ISTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF INFORMATICS

VERIFYING THE INTERFACE COMPLIANCE OF FEDERATES USING PRE- AND POSTCONDITIONS OF RTI SERVICES

M.Sc. Thesis by
Vijdan KIZILAY

Department : Computer Science

Programme : Computer Science

JANUARY 2010

ISTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF INFORMATICS

VERIFYING THE INTERFACE COMPLIANCE OF FEDERATES USING PRE- AND POSTCONDITIONS OF RTI SERVICES

M.Sc. Thesis by
Vijdan KIZILAY
(704061024)

Date of submission : 25 December 2009
Date of defence examination: 27 January 2010

Supervisor (Chairman) : Assis. Prof. Dr. Feza BUZLUCA (ITU)
Members of the Examining Committee : Prof. Dr. Nadia ERDOĞAN (ITU)
Assis. Prof. Dr. Yunus Emre SELÇUK (YTU)

JANUARY 2010

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

ÇALIŞMA ZAMANI ALTYAPISI (RTI) SERVİSLERİNİN ÖN VE SON KOŞULLARINI KULLANARAK FEDERE ARAYÜZ UYUMLULUĞUNUN GEÇERLENMESİ

YÜKSEK LİSANS TEZİ
Vijdan KIZILAY
(704061024)

Tezin Enstitüye Verildiği Tarih : 25 Aralık 2009
Tezin Savunulduğu Tarih : 27 Ocak 2010

Tez Danışmanı : Yrd. Doç. Dr. Feza BUZLUCA (İTÜ)
Diğer Jüri Üyeleri : Prof. Dr. Nadia ERDOĞAN (İTÜ)
Yrd. Doç. Dr. Yunus Emre SELÇUK (YTÜ)

OCAK 2010

FOREWORD

I would like to express my deep appreciation and thanks for my advisors Assist. Prof. Dr. Feza Buzluca and Dr. Okan Topçu.

I would like to thank Assoc. Prof. Dr. Halit Oğuztüzün for his support during this thesis.

I would like to thank my family, especially my mother for their patiance and support during the preparation of this thesis.

December 2009

Vijdan Kızılay

Computer Science

TABLE OF CONTENTS

Page

ABBREVIATIONS ..... **ix**
LIST OF TABLES ..... **xi**
LIST OF FIGURES ..... **xiii**
SUMMARY ..... **xv**
ÖZET ..... **xvii**
1. INTRODUCTION **1**
1.1 Motivation 1
1.2 Problem and the Approach 3
1.3 Related Work 4
1.3.1 Hugo/RT 5
1.3.2 vUML 5
1.3.3 Model Checking Dynamic and Hierarchical UML State Machines 5
1.3.4 Automated Distributed System Testing: Designing of An RTI Verification System 6
1.3.5 The High Level Architecture Federate Conformance Testing 6
1.4 Thesis Outline. 6
2. BACKGROUND **9**
2.1 High Level Architecture (HLA) 9
2.2 HLA Rules 10
2.3 Interface Specification 10
2.4 Object Model Template(OMT) 12
2.5 Federation Architecture Metamodel (FAMM) 12
2.6 Generic Modeling Environment (GME) 14
2.7 SPIN and Process Meta Language (PROMELA) 14
3. PROMELA IMPLEMENTATIONS **17**
3.1 Processes 17
3.1.1 RTI P-process 17
3.1.2 Federate P-process 22
3.1.3 Communication Between P-processes 24
3.2 Combatting State Space Explosion 26
4. PROMELA CODE GENERATOR **31**
4.1 Example FAM: Strait Traffic Monitoring Simulation 31
4.2 Front End 34
4.3 Back End 38
4.4 Running the PCG 45
4.5 SPIN Verification 47
5. CONCLUSION AND FUTURE WORK **53**
5.1 Future Work 53
REFERENCES **57**
APPENDICES **60**
CURRICULUM VITA **71**

ABBREVIATIONS

HLA : High Level Architecture
FAMM : Federation Architecture Metamodel
FAM : Federation Architecture Model
RTI : Runtime Infrastructure
PROMELA : Process (or Protocol) Meta Language
LSC : Live Sequence Chart
MSC : Message Sequence Chart
UML : Unified Modeling Language
SDL : Script Definition Language
GME : Generic Modeling Environment
FOM : Federation Object Model
FDD : FOM Document Data
OMT : Object Model Template
SOM : Simulation Object Model
BMM : Behavioral Metamodel
HFMM : HLA Federation Metamodel
PCG : PROMELA Code Generator
P-process : PROMELA process
FDD : FOM Document Data
STMS : Strait Traffic Monitoring Simulation
FEDEP : Federation Development and Execution Process

LIST OF TABLES

Page

Table 4.1: PROMELA code mappings of STMS model elements. 46
Table A.1 : HLA methods implemented in the PROMELA model of the RTI. 70

LIST OF FIGURES

Page

Figure 1.1 : Verification process 3
Figure 2.1: Software components in the HLA 10
Figure 2.2 : Federation architecture metamodel structure. 13
Figure 2.3 : Relationship between a federation model and the metamodel.....14
Figure 3.1 : RTI/Federate-initiated service calls..... 18
Figure 3.2 : Publish Interaction Class service in RTI P-Process. 20
Figure 3.3 : Inline construct interactionClassInFDDPreCondition()…… ……...21
Figure 3.4 : Interaction of P-processes (partial view)……22
Figure 3.5 : LSC of Station federate's behavior model in abstract syntax. ... …….. 23
Figure 3.6 : Message data structure used on channel communication..... 24
Figure 3.7 : Preparing message of Subscribe Object Class Attributes..... 25
Figure 3.8 : PROMELA code with one d_step sequence. 27
Figure 3.9 : Reduction on state space with d_step..... 27
Figure 3.10 : State space reduction of all PROMELA code.. 28
Figure 4.1 : Strait Traffic Monitoring Simulation conceptual view. 32
Figure 4.2 : Federation Architecture Modeling Environment (FAME)..... 33
Figure 4.3 : Prechart of the Station federate. 34
Figure 4.4 : LSC of the Station federate declare capability..... 35
Figure 4.5 : Class diagram of prechart..... 36
Figure 4.6 : Activity diagram of prechart traversing 37
Figure 4.7 : Generated PROMELA code of fillObjectInteractionInfo()..... 38
Figure 4.8 : Main thread of the Station federate 39
Figure 4.9 : PROMELA code of the main thread 41
Figure 4.10 : Callback thread of the Ship federate 42
Figure 4.11 : PROMELA code of the callback thread..... 43
Figure 4.12 : Running the PROMELA Code Generator..... 44
Figure 4.13 : Generating PROMELA code..... 45
Figure 4.14 : XSPIN view..... 45
Figure 4.15 : Basic verification options..... 47
Figure 4.16 : Advanced verification options..... 48
Figure 4.17 : Verification output of SPIN 49
Figure 4.18 : Join Federation Execution service call two times 50
Figure 4.19 : Main thread of the Ship federate (partial view) 51
Figure 4.20 : Update Attribute Values call without registering object instance……52

VERIFYING THE INTERFACE COMPLIANCE OF FEDERATES USING PRE- AND POSTCONDITIONS OF RTI SERVICES

SUMMARY

This thesis presents a model checking approach on the compliance of the interface behaviors of federates to the High Level Architecture (HLA) Federate Interface Specification by generating PROMELA models from Live Sequence Charts (LSCs) of federates in a federation architecture model (FAM). FAMM provides a domain specific language and a formal representation for describing the architecture of an HLA compliant federation. A federation architecture model consists of the object models and the behavioral models of participating federates. Currently, the behavioral model of each federate is required to be modeled in the same level of detail as the HLA Federate Interface Specification so as to facilitate standard-compliant code generation. However, this level of detail increases the likelihood of the modelers making mistakes in the following standart. Thus, beyond well-formedness, static checking of the well-behavedness of federate behavioral models is desirable. If it can be shown that all the preconditions of the HLA Runtime Infrastructure (RTI) services used in a behavioral model are satisfiable then we have some assurance that the interface behavior can be compliant to the HLA Federate Interface Specification.

Model checking based procedure which is presented to verify the interface behavior of an HLA federate modeled in FAMM consists of a few steps. Verification is performed automatically by the help of (1) a model interpreter that takes a FAM as input, and generates the PROMELA model of its behavioral part as output, (2) the SPIN model checker that performs model checking given the generated PROMELA model as input and then outputs the verification result in terms of the preconditions that will not hold at run-time.

ÇALIŞMA ZAMANI ALTYAPISI (RTI) SERVİSLERİNİN ÖN VE SON KOŞULLARINI KULLANARAK FEDERE ARAYÜZ UYUMLULUĞUNUN GEÇERLENMESİ

ÖZET

Bu tez federasyon mimari modelini (FAM) oluşturan federelerin Canlı Sıralama Çizelgelerinden (LSCs) PROMELA modellerini üreterek federelerin arayüz davranışlarının HLA Arayüz Spesifikasyonuna uyumluluğu üzerine bir model denetleme yaklaşımı sunmaktadır. Federasyon Mimari Metamodeli (FAMM), Alan Özel Metamodelleme yaklaşımının HLA uyumlu federasyonlarına uyarlanmasıyla federasyon için biçimsel bir gösterim ve uygulama alanına yönelik bir dil sağlamaktadır. FAMM federasyon mimari modelini oluşturan nesne modellerinin ve federasyonu oluşturan federelerin davranış modellerinin modellenmesini sağlayan bir metamodeldir. FAMM'ın kullanıldığı modelleme ortamında standart uyumlu kod üretimini kolaylaştırmak amacıyla her bir federenin davranış modelinin programlama seviyesi detayında modellenmesi gerekmektedir. Ancak bu seviyede detay modelcilerin standarda göre hata yapma olasılığını arttırmaktadır. Bu nedenle iyi bir biçimin yanında, federelerin davranış modellerinin anlamsal kavramının statik olarak denetlemesi istenir. Eğer bir davranış modelinde kullanılan HLA RTI servislerinin tüm ön koşullarının karşılanabildiği gösterilebilirse, arayüz davranışının HLA Federe Arayüz Spesifikasyonuna uyumluluğu konusunda biraz güvenceye sahip olabiliriz.

FAMM ile modellenmiş bir HLA federesinin arayüz davranışının geçerlenmesi için sunulan model denetleme tabanlı prosedür birkaç adımdan oluşmaktadır. Geçerleme işlemi otomatik olarak şu işlemler yardımıyla gerçekleştirilmektedir: (1) Federasyon mimari modelini girdi olarak alan bir yorumlayıcı modelin davranış kısmının PROMELA modelini çıktı olarak üretmektedir, (2) SPIN model denetleyicisi girdi olarak aldığı PROMELA modeli üzerinde model denetleme işlemini gerçekleştirir ve geçerleme sonuçlarını çalışma zamanında karşılanamayacak ön koşullar açısından sunar.

1. INTRODUCTION

The High Level Architecture (HLA) federates are simulation components that communicate with each other via a Runtime Infrastructure (RTI) in compliance with the HLA standard. Functional interfaces between federates and the RTI are defined by the IEEE 1516.1 Federate Interface Specification [1]. The RTI is essentially a middleware providing distributed simulation services, such as federation management, declaration management, object management, ownership management, to federates.

Verifying the interface compliance of the interface behavior model of federates to the HLA Interface Specification provides federate developers to capture the errors before runtime. This early detection prevents generating faulty interface behavior code in the federate code. The interface behavior of a federate, which means the interaction of the federate with the RTI (through the HLA service interface), can be specified using the Federation Architecture Metamodel (FAMM) [2]. FAMM provides a domain specific language and a formal representation for describing the architecture of an HLA compliant federation. A distinctive feature of FAMM is the behavioral description of federates based on Live Sequence Charts (LSCs) [3] and Message Sequence Charts (MSC)s [4].

In the context of this study, an approach is developed for verifying the interface behavior of federates by interpretting LSCs in FAM. If the federation designer models faulty behaviors, this verification process reveals errors in the model. This approach is important because it precludes the runtime errors depending on the interface behavior of federates that can occur while running the federate code generated from FAM and saves time for the federate developers.

1.1 Motivation

FAMM supports automatic code generation [5]. After constructing the FAM (conforming FAMM), the code generator generates the base code for each federate.

In order to successfully generate the code, a well-formed federation architecture model and a well-behaved interface behavior model are both necessary. Well-formedness, being a syntactical notion, is guaranteed by the conformance of a FAM to FAMM. In the context of this work, well-behavedness of the interface behavior means that the use of the RTI services in a FAM is in accordance with the HLA Federate Interface Specification in the sense that the preconditions of each invoked service are satisfiable when the federate runs.

In order to support the accordance with the HLA Federate Interface Specification, the interface behavior of federates must be checked during the modeling phase. We perform a static analysis to check that the preconditions of each RTI service invoked in the behavioral models (i.e. the behavioral part of a FAM) in a federation architecture are satisfiable or not. Relevant preconditions must be satisfied for an RTI service to fulfill its function. Note that a precondition of an RTI service can be equivalent to (or imply) a postcondition of some other RTI service. Relevant postconditions are generated as a result of performing an RTI service successfully (i.e. with no exception).

Currently, the behavioral model for each federate is required to be modeled in utmost detail with respect to the HLA Federate Interface Specification. The modeling errors are expected because of modeling the interface behavior of federates in this level of detail. The hand-crafted detailed behavioral models may contain error-prone behavior for a federation execution according to the HLA Federate Interface Specification and HLA Federation Rules Specification [6]. For example, the situation of a federate sending an interaction before publishing it is an behavior fault and should be detected. If the federation designer mistakenly models the behavior of sending an interaction before publishing it, then the generated code will not run properly. Therefore, verifying the interface behavior of federates during the modeling phase will provide some degree of assurance of the proper behavior before the automatic code generation is attempted. Detecting errors during modeling phase allows the federate developers to save time.

1.2 Problem and the Approach

The HLA Federation Development and Execution Process (FEDEP) [7] Model defines the activities necessary to construct HLA federations. The HLA FEDEP consists of the steps: (1) defining federation objectives, (2) developing federation conceptual model, (3) designing federation, (4) developing federation, (5) integrating and testing federation, (6) executing federation and preparing results. Federation design is one of these activities that takes place during the life cycle of a federation. Federation design triggers automatic code generation which is the federation development step. When the automatic code generation from a model is concerned, correctness of the model becomes an important issue. If model checking is applied to the model well, the generated code will run so successful.The detailed interface behavior models of federates in a FAM are modeled manually. Therefore, modeling interface behavior requires to be dominant on the interface specification. So the generated code of the interface behavior is likely to contain errors. In this context, we study the verification of the interface behavior models of federates in FAM in order to provide code, which runs succesfully.

The verification approach of our study consists of the steps illustrated in Figure 1.1.

