Systems. *Electronic Communications of the EASST*, 21, 2009.

[62] S. Mitsch, J.Quesel, and A. Platzer. Refactoring, Refinement, and Reasoning: A Logical Characterization for Hybrid Systems. *LNCS*, 8442:481–496, 2014.

[63] S. Yovine. Kronos: a Verification Tool for Real-Time Systems. *Int. J. on Software Tools for Technology Transfer*, 1(1-2):123–133, 1997.

[64] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. J. on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[65] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[66] G. Frehse. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. *Int. J. on Software Tools for Technology Transfer*, 10:263–279, 2008.

[67] G. Frehse, C. Le Guernic, A. Donze, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable Verification of Hybrid Systems. *LNCS*, 6806:379–395, 2011.

[68] S. Ratschan and Z. She. Safety Verification of Hybrid Systems by Constraint Propagation Based Abstraction Refinement. *ACM Transactions in Embedded Computing Systems*, 6(1), 2007.

[69] L. Benvenuti, D. Bresolin, P. Collins, A. Ferrari, L. Geretti, and T. Villa. Assume-Guarantee Verification of Nonlinear Hybrid Systems with ARIADNE. *Int. J. Robust. Nonlinear Control*, 24:699–724, 2012.

[70] A. Platzer and J.-D. Quesel. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems. *LNCS*, 5195:171–178, 2008.

[71] A. Platzer. Differential Dynamic Logic for Hybrid Systems. *J. Sutom. Reas*, 41(2):143–189, 2008.

[72] A. Platzer. The Complete Proof Theory of Hybrid Systems. *LICS*, IEEE:541–550, 2012.

[73] A. Platzer. A Uniform Substitution Calculus for Differential Dynamic Logic. *LNAI*, 9195:467–481, 2015.

[74] D. Bresolin. HyLTL: a Temporal Logic for Model Checking Hybrid Systems. *EPTCS*, 124:73–84, 2013.