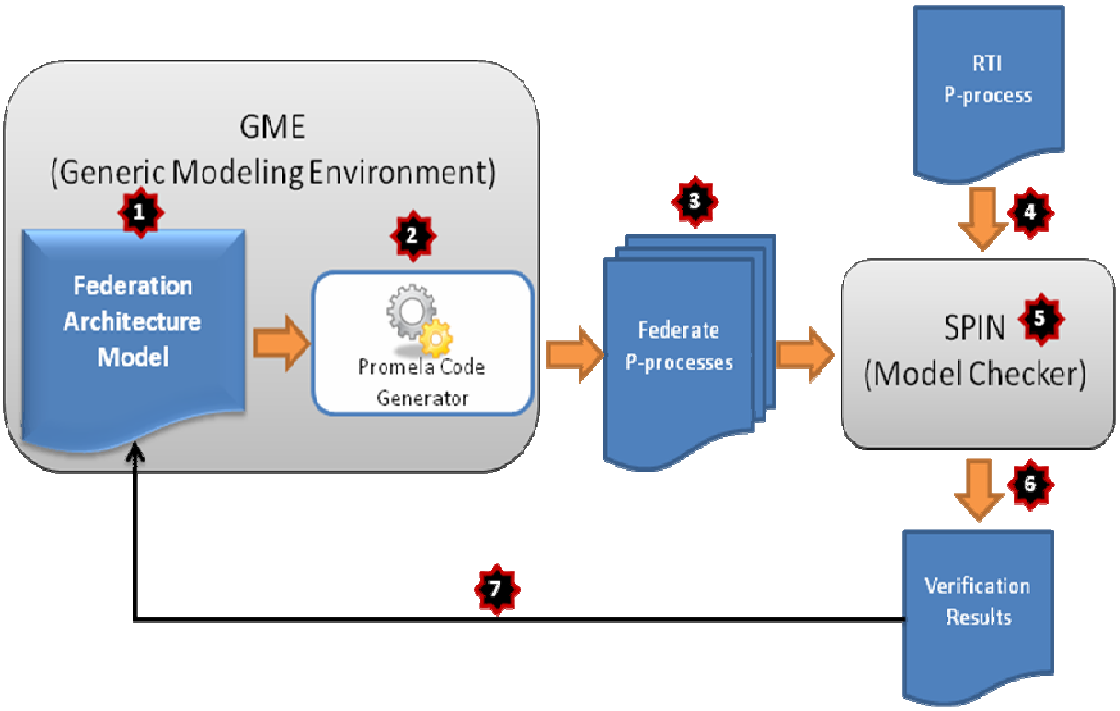

Figure 1.1 : Verification process

(1) The federation designer constructs the FAM. (2) The PROMELA Code Generator (PCG), which takes the FAM as input, is executed. (3) PCG generates a *Federate PROMELA process (P-process)* for each federate. The Federate P-process is the PROMELA code of the interface behavior of each federate. (4) The generated P-processes and the HLA Federate Interface Specification in PROMELA, called *RTI P-process*, are supplied to SPIN as inputs. The RTI P-process was coded once at the beginning of our study. (5) In this step, the federation designer can configure the settings of SPIN. (6) SPIN presents the verification results and then the federation designer interprets the results. (7) The federation designer makes corrections on the model, if necessary.

The model checker may report that a precondition of a method call is not satisfiable. Considering the input behavior model, this result can only be due to a missing method call that would establish the precondition. The generated code, then, would certainly raise an exception in that method call at run time. Thus, the corrective action of the federation designer would be to supply the prior method call that was missing. The result that the precondition is satisfiable, though, does not guarantee that it will indeed be satisfied in every run of the federate. This is because model checking ignores the possible values that the method parameters can take at run time.

1.3 Related Work

The studies represented in [8, 9, 10] applies model checking on UML state machines. State machines are transformed to PROMELA model and verified with SPIN. The interaction of objects and their state changes are verified. In our study, the interface behaviors of federates, based on LSCs, are transformed to PROMELA model. The interaction between the federates and the RTI is verified by using pre and postconditions of the RTI services. States in UML state machines maps to states of federation or federates in our study. The events in UML state machines corresponds to the RTI services in our study.

The study in [11] is about RTI verification. This study does not make verification on a model. The approach in [11] is for testing the functional behavior of RTI. Preconditions and postconditions of the RTI services are used to set the test requirements. The study [11] proposes to verify the compliance of an implemented

RTI to the interface specification. The difference with our work is that we aim to verify the compliance of federates' models to the interface specification.

Another project [13] is about HLA conformance test. Conformance to the interface specification of an implemented federate is tested. The project aims to test the conformance after the federate is implemented. However, we verify the interface compliance during the modeling phase.

The studies mentioned above are summarized in the following sections.

1.3.1 Hugo/RT

Hugo/RT is a UML model translator tool developed at the University of Munich provides model checking, theorem proving, and code generation. UML provides state machines and collaborations to model the dynamic behavior of systems. Hugo/RT verifies automatically whether the interactions expressed by a collaboration can indeed be realized by a set of state machines. This tool transforms state machines into a PROMELA model and collaborations into sets of Büchi automata. The model checker SPIN verifies the model against the automata. The approach is based on a dynamic computation of Statechart behavior rather than a pre-determined, static calculation of possible state transitions in response to input events [8] .

1.3.2 vUML

vUML is a tool that automatically verifies UML models where the behaviour of the objects is described using UML Statecharts diagrams [9] . The tool uses SPIN model checker to perform the verification.. The developers of the tool aim to make vUML automatic and transparent to the designer as possibble. The distinctive feature of the tool is that if an error is found during the verification, the tool creates a UML sequence diagram showing how to reproduce the error in the model.

1.3.3 Model Checking Dynamic and Hierarchical UML State Machines

The researchers of the project develops an approach to check UML state machines. This work focuses on a UML subset for protocol models [10]. The tool development aim is to find errors in protocols communicating using asynchronous message passing. The protocols are modeled using UML class diagrams and state machines.

1.3.4 Automated Distributed System Testing: Designing of An RTI Verification System

This project involves testing the Run Time Infrastructure [11]. It is not a model checking project but has similarities with our work in terms of RTI services verification. It is a verification system that focuses on testing of an RTI implementation. Testing of an RTI focuses on the requirements and behavior directly evident in the HLA Interface Specification. The developed Verifier consists of a Script Definition Language (SDL) to specify test scripts; an application executive controller and SDL interpreter to parse and execute scripts; test federates to connect and interact with an RTI under test; and a database to maintain requirements, tests, and test results [11]. The person who will use the Verifier must firstly understand the RTI services defined in the HLA Interface Specification and then determine a set of test requirements for entry into the system. These test requirements form the basis for generation test scripts. Preconditions and postconditions of the RTI services are used to set the test requirements. The application of this RTI Verification system can be found in [12].

1.3.5 The High Level Architecture Federate Conformance Testing Process

This study represents HLA Compliance process. The process has two parts: Conformance and Certification. HLA Conformance testing ensures that a federate performs in accordance with the Interface Specification and Object Model Template standarts, per the HLA compliance checklist [13].

Certification is the process of validating that a federate has been tested for conformance. This means that once a federate has completed conformance testing, the test results must be validated before the federate can be certified as “HLA Compliant” [13].

1.4 Thesis Outline

The preceding sections of this section presents the problem tackled in this thesis, the purpose of the study, the approach and the related work. The next sections are broken down as follows:

Section 2 gives information about the HLA standarts and tools used during the study.

Section 3 explains the PROMELA model of RTI and PROMELA models of interface behaviors of federates in a FAM. It presents the solutions for the interaction between the PROMELA models of the RTI and federates. State space explosion problem encountered during the study and the improvements made on the PROMELA code are described at the last sub-section.

Section 4 describes PROMELA code generation from live sequence charts.

Section 5 outlines the results achieved as a result of this study.

Appendix A.1 presents the PROMELA model of the RTI.

Appendix A.2 presents the HLA methods implemented in the PROMELA model of the RTI.

2. BACKGROUND

In this section, background information of the thesis is given to make the reader familiar to the terminology used in the following sections. Firstly HLA standart and its components are introduced. Secondly, the structure of FAMM, which is used for federation architecture modeling in this study, and its facilities are described which is used for federation architecture modeling in this study. Thirdly, Generic Modeling Environment (GME) is introduced. GME provides a federation architecture modeling environment, if FAMM is invoked as base paradigm. Fourthly, SPIN model checker, which we use to check our federation architecture model, to which we transform PROMELA model, is described.

2.1 High Level Architecture (HLA)

The HLA is a common framework that supports simulations composed of different simulation components. For instance, you might have simulations of several different manufacturing machines and material-handling machines. The HLA helps you create a factory floor simulation from the pieces [14].

The simulation components compose a federation. Each simulation component that is a component of a federation is called federate. A federation contains the following elements:

- Underlying software infrastructure called the Runtime Infrastructure
- A common object model for the data exchanged between federates in a federation, called the Federation Object Model (FOM)
- Some number of federates

A federate is a member of a federation, one point of attachment to the RTI. A federate could represent one platform, such as a cockpit simulator or could represent an aggregate simulation, such as an entire national simulation of air traffic flow.

Federates and RTI are software. The relationship of the software components is depicted in Figure 2.1. Federates are shown in the figure as simulations, surrogates for live players, or tools for distributed simulation. From the perspective of the HLA, a federate is defined by its single point of attachment to the RTI [14].

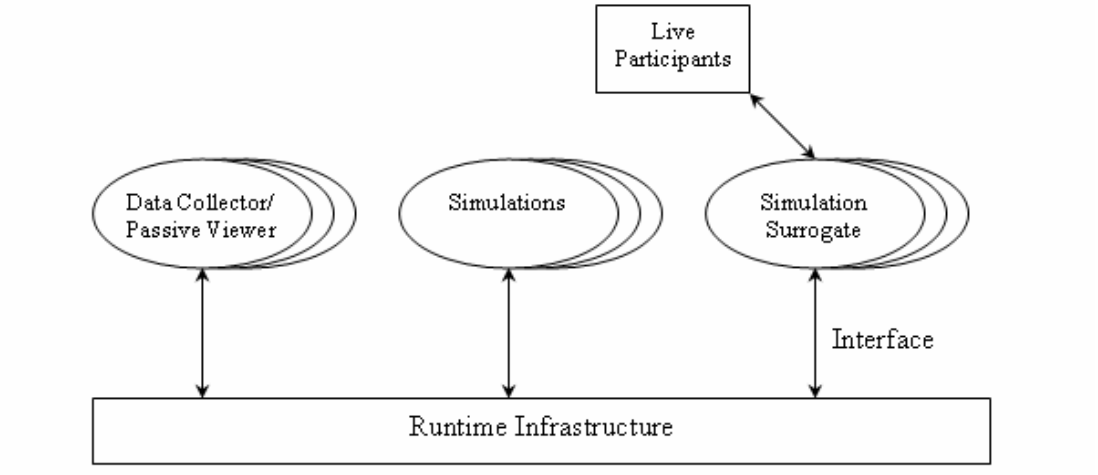

Figure 2.1 : Software components in the HLA [14]

The HLA consists of three components:

- Federation Rules
- Interface Specification
- Object Model Template (OMT).

2.2 HLA Rules

The HLA Rules are principles and conventions that must be followed to achieve proper interaction of federates during a federation execution [14]. They are design principles for the Interface Specification and Object Model Template. The HLA rules are divided into two groups consisting of five rules for HLA federations and five rules for HLA federates. The rules are found in [14].

2.3 Interface Specification

The interface specification defines a stantart for the interface between federates and the Run-Time Infrastructure (RTI). The RTI is software that allows a federation to execute together. The interface between the RTI and federates is standardized. Different implementations of RTI are found in the software market.

The interface specification is divided into 6 management areas. The areas are:

- Federation Management: Federation management services manage a federation by defining a federation execution in terms of existence and membership and by accomplishing federation-wide operations. To define a

federation, there are services to create a federation execution and to permit a federate to join the execution or resign from it.

Federation-wide operations include the coordination of federation saves (checkpoints) and restores.There are also services to allow a federation to define and meet a federation-wide synchronization point [14].

- Decleration Management: The HLA is characterized by an implicit-invocation style of data exchange. Federates don't send data to other federates by name; they make it available to the federation, ant the RTI ensures its delivery to interested parties. The decleration management services are the way federates declera their intent to produce (publish) or consume (subscibe to) data. The RTI uses these declerations for routing data, transforming data, and interest management [14].
- Object Management: Object management services are those used for the actual exchange of data. A federate uses services from this group to send and receive interactions. These services are also used to register new instances of an object class and to update its attributes. Other federates will have services from this group invoked on them to receive interactions, discover new instances, and receive update of instance attributes [14].
- Ownership Management: The ownership management services in the RTI implement the HLA's notion of responsibility for simulating an entity. They allow that responsibility to be shared or transferred among federates [14].
- Time Management: The RTI's time management services allow each federate to advance its logical time in coordination with other federates. And, they control the delivery of time-stamped events so the federate need never receive events from other federates in its "past" [14].
- Data Distribution Management: Data distribution management (DDM) services control the producer-consumer relationships among federates. Whereas the decleration management services manage those relationships in terms of interaction and object classes, DDM manages in terms of object instances and abstract routing spaces. DDM provides powerful tools to refine producer-consumer relationships [14].

2.4 Object Model Template (OMT)

OMT [15] provides a common framework for HLA object model computation. The OMT defines the Federation Object Model (FOM), the Simulation Object Model (SOM) and the Management Object Model (MOM). Every federation has a FOM. The OMT prescribes the allowed structure of every FOM. The OMT is the meta-model for all FOMs [14].

2.5 Federation Architecture Metamodel (FAMM)

FAMM provides a domain-specific language for the formal representation of the HLA-compliant federation architectures [2]. Federation Architecture Model (FAM) is the main portion of the federation design documentation. Federation design for HLA based distributed simulations includes the following activities:

- Forming HLA Object Model
- Specifying the behaviors of participating federates so that they can fulfill their responsibilities within the federation

FAMM consists of two main sub-metamodels (Figure 2.2). These are the Behavioral Metamodel (BMM) and the HLA Federation Metamodel (HFMM). The BMM is for specifying the observable behaviors of the federates and the HFMM for defining both the HLA Federation Object Model (FOM) and the service interface.

HLA Object Metamodel provides HLA specific data model for the behavioral models.

Federation Structure Metamodel is defined for modeling the federation structure. This metamodel provides the federation designer to define a federation and its participating federates, and to connect them to their respective FOM and SOMs.

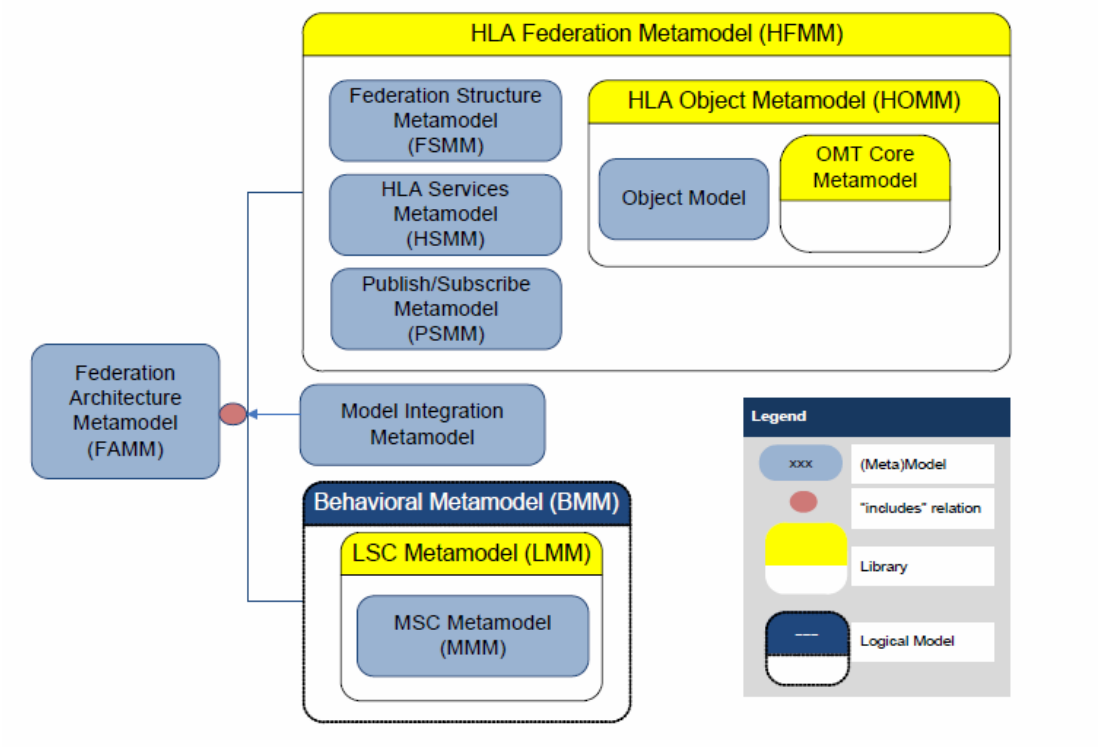

Figure 2.2 : Federation architecture metamodel structure [2]

The HLA Federate Interface Specification defines the standart services of and interfaces to the HLA RTI. HSMM provides the model elements necessary to model the HLA services interface.

BMM encompasses the Live Sequence Chart (LSC) Metamodel (LMM), which is extended from the Message Sequence Chart (MSC) Metamodel (MMM). BMM provides an abstract syntax for specifying the observable behaviors of a federate. The observable behaviors of a federate are represented by means of LSCs, specialized for HLA federates. Specialization involves, in essence, formulating the RTI methods as MSC/LSC messages and integrating the HLA Object Model as the data language of MSC/LSC. Initially, MSC is formalized as the basis of the behavioral metamodel, and then LSC extensions are added on top of the MSC metamodel [2].

A federation architecture model conforming to FAMM is depicted on Figure 2.3. A federation architecture encompasses an object model and LSCs for each participating federate. The observable behavior of a federate contains its interaction with the RTI and with other simulation entities. The observable behavior behavior of a federate is modeled as LSCs. FOM is described in conformance with HLA Object Metamodel and Federation Structure Metamodel. Each participating federate's behavior is

modeled conforming to the behavioral metamodel and the HLA Services Metamodel [2].

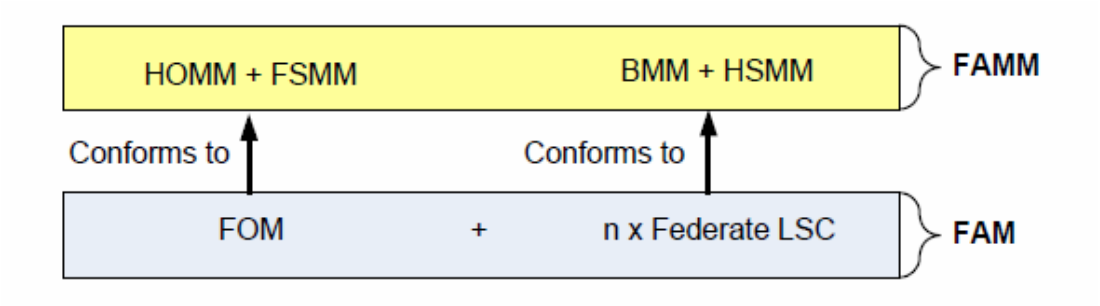

Figure 2.3 : Relationship between a federation model and the metamodel [2]

2.6 Generic Modeling Environment (GME)

The Generic Modeling Environment (GME 7) is described as "domain-specific, model-integrated program synthesis tool for creating and evolving domain-specific, multi-aspect models of large-scale engineering systems" in [16]. GME is an open source tool that developed and maintained by Institute for Software Integrated Systems at Vanderbilt University.

The GME is a configurable, which means it can ben programmed to work with vastly different domains. GME provides a federation architecture modeling environment (FAME), once FAMM is invoked as base paradigm. Using FAME, one can construct a federation architecture model [2].

The GME with FAMM base paradigm is used for modeling a federation architecture in this thesis. The GME BON2 application interface (API) provides a quick and easy way to walk on the input model. The C++ version of the BON2 API is used to traverse input federation architecture model before PROMELA code generation.

More information about the GME and its concepts can be found in [16].

2.7 SPIN and Process Meta Language (PROMELA)

SPIN is an efficient verification system for models of distributed software systems. It has been used to detect design errors in applications ranging from high-level descriptions of distributed algorithms to detailed code for controlling telephone exchanges [17]. SPIN takes PROMELA (Process or Protocol Meta Language) model

as input. The syntax of PROMELA is like C. Concurrent processes can be created dynamically with the language.

SPIN can perform random or interactive simulations of the input model system or it can generate a C program that performs a fast exhaustive verification of the system state space. During simulations and verifications SPIN checks for the absence of deadlocks, unspecified receptions, and unexecutable code [18].

In this thesis XSPIN [19] tool is used to run SPIN. XSPIN is the graphical interface to SPIN. The tool is independent from SPIN. It provides SPIN commands to the user from the interface. It combines and executes SPIN commands in the background, in response to user inputs from the graphical interface.

3. PROMELA IMPLEMENTATIONS

The representatives of RTI and interface behaviors of federates in PROMELA are represented in this section. State space explosion problem faced on the PROMELA models of RTI and interface behaviors of federates and improvements made on the PROMELA code are presented in this section.

3.1 Processes

RTI and interface behavior of federates are transformed to PROMELA as processes. Processes are the basic execution units of the PROMELA models. A SPIN model is used to describe the behavior of systems of potentially interacting processes: multiple, asynchronous threads of execution [20]. RTI P-process contains the RTI services specified in the HLA Federate Interface Specification and it models the behavior of an RTI. Federate P-processes are the interface behavior models of each federate in FAM. Figure 1.1 depicts the P-processes which are given as input to the SPIN model checker.

The interface behavior of a federate specified in a FAM is represented as the message exchanges between the RTI P-process and the federate P-process. The RTI P-process is implemented once and for all representing the interfaces of the RTI services. Further, for each federate, the interface behavior part of the FAM is automatically transformed into the federate P-process by the PCG.

3.1.1 RTI P-Process

The RTI P-process represents the RTI services specified in the HLA Federate Interface Specification including the pre- and postconditions. It models the behavior of an RTI (e.g. keeping information and federate states, sending messages to the federates, etc). Interface behavior of a federate and the behavior of the RTI is exactly defined in the interface specification.

An RTI service definition consists of;

- service name,
- supplied/returned arguments,
- pre- and postconditions,
- exceptions,
- and a semantic description.

The service calls made by the RTI are referred as `RTI-initiated` and the RTI services called by the federate are referred as `federate-initiated` as depicted in Figure 3.1.

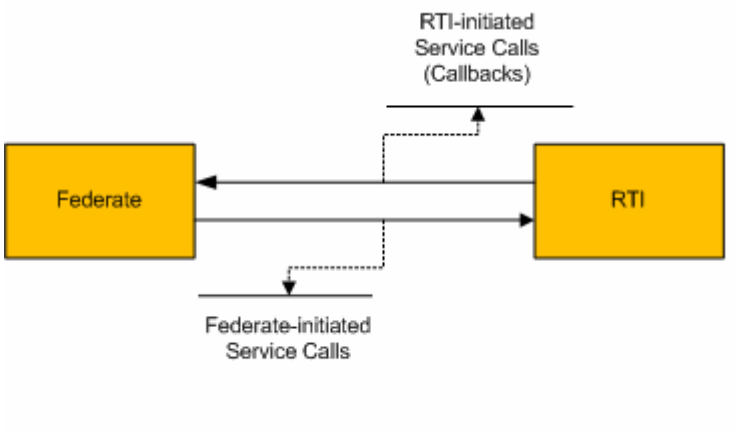

Figure 3.1 : RTI and Federate-initiated service calls

There are two methods developed for sending RTI-initiated messages. In the first method, the RTI-initiated service calls are triggered by the RTI P-process automatically (i.e. the RTI P-process itself decides when to send a message). Therefore, the RTI P-process is referred as automatic RTI. When the federate makes a service call, the RTI P-process receives this call and handles the preconditions and postconditions of the service. If there is an RTI-initiated service defined in the HLA Federate Interface Specification which must be sent after coming service call, then the RTI-initiated service call is made by the RTI P-process to the necessary federates. Checking the preconditions of the RTI-initiated service calls and implementation of the postconditions are made by the RTI P-process.

The second method, can be where a federate P-process triggers the RTI P-process to send a message (i.e. an RTI-initiated callback method). Thus, the federate-initiated and the RTI-initiated messages are triggered by the federate. Messages in the

interface behavior of a federate, are all enumerated when they are transformed to the PROMELA model. `Message type`, which determines whether the message is RTI-initiated or federate-initiated, is also hold in the PROMELA model. For example, if a Ship federate sends a `Request Federation Save` message to the RTI (i.e. a federation execution), then `Initiate Federate Save` message must be sent to the Ship federate and other participating federates. Assume that the sequence number of Request Federation Save message is 10. The RTI P-process controls whether the 11th message is RTI-initiated or not, if so, it sends the 11th message to the Ship federate.

The first method is decided to be used. Because (1) In the second method, the RTI-initiated messages are only sent to the federate, which sent the previous federate-initiated message. Although RTI-initiated messages must be sent to all necessary federates according to the HLA Interface Specification, the RTI-initiated messages are not sent to all necessary federates. RTI P-process checks the preconditions of the message sent by a federate and constitutes the postconditions for only sending federate. (2) When the model elements like `Loop` structures or `Parallel` structures are needed to transform to the PROMELA code, this approach remains inadequate because of message numbers and RTI-initiated messages needs to be handled dynamically for each federate. This situation brings extra effort for transformation of RTI-initiated service calls to PROMELA model. (3) The advantage of the first method is that the federate P-processes do not have to track the RTI-initiated messages. The RTI P-process sends the RTI-initiated messages automatically. Federate P-processes only receive the messages and handle the events that must occur after the received RTI-initiated message.

The RTI P-process is implemented as generic (i.e. not specific to a FAM), instead of generating it each time according to a given FAM. Implementing a generic RTI P-process beforehand is simplified the PCG implementation because, now, the code generator (i.e. the PCG) must handle only the federate part.

In the first method, all information (e.g. the federate states, published objects, subscribed objects information) about the federates are handled in the RTI P-process. The information of federates is updated only in the RTI P-process. An example of a federate-initiated service `Publish Interaction Class` PROMELA implementation can be seen in Figure 3.2. `Publish Interaction Class`

service has five preconditions defined in the interface specification document. These are:

- PreCon-1: The federation execution exists.
- PreCon-2: The federate is joined to that federation execution.
- PreCon-3: The interaction class is specified in the FOM Document Data (FDD).
- PreCon-4: Save not in progress.
- PreCon-5: Restore not in progress.

```
::(msg.msgType == PublishInteractionClass) ->
d_step{
 federationExecExistPreCondition();// PreCon-1
 isJoinedPreCondition(fedpid); // PreCon-2
 interactionClassInFDDPreCondition(); //PreCon-3
 saveNotProgressPreCondition(); //PreCon-4
 restoreNotProgressPreCondition(); //PreCon-5
 if /*Implement postconditions if preconditions are
provided*/
 ::(mPreConditionResult == true  ) ->
   publishInteractionPostCondition();
 ::else ->
 fi;
}
```

Figure 3.2 : `Publish Interaction Class` service in the RTI P-Process

Preconditions and postconditions are handled as inline constructs in the RTI P-process. Because PROMELA does not have functions. Inline constructs provide statements to group together and to be used in other places of the program.

`interactionClassInFDDPreCondition()` is an example inline construct called like a procedure in RTI P-process. Inline constructs of preconditions and postconditions are named with precondition and postcondition suffixes. `interactionClassInFDDPreCondition()` can be seen in Figure 3.3

```
inline interactionClassInFDDPreCondition() {
 d_step{
 result = false;
 k =0;
 do
 ::if
  ::( k == INTERACTIONNUM ) -> break
  ::else  ->
  if
  ::(interactionFDDArray[k].mClass  ==  msg.mClass  )  ->
result = true;
   break
  ::else
   fi;
  fi;
  k++
  od;
  if
   ::( result == false ) ->
    printf("Interaction Class = "); printm(msg.mClass);
    printf("is not specified in FDD\n");
    mPreConditionResult = mPreConditionResult && false;
k = -1;
   ::else->
    printf("Interaction Class = ");printm(msg.mClass);
    printf(" is specified in FDD\n");
  fi;
 }   }
```

Figure 3.3 : Inline construct `interactionClassInFDDPreCondition()`

Figure 3.4 depicts the interaction of P-processes. Federates seen in the Figure 3.4 makes `Create Federation Execution` and `Join Federation Execution` service calls for joining to the federation initially. Ship federate publishes the interaction class `RadioMessage` by calling the service `Publish Interaction Class` and Station federate subscribes to the interaction class `RadioMessage` by calling the `Subscribe Interaction Class` service. Ship federate sends interaction and then RTI P-process checks if there are any federates subscribed to the interaction class. Station federate is subscribed to the interaction class so RTI P-process makes `Receive Interaction` service call to the callback process of Station federate. A joined federate invoking the `Send Interaction` service shall not receive the induced `Receive Interaction` service invocation, regardless of the subscription/region situation [1]. This rule and similar rules are implemented in RTI P-process.

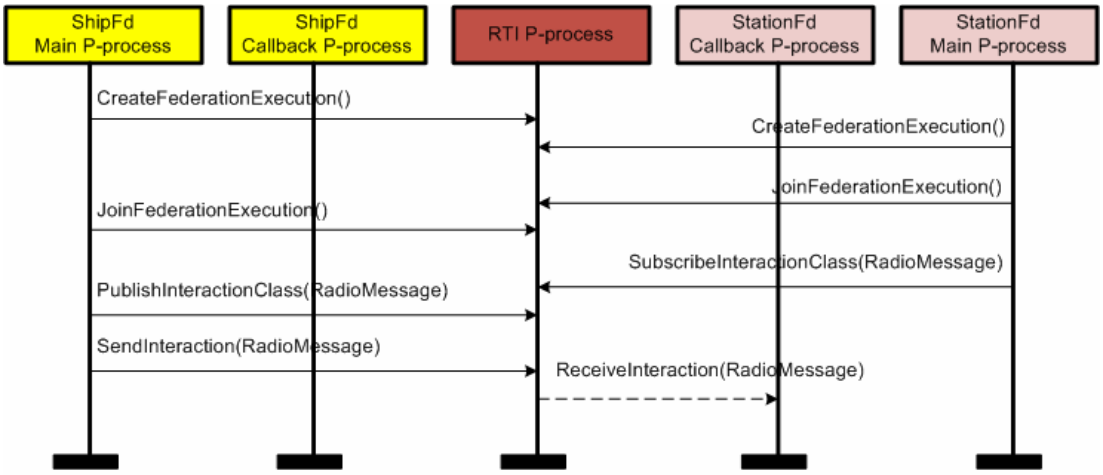

Figure 3.4 : Interaction of P-processes (partial view)

The code of the RTI P-process is presented in Appendix A.1.

3.1.2 Federate P-process

The PROMELA model of federate interface behavior is generated dynamically according to the behavioral model of each federate. The PCG generates the PROMELA code of the interface behavior of each federate by making static analysis of the federation architecture model. Thus, the PROMELA models of the federates are FAM specific and are created dynamically for each federate model.

Federates of a federation are transformed to PROMELA model as processes. Federates run simultaneous and asynchronous. Some of the federate-initiated service calls trigger the RTI P-process to make a service call. Preconditions and postconditions of the RTI-initiated services are handled by the RTI P-process in the PROMELA model. For example, after the `Request Federation Save` service call has been made to the RTI P-process, the RTI P-process must make the `Initiate Federate Save` service call to other federates according to the interface specification. In our model, the RTI P-process checks the preconditions for all the participating federates and then forms the postconditions of `Initiate Federate Save` service for each federate in the federation.

The body part of the federate typically consists of the main thread and callback thread that run parallel to each other as seen in Figure 3.5. The callback thread handles the RTI-initiated messages which may come at any time during the simulation. The main thread is the main simulation loop and it sends the federate-initiated messages to the RTI. The parallel structures are transformed (`par` in LCS jargon) from FAM to PROMELA model as separate processes. `Par` structure is represented as main process and callback process in the PROMELA model. Main and callback processes contain the interface behavior of FAM.

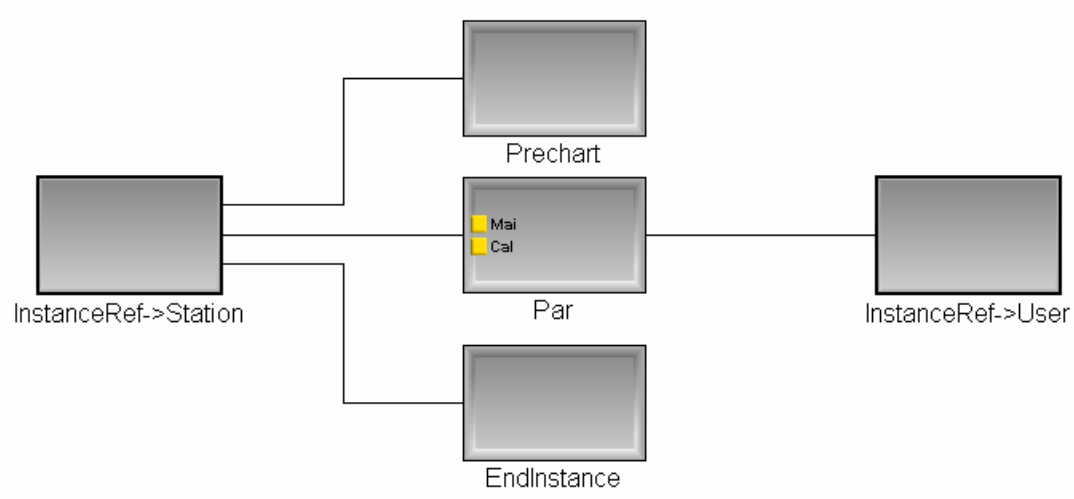

Figure 3.5 : LSC of Station federate's behavior model in abstract syntax

Callback process waits for the messages that come from the RTI P-process. Communication between the RTI P-process and a callback P-process is done with a callback channel. Callback channel of each federate is hold in an array named `callbackChannelArray`. The callback channels are accessed according to the

process identities. After receiving an RTI-initiated message from the RTI P-process, if federate-initiated RTI services are modeled, the callback process sends the federate-initiated messages to the RTI P-process. FAMM contains many model elements to model the LSCs of FAM. FAMM model elements are mapped to the PROMELA structures in the process of transformation from FAM to the PROMELA model. The transformation is done automatically using the PCG. Transformation and the generator is explained in Chapter 4.

3.1.3 Communication Between P-processes

The RTI P-process and the federate P-processes communicate via rendezvous channels. RTI P-process listens the channel. Here we used rendezvous channel due to state space explosion (explained in section 3.3). A channel declared with a capacity of zero is a rendezvous channel [21]. This means that if a process wants to write to the channel but the channel is full then the process has to wait the channel to become empty. When a message comes to the channel, the RTI P-process reads the message and then checks the preconditions of coming service call. If the preconditions are satisified, the postconditions of the service call are formed by the RTI P-process. `Message` data structure is used for communication on the channel. All the supplied arguments of the RTI services that we work on, are defined in the `Message` data structure. Figure 3.6 depicts the Message data structure.

```
typedef Message{
     int msgType;
     /*interaction class or object class*/
     mtype mClass;
    /*Attribute array for objects or parameters of
Interactions*/
     int mAttribute[OBJECTATTRIBUTENUM];
     /*Object attribute number */
     int mAttributeSize;
     /*object instance = object instance handle*/
```

```
    mtype mObjectInstance;
    /* Save Label */
    int mSaveLabel;
    byte mSaveFedereArray[FEDERENUM];
    mtype mDimensionDesignator[DIMENSIONNUM];
    int mDimensionCounter;
    mtype mRegionDesignator;
    mtype mRegionArray[REGIONNUM];
    int mRegionCounter }
```

Figure 3.6 : `Message` data structure used on channel communication

The federation designer fills the supplied arguments of RTI services during the modeling process. The PCG accesses the suplied arguments from the model. Figure 3.7 depicts preparing of `SubscribeObjectClassAttributes` message. `Object class designator` is defined as supplied argument of the `Subscribe Object Class Attributes` service in the interface specification document. The `class designator` in this example is Station. The second supplied argument is `set of attribute designators`. Attributes are not hold with their names because string type is not provided in PROMELA. Object ids of the attributes are taken from FAM. Object ids are unique given to FAM elements. So object ids are used instead of attribute names. It is not possible to reach the array size from the array in PROMELA, so attribute size is also sent in the message. Federate P-process prepares the message and writes it to the channel.

```
msg.msgType = SubscribeObjectClassAttributes;
msg.mClass = Station;
msg.mAttribute[0]=11;
msg.mAttributeSize =  1;
rtiChannel!msg,pPid;
```

Figure 3.7 : Preparing message of `Subscribe Object Class Attributes`

3.2 Combatting State Space Explosion

State space of a program is the set of states that can possibly occur during a computation [21]. State vector is the information to uniquely identify a system state; it contains:

- global variables,
- contents of the channels,
- local variables and
- process counter of the process for each process in the system [22].

As the state vector and state space increase, the amount of memory needed grows. As the number of federates in the FAM increase, the state space of the PROMELA model grows parallel with the increase in the number of the federate P-processes. Although the model of a system is finite-state, it typically grows exponentially [22] and we face with the state space explosion. The state space explosion is the main challenge in practical applications of model checking.

Some improvements are made on the PROMELA code to reduce the state vector size, so the state space size and to improve time and space utilization. These improvements are:

- Using unnecessary variable definitions are avoided to minimize the size of the state vector.
- d_step is used if possible.
- The usage of global variables are avoided.
- Seperate channels are used for communication between different processes.

The code shown in Figure 3.8 is grouped into a single d_step sequence, and the resulting state space of the PROMELA model is reduced by 13%. A d_step sequence is executed as if it were one single indivisible statement [20] disallowing interleaving of the statements in the sequence with other statements. The verification outputs in two cases are shown in Figure 3.9. When the d_step sequences are applied to the all code, the state space is nearly reduced to the half of its initial size as shown in Figure 3.10.

```
d_step {
 msg.mClass = Ship;
 msg.mObjectInstance = DiscoveredShipObject;
 msg.mAttribute[0] = 1;
 msg.mAttributeSize = 1;
 federationExecExistPreCondition();
 isJoinedPreCondition();
 joinedFederateDoesNotKnowAboutObjInsPreCondition();
 classAttribIsSubscribAtJoinedFederatePreCondition ();
}
```

Figure 3.8 : PROMELA code with one d_step sequence

(Spin Version 5.1.7 – 23 December 2008)	(Spin Version 5.1.7 – 23 December 2008)
+Partial Order Reduction	+Partial Order Reduction
State-vector 7536 byte, depth reached 254, errors: 0	State-vector 7536 byte, depth reached 250, errors: 0
255 states, stored	221 states, stored
0 states, matched	0 states, matched
225 transitions(= stored + matched)	221 transitions(= stored + matched)
0 atomic steps	0 atomic steps
Hash conflicts : 0 (resolved)	Hash conflicts : 0 (resolved)

Figure 3.9 : Reduction on state space with `d_step`

(Spin Version 5.1.7 – 23 December 2008) +Partial Order Reduction State-vector 2008 byte, depth reached 683, errors: 0 872 states, stored 135 states, matched 1007 transitions(= stored + matched) 0 atomic steps Hash conflicts : 0 (resolved)	(Spin Version 5.1.7 – 23 December 2008) +Partial Order Reduction State-vector 2008 byte, depth reached 277, errors: 0 455 states, stored 135 states, matched 590 transitions(= stored + matched) 0 atomic steps Hash conflicts : 0 (resolved)

Figure 3.10 : State space reduction of all PROMELA code

The usage of global variables are avoided as far as possible in the PROMELA code. Information of federates (i.e. published/subscribed object classes, registered object instances) is hold locally in RTI P-process. A federate information is only changed by the RTI P-process. So there isn't any necessity to hold the information globally. But the RTI channel which is reached from all federate P-processes and RTI P-process must be defined globally. Consequently, if it is inevitable to use global variables, they are used.

Separate channels are used for the messages that are sent from the RTI P-process to the P-processes of federates, because the interleaving of independent message streams into a single channel can be a huge source of complexity [20]. In our PROMELA model, each federate has its own callback channel for the callbacks from the RTI P-process to the federate P-process.

Although some improvement is made to reduce the state space, state space explosion can be still a problem for large federation architectures. As the number of participating federates to the federation is increased, the state space explosion seems inevitable.

SPIN has three search modes, namely, the exhaustive, bitstate and hash-compact modes. When the system is too large to verify with exhaustive mode, bitstate or hashcompact modes can be used. Bitstate hashing and hash compact are lossy because they may not store all the states that have been visited, so some parts of the state diagram may not be searched and some counterexamples may not found. That

is, a false positive is possible where no errors are reported although some may exist. Nevertheless, these methods are useful because they do not give false negatives, that is, if a counterexample is found, it represents a true error [21]. The federation designer can configure SPIN settings of search modes. XSPIN provides a menu that contains search modes. The user can easily select one of the search modes according to the situation.

4. PROMELA CODE GENERATOR

The PROMELA Code Generator (PCG) tool will be presented in this section. PCG takes FAM as input and generates the PROMELA code of the model as output (i.e. the federate P-process). The PCG consists of front end and back end parts. The front end traverses FAM, interprets the interface behavior of federates and takes the necessary arguments from the model. The back end generates the PROMELA code of the interface behavior of federates.

4.1 Example FAM: Strait Traffic Monitoring Simulation

Before explaining the PROMELA code generation process, we introduce Strait Traffic Monitoring Simulation (STMS). Later sections are accompanied by this example.

A traffic monitoring station tracks the ships passing through the strait. Any ship entering the strait announces her name and then periodically reports her position to the station and to the other ships in the strait using the radio channels. Channel-1 is used for ship-to-ship and channel-2 is used for ship-to-shore communication. The traffic monitoring station tracks ships and ships track each other through these communication channels. All radio messages are time-stamped to preserve the transmission order.

The traffic monitoring station and the ships are represented with two types of applications: a station application and a ship application, respectively. The ship application is an interactive federate allowing the player to pick up a unique ship name, a direction (eastward or westward), and a constant speed by means of a textual interface. Joining a federation corresponds to entering the strait, and resigning from the federation corresponds to leaving the strait. The station application is a monitoring federate, which merely displays the ships (in the strait) and their positions.

The conceptual view of the application is presented in Figure 4.1.

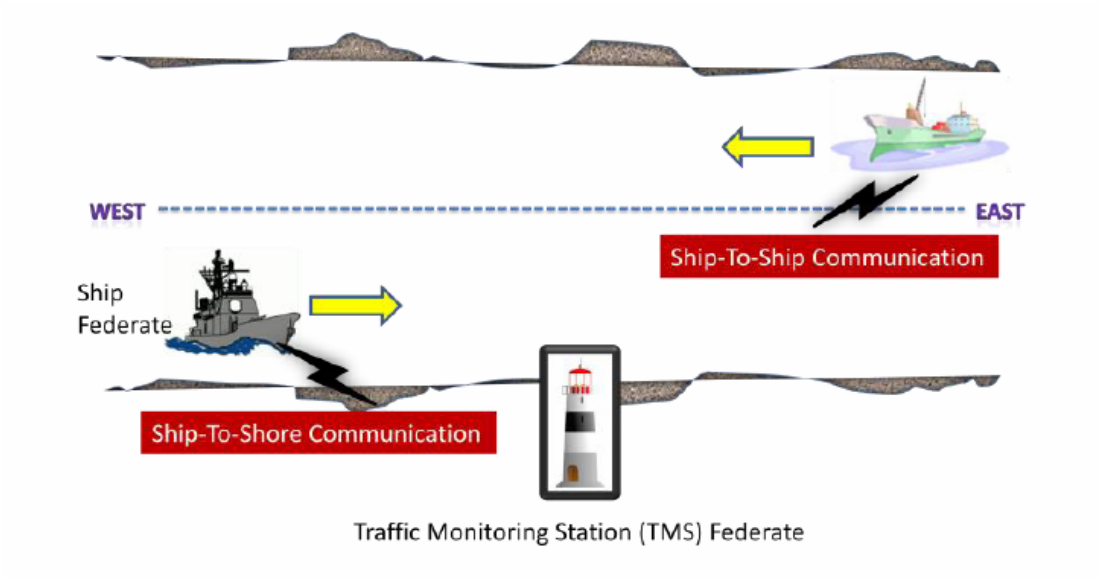

Figure 4.1 : Strait Traffic Monitoring Simulation conceptual view [2]

Figure 4.2 presents a screen shot of the project for the STMS federation architecture. The root folder (e.g., StraitTrafficMonitoringSimulation in the screen shot) serves as a project container for the federation architecture. It includes three major sub-folders, namely, federation structure, behavioral models, and HLA object models. The federation structure folder contains information about the federation, such as location of the FDD file, the link for the related FOM, and the structure of the federation, where the participating federate applications and their corresponding Simulation Object Models (SOMs) are linked. The folder for behavioral models includes an MSC document for each participating federate. HLA object models folder includes the FOM, SOMs, and the other Object Model Template related information (e.g., data types, dimensions, etc.). In the example, SOMs for ship and station applications and a FOM for the STMS federation are provided. IEEE 1516.1 Methods Library which contains the RTI services defined in the interface specification document, is attached to the project.

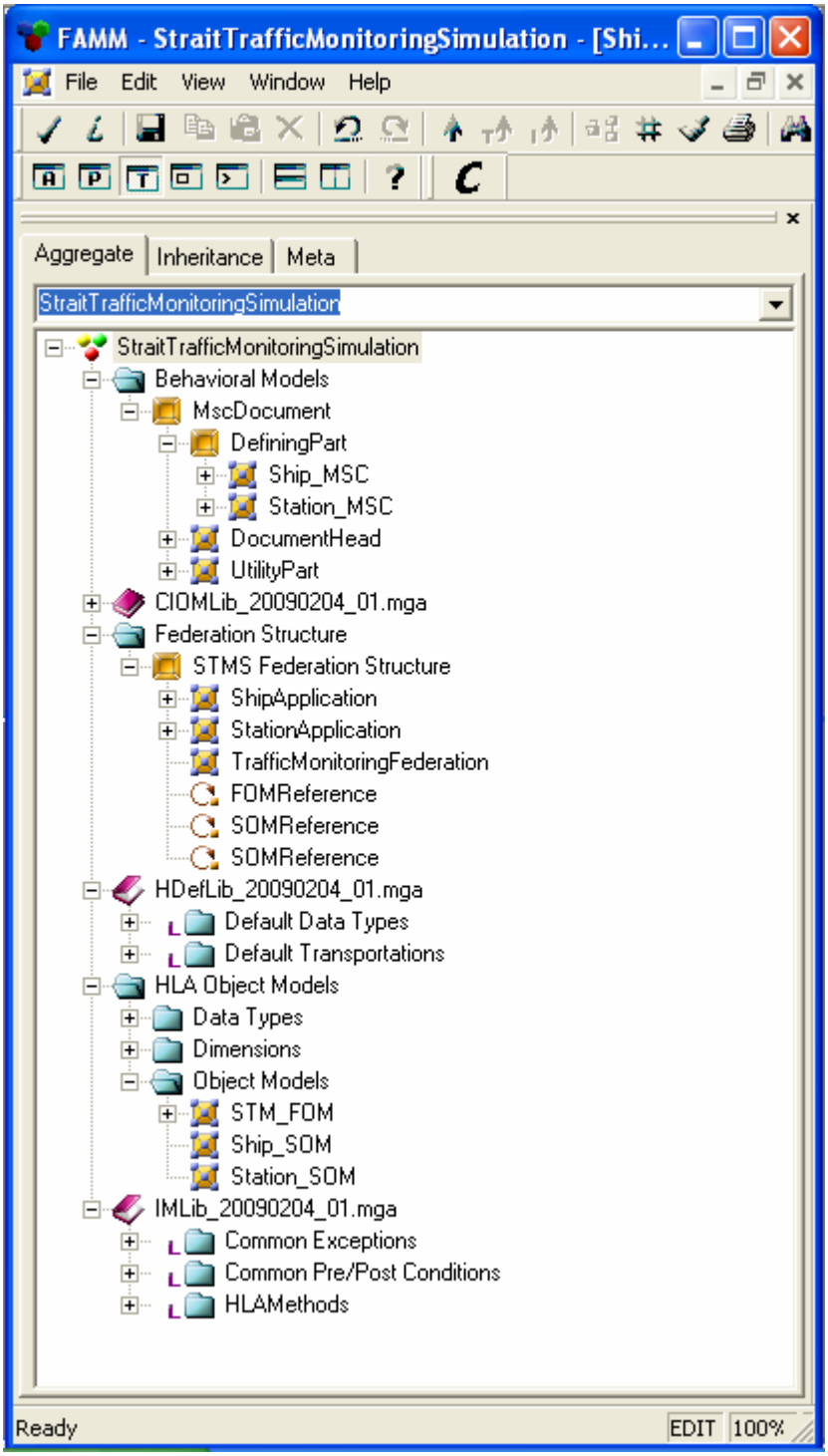

Figure 4.2 : Federation Architecture Modeling Environment (FAME)

4.2 Front End

The generator starts to traverse the Message Sequence Charts (MSCs) which are under the `DefiningPart` model of the MSC document in the `Behavioral Models` folder. It is assumed that each MSC under `DefiningPart` model belongs to one federate. MSCs and LSCs are nested structures. A LSC can contain another LSC. So recursive approach is followed for transforming FAM model elements to data structures. Figure 4.3 shows the prechart of the Station federate. `Prechart` is like a precondition, until it executes, the body part of the LSC will not execute. `Prechart` seen in Figure 4.3 consists of two `out` events and one reference to the MSC of station declera capability.

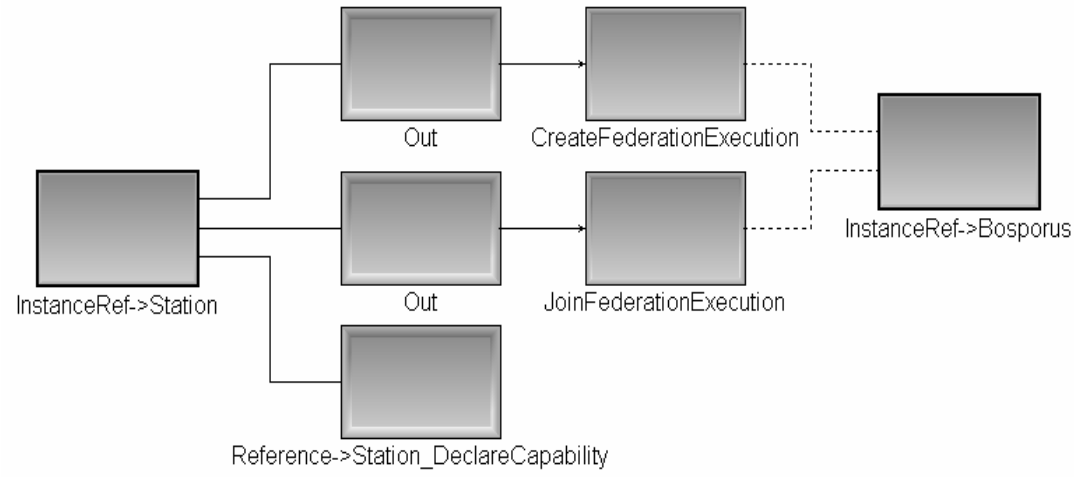

Figure 4.3 : `Prechart` of the Station federate

The outgoing message events are called `out` in FAMM. `Call` and `reply out` model elements are also used for outgoing message events. In the LSC of Station_DeclareCapability, published/subscribed object classes and interaction classes are modeled. Reference model element named Reference->StationDeclareCapability is a reference to Station_DeclareCapability MSC. This MSC includes a LSC and this LSC is represented in Figure 4.4.

In the front end of the PCG, `InlineExpression` and `InlineOperand` classes are defined to hold the model elements of FAMM. The model elements in the FAM are hold in a list data structure which has elements in type of `InlineExpression`. The `InlineExpression` object class has `InlineOperand` object as an attribute. If there are any inline expressions under an

inline expression, then these inline expressions are hold in `InlineOperand` object. `InlineOperand` class includes a list of `InlineExpression` objects. The relation between the object classes of traversing `prechart` can be seen in Figure 4.5 and activity diagram of traversing `prechart` is sketched in Figure 4.6.

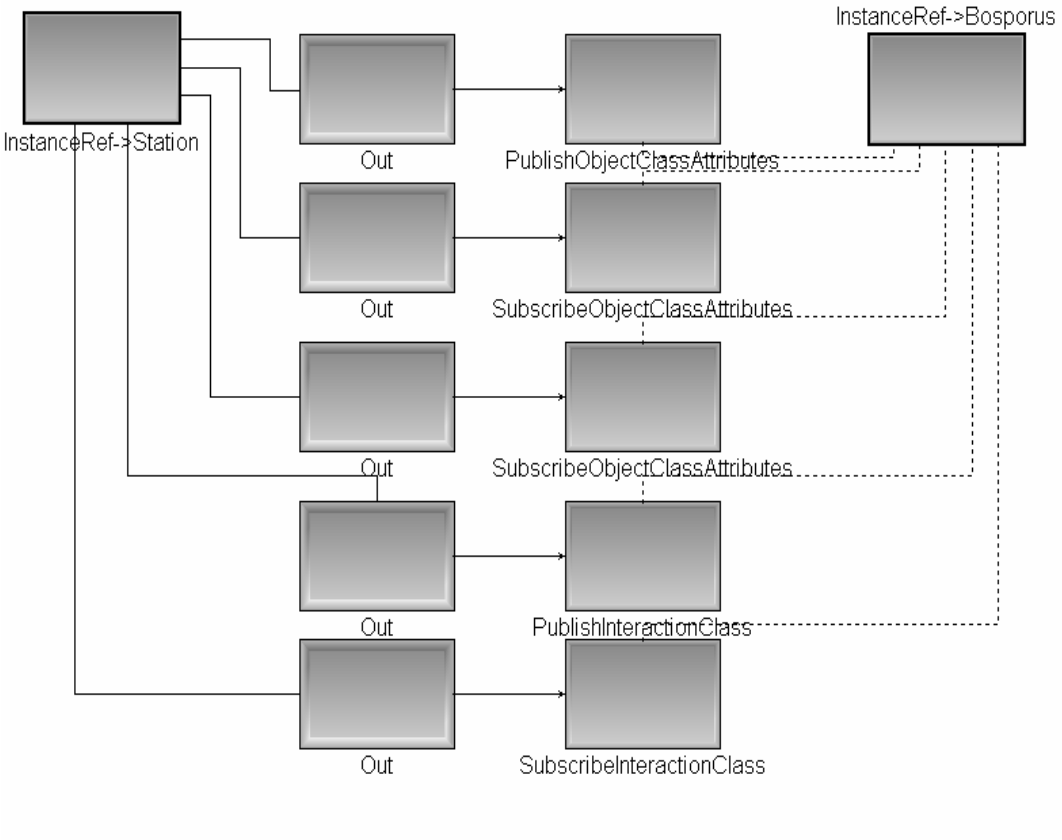

Figure 4.4 : LSC of the Station federate declare capability

LSCs under the MSCs of federates are firstly added to inline expression list while traversing the FAM. When an `InlineExpression` object is created, an `InlineOperand` object is created. The `InlineOperand` object is an attribute of the `InlineExpression` object. Model traversing goes on until finding `in` and `out` events. `In` and `out` events are added as `InlineExpression` object in the front end. `InlineOperand` object of the `InlineExpression` is instantiated. If the destination of the `in` and `out` event is an HLA method then `HLAMethodStruct` object is instantiated and name, supplied/returned arguments, the initiator attribute (e.g. federate-initiated or RTI-initiated) of the HLA method is taken from the `HLAMethod` model element.

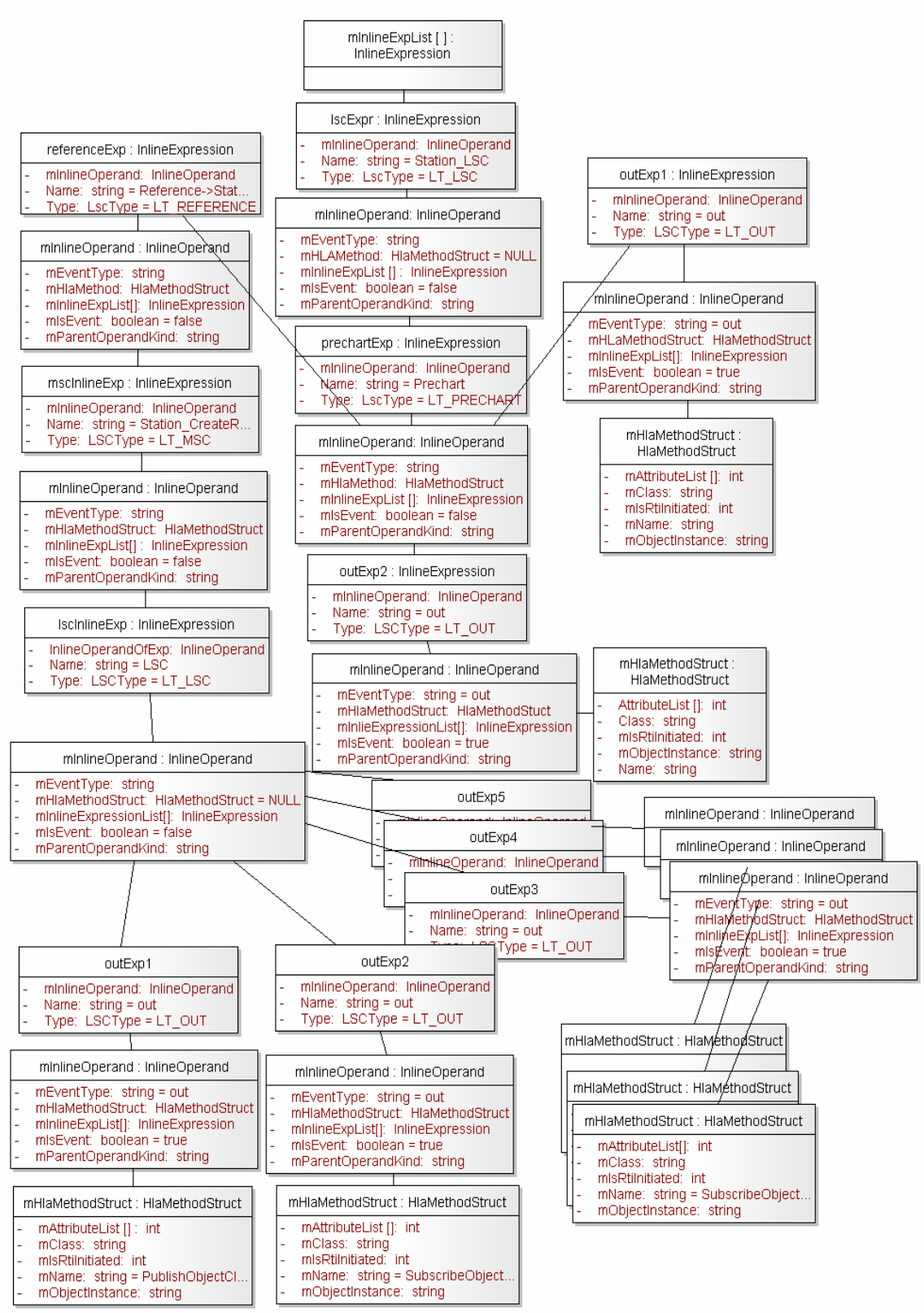

Figure 4.5 : Class diagram of prechart

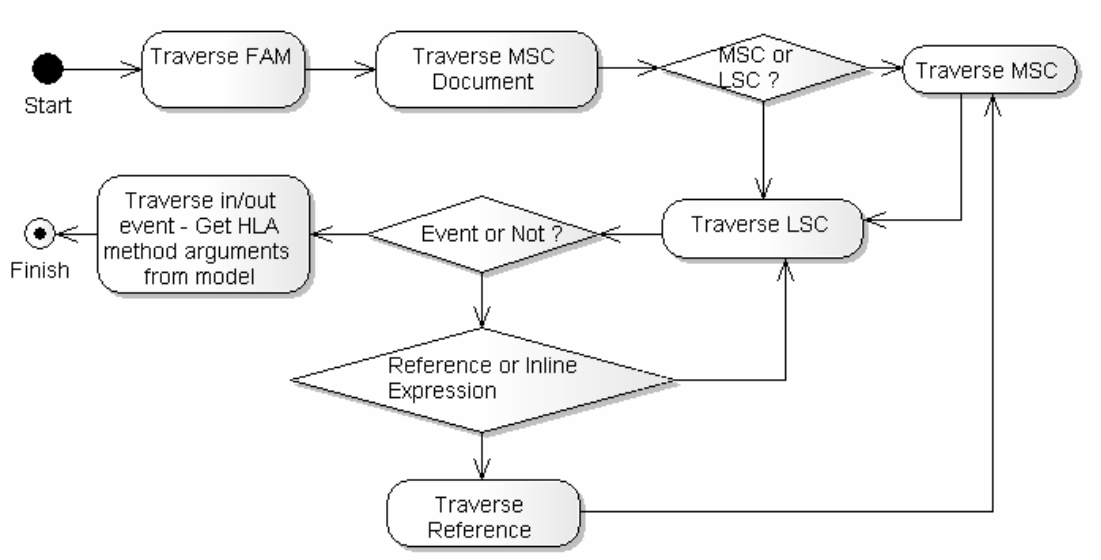

Figure 4.6 Activity diagram of prechart traversing

4.3 Back End

After traversing FAM, the MSC constituents (the software data structures of MSCs) has been created. Thus PROLEMA code generation is done from the front end output. Firstly, it is necessary to define object classes, interaction classes defined in FOM and registered object instances of federates. PROMELA does not support string variables. Object/interaction class definitions and object instance definitions are made with `mtype` in PROMELA. An `mtype` declaration allows for the introduction of symbolic named for constant values. There can be multiple `mtype` declarations in a verification model [20]. The root folder (e.g., `StraitTrafficMonitoringSimulation` in the screen shot) serves as a project container for the federation architecture [2]. FOM is reached under the `HLA object models` folder and it contains the interaction and object classes. STMS includes the object classes:

- `Ship`
- `Station`

The interaction class defined for STMS is:

- `RadioMessage`

Generated PROMELA code for object and interaction classes is:

```
mtype = {Station,Ship}

mtype = {RadioMessage,DefaultInterValue}
```

Interaction class parameters and object class attributes must be produced in PROMELA. To meet these needs `fillObjectInteractionInfo()` inline construct is produced. This inline construct is called from RTI P-process. `fillObjectInteractionInfo()` produced for STMS is shown in Figure 4.7. PROMELA does not support string variables. Thus parameters and attributes are hold in PROMELA code with their object identities (ids).

```
inline fillObjectInteractionInfo()
{
objectClassFDDArray[0].mObjectClass = Station;
objectClassFDDArray[0].mAttribute[0].mDesignator =11;
objectClassFDDArray[0].mAttribute[1].mDesignator =12;
objectClassFDDArray[0].mAttributeSize = 2;

objectClassFDDArray[1].mObjectClass = Ship;
objectClassFDDArray[1].mAttribute[0].mDesignator =7;
objectClassFDDArray[1].mAttribute[1].mDesignator =9;
objectClassFDDArray[1].mAttribute[2].mDesignator =6;
objectClassFDDArray[1].mAttribute[3].mDesignator =8;
objectClassFDDArray[1].mAttributeSize =4;

interactionFDDArray[0].mClass = RadioMessage;
interactionFDDArray[0].mAttribute[0] =16;
interactionFDDArray[0].mAttribute[1] =17;
interactionFDDArray[0].mAttributeSize =2;
}
```

Figure 4.7 : Generated PROMELA code of `fillObjectInteractionInfo()`

Object instances are taken from the federate instances in the FAM. A federate instance includes a `VariableList`. Object instances are modeled under the variable list. Models kind of ObjectClass is taken from the `VariableLists`. The generated PROMELA code for the STMS is:

```
mtype = {DiscoveredStationObject, DiscoveredShipObject,
RegisteredShipObject, RegisteredStationObject}
```

An inline expression list is produced as front end output. During the PROMELA code generation from the LSCs, a different approach is followed for each LSC inline expression. Federates of a federation are transformed to processes (P-processes). Federate P-processes run in parallel. For `while-do` inline expression, `do-statement` in PROMELA is used. PROMELA provides `do-statement` idioms. Main thread of the Station federate can be seen in Figure 4.8 and its PROMELA code is in Figure 4.9. Main simulation loop (`while-do`) of the Station federate terminates when the counter (`i` variable in Figure 4.9) of the loop exceeds the conditional value (This value is two in Figure 4.9).

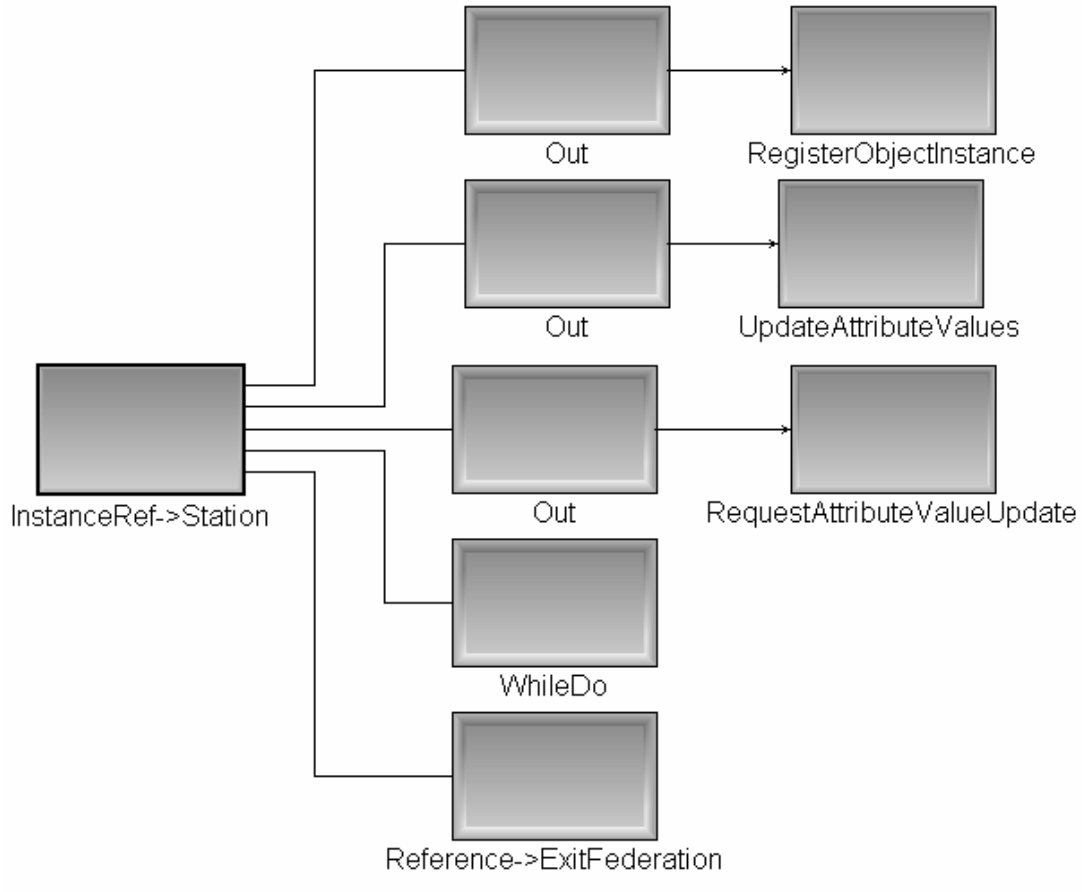

Figure 4.8 : Main thread of the Station federate

```
proctype  MainThread0(int pPid)
{
   Message msg;
   msg.msgType = RegisterObjectInstance;
   msg.mClass = Station;
   msg.mObjectInstance = RegisteredStationObject;
   rtiChannel!msg,pPid;

   msg.msgType = UpdateAttributeValues;
   msg.mClass = Station;
   msg.mObjectInstance = RegisteredStationObject;
   msg.mAttribute[0]=105;
   msg.mAttributeSize =  1;
   rtiChannel!msg,pPid;
   msg.msgType = RequestAttributeValueUpdate;
   rtiChannel!msg,pPid;
   int i=0;
 do ::
  if
       ::(i != 2) ->
        msg.msgType = SendInteraction;
        msg.mClass = RadioMessage;
        msg.mAttributeSize =  0;
        rtiChannel!msg,pPid;
        msg.msgType = SendInteractionWithRegions;
```

```
        rtiChannel!msg,pPid;
        i = i + 1;
      ::else ->break;
   fi;
 od;skip
}
```

Figure 4.9 : PROMELA code of the main thread

The MSC metamodel contains many constituents. We handle `out` and `in` from the message events, `while-do` and `par` inline operators from inline expressions and `coregions`. `Out` and `in` events provide to access HLA methods in FAM.

`Coregion` transformation to PROMELA is implemented. A `coregion`, specified with the start and the end events (`concurrent` and `endconcurrent` respectively), is a part of the instance axis for which the events connected to that part are assumed unordered [2]. `Coregions` are usually used in callback thread of FAM to model the RTI-initiated messages whose order is unspecified. A `coregion` can be transformed to switch-clause. PROMELA does not support `switch-clause` so `coregions` are transformed as `if-clauses` to PROMELA.

`Parallel inline expression (par)` is defined for parallel execution. It can include one or more operands. Each operand is transformed as a seperate process to PROMELA. Generated processes are not active processes. Keyword `active` defines a set of processes that are required to be active (i.e., running) in the initial system state [20]. The processes, which are not active, are runned from the main P-process of the federate. The federate P-process consists of the main thread and callback thread that run in parallel. The callback thread handles RTI-initiated messages. The main thread is the main simulation loop and it sends the federate-initiated messages to the RTI. These threads are modeled with `par` inline expression. Callback thread model is transformed as a process to PROMELA. The LSC of callback thread can be seen in Figure 4.10. The behavior seen in Figure 4.10 is a `while-do` loop. Inside the loop a `coregion` is used for the unordered RTI-initiated services. They are modeled with `coregion`, because RTI-initiated service

calls may come at any time from the RTI. The `coregion` is transformed to PROMELA code as if-clause. Figure 4.11 depicts the PROMELA model of the callback thread.

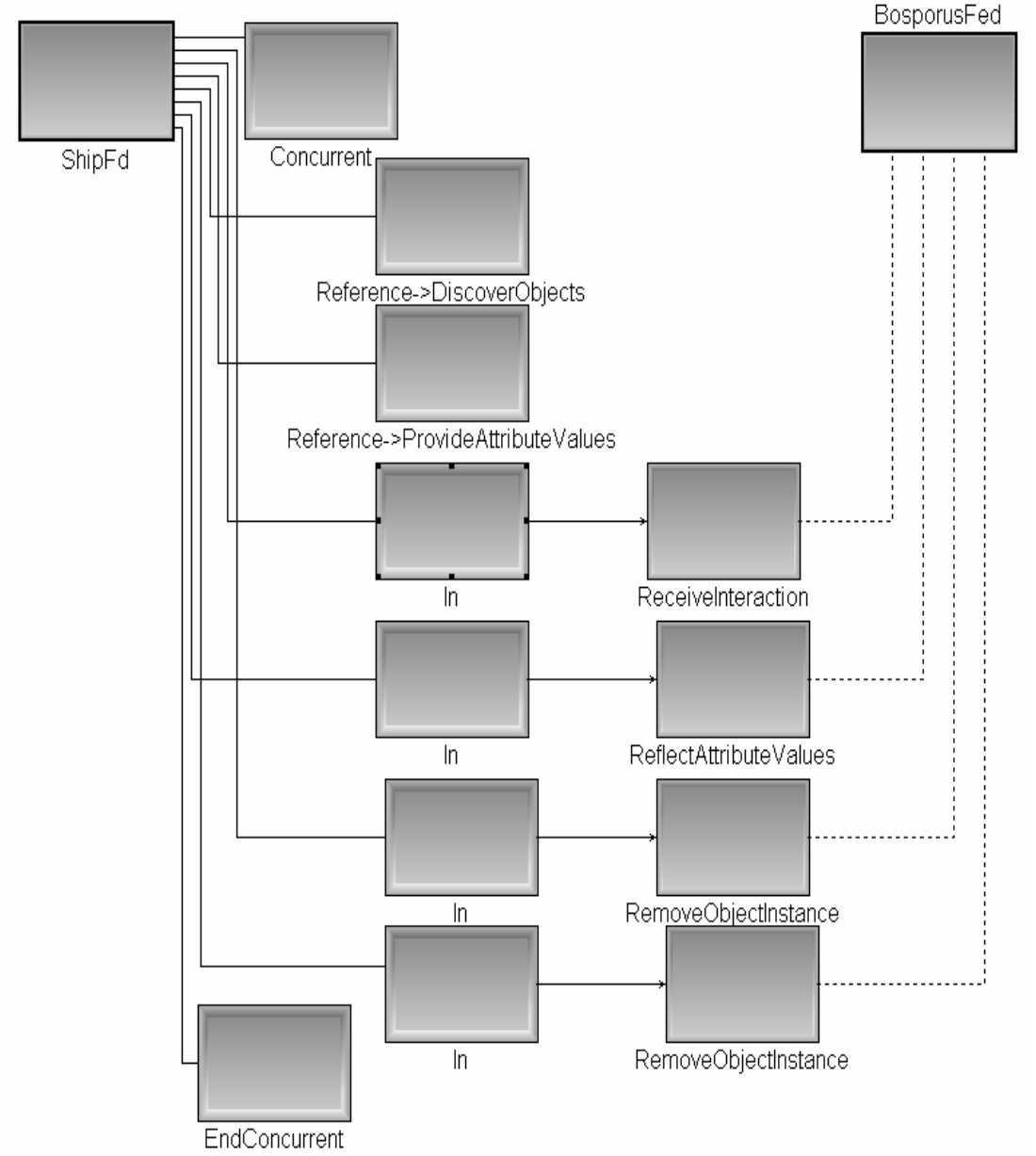

Figure 4.10 : Callback thread model of the Ship federate

```
proctype  CallbackThread1(int pPid)
{
  Message msg;
  end:
  do ::
    callbackChannelArray[pPid-1] ? msg;
    if
    ::(msg.msgType == ProvideAttributeValueUpdate) ->
       msg.msgType = UpdateAttributeValues;
       msg.mClass = Ship;
       msg.mObjectInstance = RegisteredShipObject;
       msg.mAttribute[0] = 6;
       msg.mAttributeSize =  1;
       rtiChannel!msg,pPid;
     ::(msg.msgType == ReceiveInteraction) ->
     ::(msg.msgType == ReflectAttributeValues) ->
     ::(msg.msgType == RemoveObjectInstance) ->
     ::(msg.msgType == RemoveObjectInstance) ->
      ::else -> break;
      fi;
 od;skip
}
```

Figure 4.11 : PROMELA code of the callback thread

4.4 Running the PCG

First run the GME. Click `File` and then select `Run Interpreter`. This selection displays the interpreters registered to the GME. Select `PCG Interpreter` as seen in Figure 4.12.

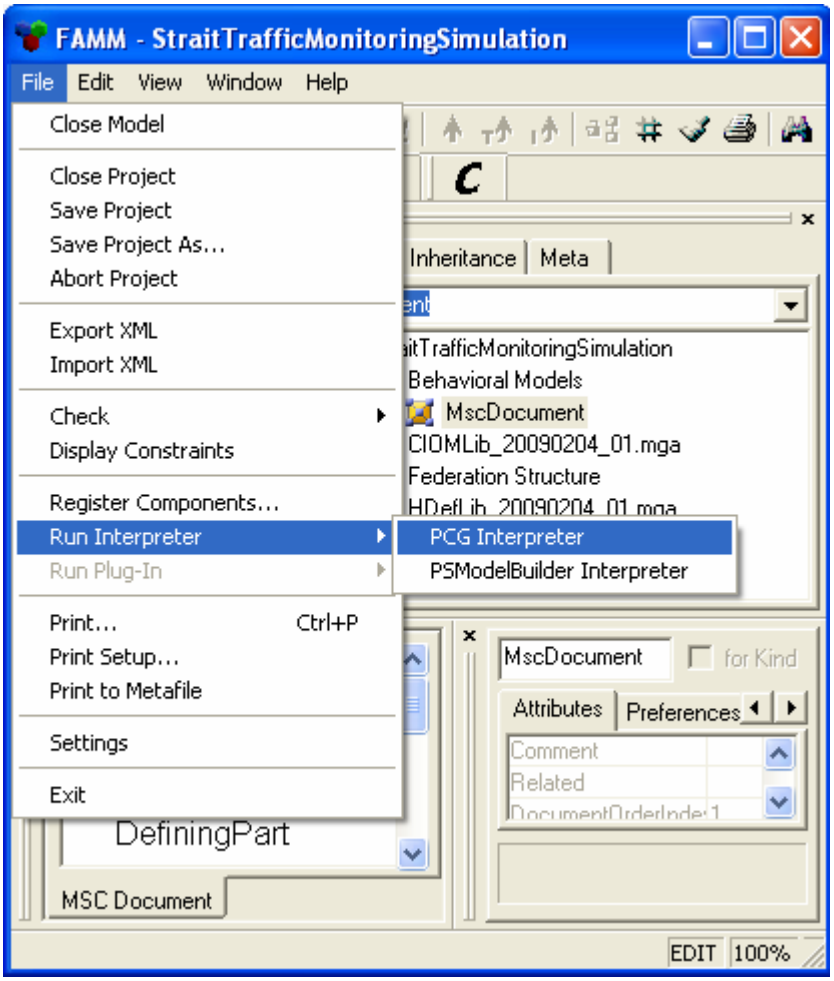

Figure 4.12 : Running the PCG

After selecting `PCG Interpreter`, `PROMELA Code Generator` dialog will appear as seen in Figure 4.13. The federates modeled in FAM is listed in the list box. The user can select the federates which she/he wants to verify. Selecting more than one federate can make the verification more efficient. Because one federate's federate-initiated service call may trigger the RTI to make an RTI-initiated service call to other federates. So the RTI-initiated service call will be verified in terms of

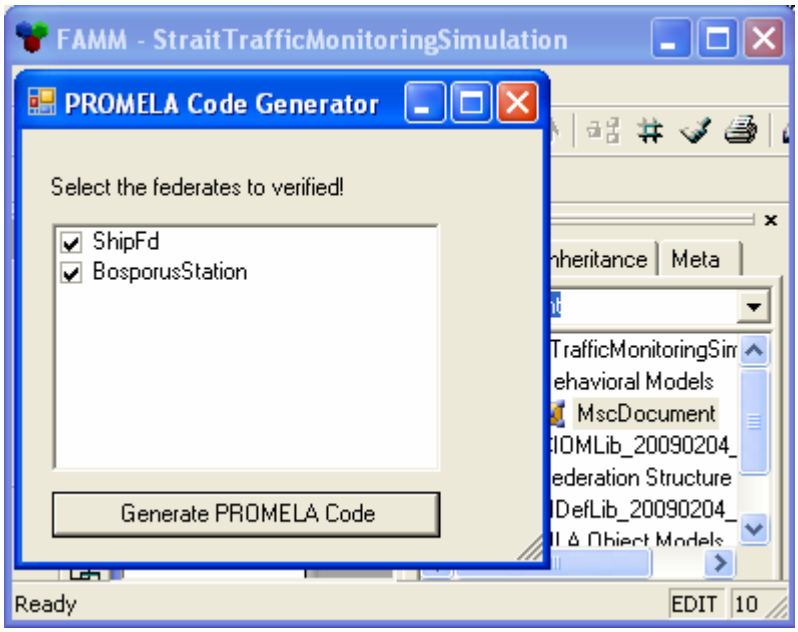

Figure 4.13 : Generating PROMELA code

other federates. If only one federate is selected, then only the federate-initiated messages will be verified. Finally, click `Generate PROMELA Code` button. This operation will produce the PROMELA code named `output.pml` which will be the input to the SPIN and XSPIN starts automatically with loaded file `output.pml` as seen in Figure 4.14.

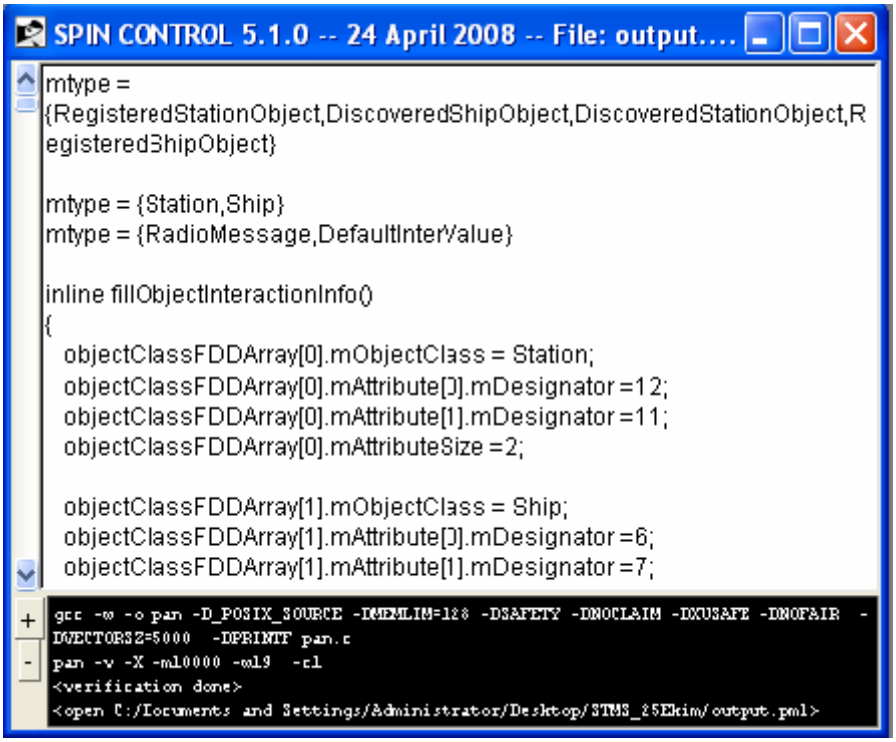

Figure 4.14 : XSPIN view

The mappings of STMS model elements to PROMELA can be seen in Table 4.1.

Table 4.1: PROMELA code mappings of STMS model elements

MSC Constituent	**PROMELA Code**

Par (with two inline operands)- MainThread and Callback Thread	`proctype MainThread0(int pPid)`
	`{`
	`Message msg;`
	`msg.msgType = RegisterObjectInstance;`
	`msg.mClass = Ship;`
	`msg.mObjectInstance=RegisteredShipObject;`
	`entry!msg,pPid;`
	`……`
	`}`
	`proctype CallbackThread1(int pPid)`
	`{`
	`Message msg;`
	`end:`
	`do`
	`::callbackChannelArray[pPid -1] ? msg;`
	`if`
	`::(msg.msgType == DiscoverObjectInstance)`
	`->`
	`……`
	`}`

While-do	`do`
	`::if`
	`::(i != 1) ->`
	`msg.msgType = SendInteraction;`
	`i = i + 1;`
	`……`
	`::else -> break`
	`fi;`
	`od;skip`

Coregion	`if`
	`::(msg.msgType == DiscoverObjectInstance)`
	`->`
	`::(msg.msgType == ReceiveInteraction) ->`
	`::(msg.msgType == ReflectAttributeValues)`
	`->`
	`::(msg.msgType == RemoveObjectInstance)->`
	`::else`
	`fi;`

4.5 SPIN Verification

XSPIN graphical interface is used for running SPIN. XSPIN provides a menu for user to select verification options shown in Figure 4.15. The user must click on the `Run` button from the menu and then select the `Set Verification Parameters` to reach the `Basic Verification Options` window.

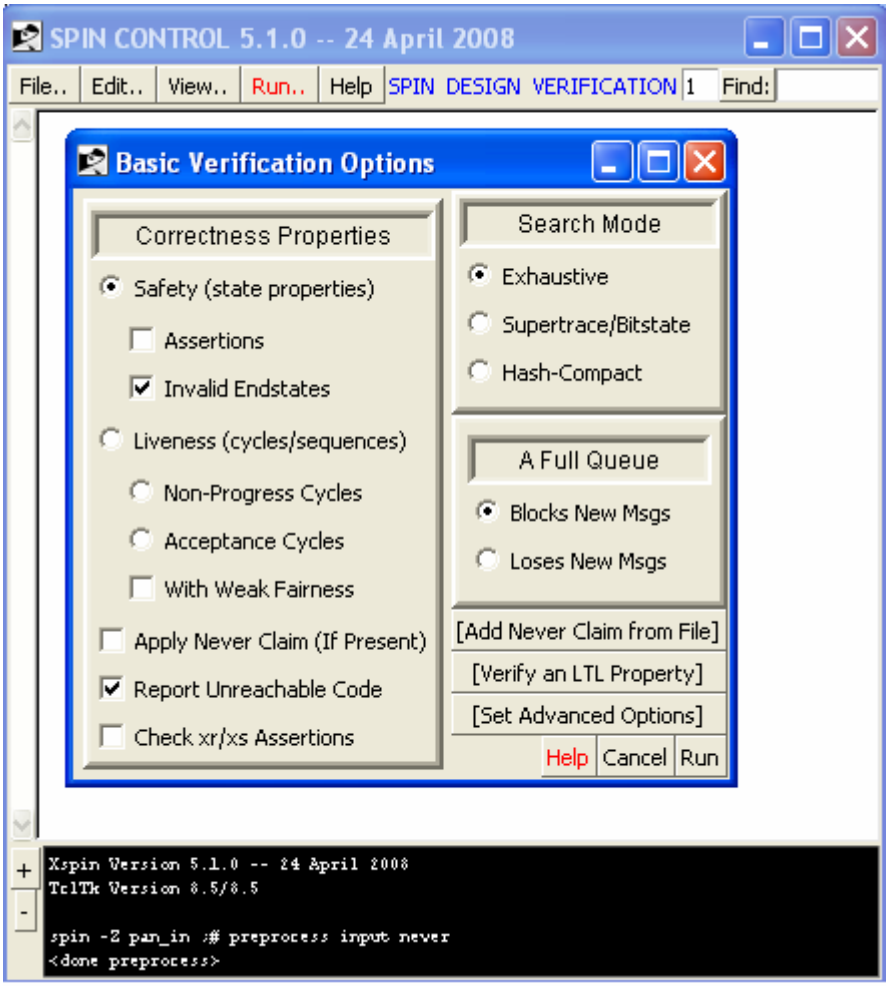

Figure 4.15 : Basic verification options

In the thesis, we used default correctness properties. Exhaustive mode is used as search mode which stores all visited states in the state space. When the state space of the PROMELA is large, then some other commands must be entered for running SPIN. `Advanced Verification Options` in Figure 4.16 presents more rarely used settings. These settings facilitate to verify models with large state space. While compiling the PROMELA model of STMS, vector size is 4116 byte. Default vector size of SPIN is 1024 byte, so the vector size is set to 4500 byte in the `Extra Compile Time Directives` field from the `Advanced Verification`

Options window. PROMELA printf statements are normally disabled. We need to see the checking results of the preconditions. So the verifier is compiled with the extra compile time directives: –DVECTORSZ = 4500 –DPRINTF.

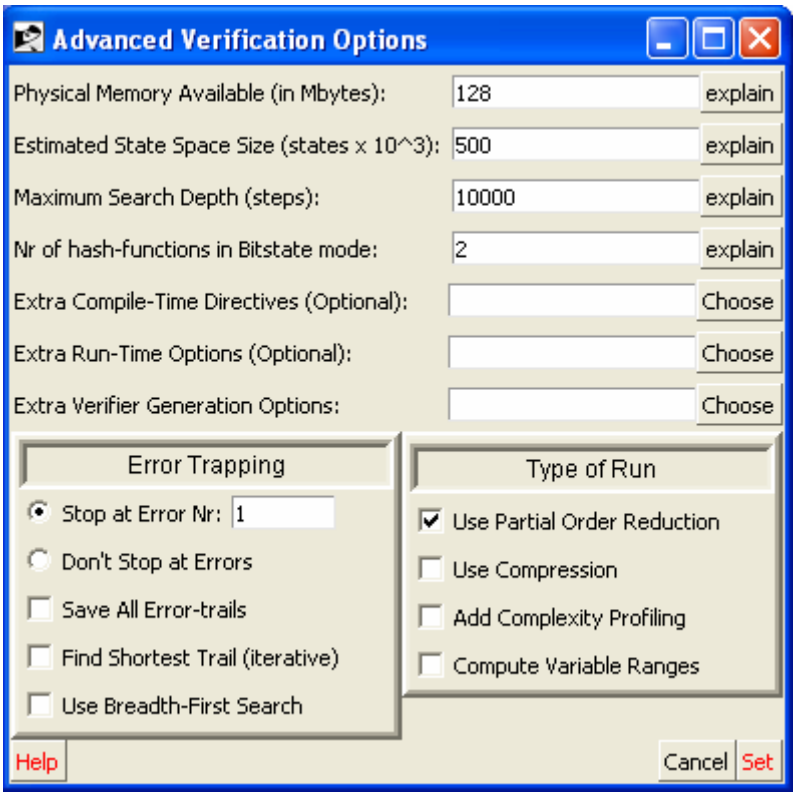

Figure 4.16 Advanced verification options

Verification process with SPIN is made step by step. As said, the PCG generates output.pml file, which contains the federate P-processes. SPIN executes and produces the pan.*c* file. Pan.c file is the verifier. The verifier pan.c is compiled and then output pan.exe is executed and the verification is performed. XSPIN abstracts these operations from the user.

Verification results of STMS federation architecture model is seen in Figure 4.17. Partial order reduction algorithm is used. This algorithm reduces the size of the state space to be searched by the verifier. State vector size is 4116 byte and it exceeds the default vector size (1024 Byte). Unreached statements are not error. They only give information about the statements that are not executed in the PROMELA code (dead code). All the services handled by the RTI P-process are not always called by the federate P-processes. So the code of services, which are not called by the federate P-processes, are dead code.

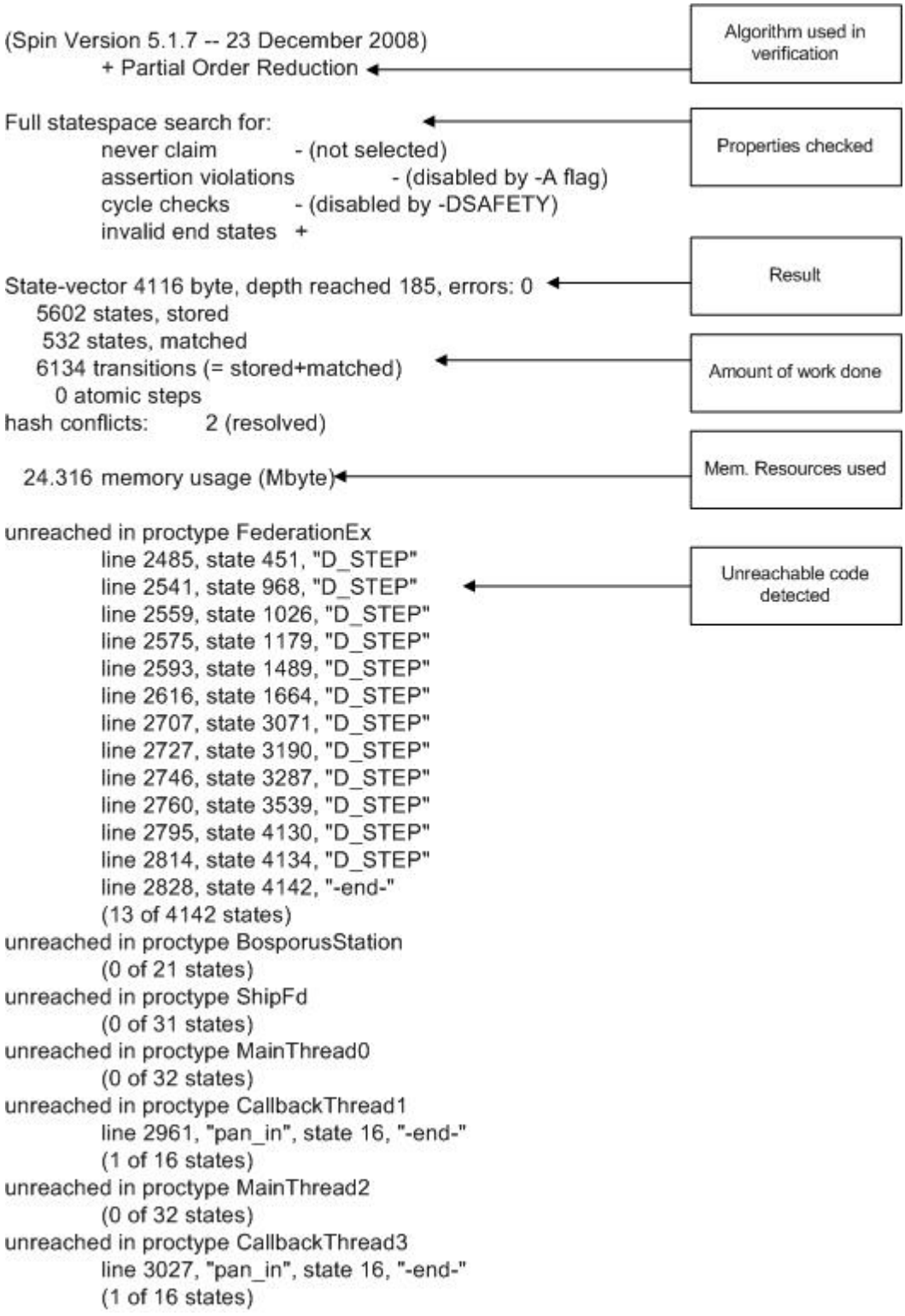

Figure 4.17 : Verification output of SPIN

In the verification of STMS PROMELA model printf statements are enabled. So verification output is not all seen in Figure 4.17. Results of checking the preconditions of each RTI service call coming to RTI P-process is displayed by the

help of printf statements. The federation designer detects the errors on interface behavior of federates by evaluating the verification output. Assertions can be used in PROMELA code. In this thesis, assertions are used because assertions makes easy to trace the error. printf statements are also helpfull to show what the error is. Figure 4.18 depicts some results of the Ship federate interface behavior verification. `Join Federation Execution` service is called twice in the model of Ship federate. One of the preconditions of `Join Federation Execuion` service is: The federate is not joined to that federation execution. So there will be an error at the second call like "Federate with pid = 2 has already joined to the federation" is displayed at the verification output. Postconditions of the service will not actualize. The federation designer interprets the output and makes the necessary changes on the FAM. Hereby, exceptions that will be arised from this error in the federate code, which is generated automatically from the FAM, is prevented.

```
Ship Federate Process with id = 2
BosporusStation Federate Process with id = 1
RTI Process with pid = 0
Message  = CreateFederationExecution from federate with  pid = 2
RTI Service: Create Federation Execution
The federation execution does not exist.
Message = JoinFederationExecution from federate with  pid = 2
RTI Service: Join Federation Execution
Federation has been created
Save not in progress
Restore not in progress
Join Precondition Control:
Federate with pid = 2 is not joined to the federation
Fedele is joined to federation
Message = JoinFederationExecution from federate with  pid = 2
```

RTI Service: Join Federation Execution

Federation has been created

Save not in progress

Restore not in progress

Join Precondition Control:

Federate with pid = 2 has already joined to the federation

pan: assertion violated 0 (at depth 21)

pan: wrote pan_in.trail

Figure 4.18 : Join Federation Execution service call two times

Another modeling error in our example FAM can be seen in Figure 4.19. Ship federate calls the `Update Attribute Values` service before calling the `Register Object Instance`.

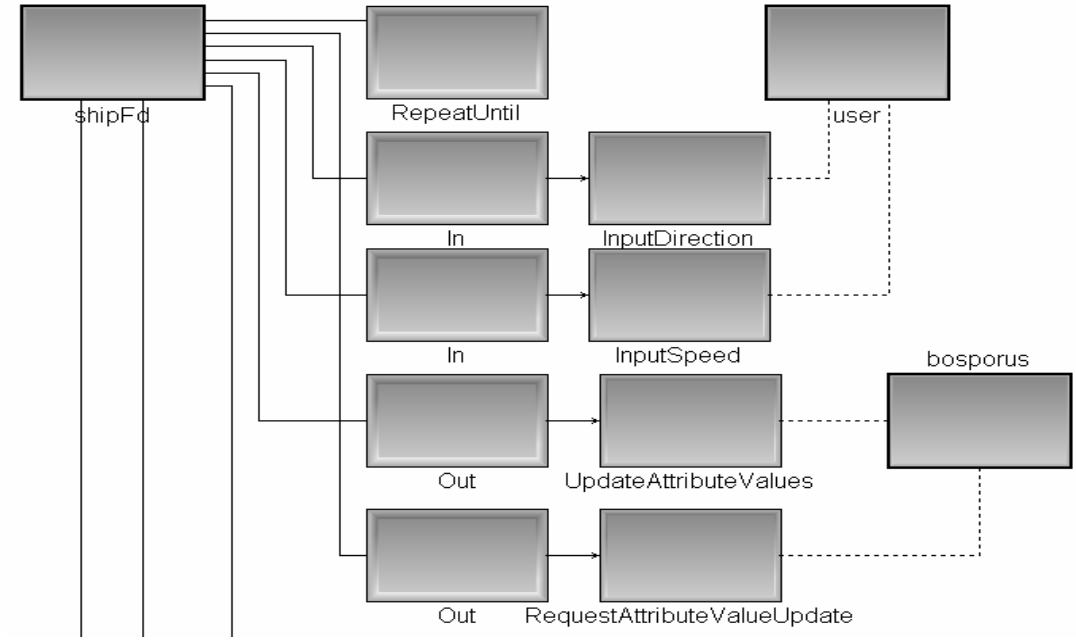

Figure 4.19 : Main thread of the Ship federate (partial view)

One of the preconditions of the `Update Attribute Values` service is: The joined federate knows about the object instance with the specified designator. Ship federate does not know about the object instance because it does not register or

discover the object instance. So the precondition of the `Update Attribute Values` service can not be provided. The verification result of the `Update Attribute Values` is seen in Figure 4.20.

Message = UpdateAttributeValues from federate with pid = 2

RTI Service: Update Attribute Values

Federation has been created

The federate with pid 2 is joined to that federation execution

Instance attributes are not owned by the joined federate

Object Class is specified in FDD

Attribute = 6 is defined in FDD

Attributes are available

An object instance with the specified designator does not exist.

The joined federate does not know about the object instance with the specified designator

Save not in progress

Restore not in progress

pan: assertion violated 0 (at depth 54)

pan: wrote pan_in.trail

Figure 4.20 : `Update Attribute Values` call without registering object instance

5. CONCLUSION AND FUTURE WORK

This study implements a model-checking approach to check the interface behavior of an HLA federate modeled in FAMM. PROMELA models of interface behaviors of federates are generated by analyzing LSCs. PROMELA model of RTI is also implemented once besides automatically generated PROMELA models of interface behaviors of federates. PROMELA models are given as input to the SPIN model checker. SPIN runs the PROMELA models. RTI PROMELA model checks the preconditions of all HLA services (federate-initiated or RTI-initiated). If the preconditions of an HLA service are not provided, the verification stops and the result is returned to the modeler. If all the preconditions of an HLA service are provided, postconditions of the HLA service are formed. If it is shown that all the preconditions of the HLA RTI services used in the behavioral model are satisfiable then the modeler can have some assurance that the interface behavior can be compliant to the HLA Federate Interface Specification.

Verification is performed automatically by the help of (1) a model interpreter that takes the FAM as input and generates the PROMELA code as output, (2) the SPIN model checker that performs model checking using the generated PROMELA code as input and then outputs the verification results. This early verification of interface behaviors of federates makes the following contributions to the federation development process:

- Verifying that all preconditions of the RTI services used in a behavioral model are satisfiable allows the federation designer to have some confidence that the interface behavior (in terms of services in the behavioral model) is modeled according to the HLA Federate Interface Specification.
- As a result of this study, the well-behavedness of a FAM can be checked facilitating to generate the federation code successfully for a prototype federation. The federation designer can detect the mistakenly modeled interface behavior of federates in the FAM by using the preconditions of the RTI services for verification.
- Federate developers save time by finding faulty behavior model during modeling phase. Because when the generated code runs and produces run time errors, it is more difficult to find errors.

- Through the verification process the federate developers examine the interface behaviors of federates in a more detailed form. This provides to see which RTI services must be called in the beginning of the federate execution.
- During verifications SPIN checks for the absence of deadlocks and unexecutable code. If there is any meaningless interface behavior model exists, this is catched as a result of verification process.

5.1 Future Work

Time Management, which brings up the time stamp ordered (TSO) events, is left as a future work, because implementation of TSO events differs from receive-ordered (RO) event management. TSO events are queued according to their timestamp values. TSO events can be accessed by the federate if their timestamps are less than or equal to the federate's current time, so TSO events are not immediately available to the federate as receive ordered events and they use different event queue than RO events. As a future work, a PROMELA model for time management can be developed.

State space is an important point in model checking. We encounter with the state space explosion if the resources are not enough. There can be made some improvements to solve this problem, such as creating some sub-models to be verified distinctly and the verification results can be combined afterwards, so that much bigger federations can be verified with limited resources. The model checking can be reduced to only one federate's interface behavior and in this context assume-guarantee reasoning can be examined.

Modeling the behavior of a federate can involve not only the HLA-specific behavior, but also the interactions between the components of the federate and the actors (e.g., interactive users and live entities) in the environment. So verification of behavior models, which are not HLA-specific, can be studied.

It can also be aimed to minimize the behavioral modeling effort without loosing any details in federate's behavior in order to simplify the federation designer's work. The federation designer can specify the minimal and basic behaviors in model (behavioral model in trimmed form – user friendly), and then the model transformer can provide the completion of the missing methods to generate an RTI-friendly full model by evaluating the verification results of pre- and postconditions.

Current PCG presents the verification results with the help of printf statements. This output can be expanded. The problem in the model can be sketched as sequence diagram and custom messages about the problem can be given to the federation designer. This approach will facilitate the tracebility of source FAM and PROMELA model.

REFERENCES

[1] **Federate Interface Specification,** 2000: IEEE 1516.1 Standart for Modeling and Simulation (M&S) High Level Architecture (HLA), 21 September.

[2] **Topçu O., Adak M., Oğuztüzün H.,** 2008: A Metamodel for Federation Architectures, ACM Transactions on Modeling and Computer Simulation (TOMACS), vol. 18, issue 3, article no. 10, pp.10:1-10:29, DOI:10.1145/1371574.1371576 <*http://doi.acm.org/10.1145/1371574.1371576*>, July.

[3] **Damm W., Harel D.,** 2001: LSCs: Breathing Life Into Message Sequence Charts, in Formal Methods in System Design, 19, 45-80.

[4] **ITU-T Recommendation Z.120,** 2004: Formal Description Techniques (FDT) – Message Sequence Charts. Pre-published Recommendation Telecommunication Standardization Sector of International Telecommunication Union (ITU-T).

[5] **Adak M., Topçu O., Oğuttüzün H.,** 2009: Model-based Code Generation for HLA Federates, will appear in Wiley Interscience Software: Practice and Experience (SPE) Journal, 2009.

[6] **Framework and Rules,** 2000: IEEE 1516 Standart for Modeling and Simulation (M&S) High Level Architecture (HLA), September.

[7] **Federation Development and Execution Process (FEDEP),** 2003: IEEE 1516.3 Standard for IEEE Recommended Practice for High Level Architecture (HLA), Apr 23.

[8] **Schafer T., Knapp A., Merz S.,** 2001: In Scott D Stoller and Willem Visser, editors, Proc. Wsh. Software Model Checking, volume 55(3) of Electr. Notes Theo. Comp. Sci., 13 pages.

[9] **Lilius J., Paltor P. I.,** 1999: vUML: A Tool for Verifying UML Models. ASE, pp.255, 14th IEEE International Conference on Automated Software Engineering (ASE'99).

[10] **Jussila T., Dubrovin J., Junttila T., Latvala T., Porres I.,** 2006 : Model Checking Dynamic and Hierarchical UML State Machines. In Proc. 3rd International Workshop on Model Development, Validation and Verification (MoDeV2a) pages, 94-110, Genova, Italy, 12-15 October.

[11] **Tufarolo J., Nielsen J., Symington S., Weatherly R., Wilson A., Ivers J., and Hyon C. T.,** 1999: Automate Distributed System Testing: Designing an RTI Verification System. Proceedings of the 1999 Winter Simulation Conference.

[12] **Tufarolo J., Nielsen J., Symington S., Weatherly R., Wilson A., Ivers J., and Hyon C. T.,** 1999: Automate Distributed System Testing: Application of an RTI Verification System. Proceedings of the 1999 Winter Simulation Conference.

[13] **Loper M., McLean T., Horst M., Crawford K.,** 1997: The High Level Architecture Federate Conformance Testing Process (Revised). 1997

Fall Simulation Interoperability Workshop (SIW), September 8-12, 1997.

[14] **Kuhl F., Weatherly R., and Dahmann J.,** 1999: Creating Computer Simulation Systems. Prentice Hall PTR.

[15] **Object Model Template Specification,** 2000: IEEE 1516.2 Standard for Modeling and Simulation (M&S) High Level Architecture (HLA), 21 September.

[16] **GME 7 User's Manual:** Version 7.0. Institute for Software Integrated Systems, Vanderbilt University.

[17] **Holzmann J. G.,** 1997: The Model Checker SPIN. IEEE Transactions on Software Engineering, Vol. 23, and No. 5, May.

[18] **Url-1** <*http://spinroot.com/spin/Man/Manual.html*>, accessed at 10.11.2009.

[19] **Url-2** <*http://spinroot.com/spin/Man/GettingStarted.html*>, accessed at 9.11.09.

[20] **Holzmann J. G.,** 2003: The Primer and Reference Manual. Addison Wesley, September 04.

[21] **Ben-Ari M.,** 2008: Principles of the SPIN Model Checker.Springer.

[22] **Ruys C. T.,** 2002: SPIN Beginner's Tutorial. SPIN 2002 Workshop, University of Twente, April.

APPENDICES

APPENDIX A.1 : PROMELA Model of the RTI
APPENDIX A.2 : HLA Methods Implemented in the PROMELA Model of the RTI

APPENDIX A.1

```
mtype = {
CreateFederationExecution,DestroyFederationExecution,JoinFederationE
xecution,ResignFederationExecution,RequestFederationSave,InitiateFed
erateSave, FederateSaveBegun, FederateSaveComplete,FederationSaved,
RequestFederationRestore,ConfirmFederationRestorationRequest,Federat
ionRestoreBegun,InitiateFederateRestore,FederateRestoreComplete,Fede
rationRestored,PublishObjectClassAttributes,
SubscribeObjectClassAttributes,PublishInteractionClass,UnpublishInte
ractionClass,SubscribeInteractionClass,
RegisterObjectInstance,DiscoverObjectInstance,UpdateAttributeValues,
ReflectAttributeValues, SendInteraction, ReceiveInteraction,
UnconditionalAttributeOwnershipDivestiture,CreateRegion,CommitRegion
Modifications,
SubscribeInteractionClassWithRegions,SendInteractionWithRegions,Dele
teObjectInstance,
RequestAttributeValueUpdate,ProvideAttributeValueUpdate,RemoveObject
Instance};

mtype = { NOTCREATED,CREATED };


typedef RegionInfo
{
  mtype mRegion;
  mtype mDimensions[DIMENSIONNUM];
  int mDimensionCounter;
  int mFedId; /*id of federate (creates the region)*/
}

typedef AttributeInfo
{
  int mDesignator;
  mtype mDimensionDesignator[DIMENSIONNUM];
  int mDimensionCounter;
}

typedef ObjectClassInfo
{
  mtype mObjectClass;
  AttributeInfo mAttribute[OBJECTATTRIBUTENUM];
  int mAttributeSize;
  mtype mObjectInstance;

};
typedef InteractionClassInfo
{
  mtype mClass;
  int mAttribute[OBJECTATTRIBUTENUM];
  int mAttributeSize;
  mtype mDimensions[REGIONNUM];
  int mDimensionCounter;
};
```

```
typedef SubscribedInteractionInfo
{
  mtype mClass;
  bool mHasRegion;
  mtype mRegionArray[REGIONNUM];
  int mRegionCounter;
};

typedef SaveInfo
{
   int mSaveLabel;
   /*Federates when the save was hold.*/
   byte mSaveFedereArray[FEDERENUM];
   bool mSaveAccomplished;
};

typedef FederateInfo
{

   bool mSaveScheduled;/*precondition of Initiate Federate Save*/
   bool mInitFedSaveInvoked;
   bool mFederateSaveBegunInvoked;
   bool mSaveCompleted;

   bool mHasFedResRequest;
   bool mInitFedRestoreInvoked;
   bool mFederateRestoreCompleted;

   /*Published object class attributes*/
   ObjectClassInfo  mPublishedObjects[OBJECTCLASSNUM];
   int mPublishedObjCounter;

   /*Subscribed object class attributes*/
   ObjectClassInfo mSubscribedObjects[OBJECTCLASSNUM];
   int mSubscribedObjCounter;

   /*Registered object instances*/
   ObjectClassInfo registeredObjectInstances[REGISTEREDOBJINS];
   int mRegisteredObjectInstancesCounter;

   /*Discovered object instances*/
   ObjectClassInfo  discoveredObjectInstances[DISCOVEREDOBJINS];
   int    mDiscoveredObjectInstancesCounter;

   /*Published interactions*/
   InteractionClassInfo mPublishedInteractions[OBJECTCLASSNUM];
   int mPublishedInteractionCounter;

   /*Subscribed interactions*/
   SubscribedInteractionInfo
   mSubscribedInteractions[OBJECTCLASSNUM];
   int mSubscribedInteractionCounter;

   /*Save flags*/
   int mSaveLabelArray[SAVENUM];
   int mSaveLabelCounter;

   /*Deleted object instances*/
   ObjectClassInfo mDeletedObjectInstances;
   int mDeletedObjInstanceCounter;
```

```
};

typedef Message
{
      int msgType;
    /*interaction class or object class*/
      mtype mClass;
    /*Attribute Array for Objects or Parameters for Interactions*/
      int mAttribute[OBJECTATTRIBUTENUM];
      /*Object attribute number used bu federate  */
      int mAttributeSize;
    /*object instance = object instance handle*/
      mtype mObjectInstance;

    /*Save Label */
      int mSaveLabel;
      byte mSaveFedereArray[FEDERENUM];

      mtype mDimensionDesignator[DIMENSIONNUM];
      int mDimensionCounter;

      mtype mRegionDesignator;
      mtype mRegionArray[REGIONNUM];
      int mRegionCounter;

}


/*message , process id - channel for federates to RTI
communication*/
chan entry = [0] of { Message, int  };
/*Channels for RTI to federates communication*/
chan callbackChanArray[FEDERENUM] = [0] of {  Message };


/*9.3.3 c) The regions exist. */
/* 9.3.3 d) The regions were created by the invoking joined federate
using the Create Region service.*/
inline regionsExistAndCreatedPreCondition()
{

d_step
{
   result = false;
   i = 0;
   j = 0;
   attributeCounter = 0;
   do
     ::if
     ::(i == msg.mRegionCounter  )-> break;
     ::else ->
      do
      ::if
      ::( j == mRegionCounter) -> break;
      :: else ->
         if
         ::( msg.mRegionArray[i]  == mRegionArray[j].mRegion  &&
mRegionArray[j].mFedId == fedpid) -> attributeCounter++;
        break;
        ::else ->
        fi;
```

```
        fi;
        j++;
        od;
       fi;

   i++;
   j = 0;
od;

if
::( attributeCounter == msg.mRegionCounter && attributeCounter > 0 )
->
printf("The regions exist and created by the joined federate.\n");
::else ->
printf("The regions do not exist and are not created.\n");
mPreConditionResult = mPreConditionResult && false;

fi;

}
}
/*4.4.3 The federate is not joined to that federation execution*/
inline joinPreCondition( )
{

d_step
{
  printf("Join Precondition Control:\n ");
  result = false;
  k = 0;
  do
  ::if
  ::( k == FEDERENUM ) -> break
  ::else  ->
    if
    ::(federeArray[k] == fedpid ) ->
      result = false;
      printf("Federe with pid = %d has already joined to the
federation\n",fedpid );
      mPreConditionResult = mPreConditionResult && false;
      break;
    ::else ->  result = true;
    fi;
 fi;
 k++
 od; skip;

 if
 ::(result == true) ->
 printf("Federe with pid = %d is not joined to the
federation\n",fedpid );
 ::else
 fi;
}

}
.....................

inline sendInteractionCallback()
{
```

```
 printf("\n Send Interaction Callback : Send interaction calls the
receive interaction of other federates = %d\n",fedpid);
 result = false;
 i = 0;
 Message msg1;
 do
  ::if
   :: (i == FEDERENUM ) -> break
   ::else  ->
     if
     ::(i != (fedpid-1) ) ->
      printf("Federate with pid = %d\n",i+1);
      mPreConditionResult = true;
      interClassIsSubscribedAtJoinedFederatePreCondition(i);
      if
      ::(mPreConditionResult == true ) ->
      printf("Receive Interaction message is sent to federate with
      pid = %d\n",i+1);
      federationExecExistPreCondition();
      isJoinedPreCondition(i+1);
      /*6.9.3 e left*/
       if
       ::(mPreConditionResult == true ) ->
       msg1.msgType = ReceiveInteraction;
       callbackChanArray[i] ! msg1;
       ::else
       fi;
      ::else
      fi;
      ::else
      fi;
    fi;
    i++;
  od;skip
}


.....................

active proctype FederationEx()
{
      printf("RTI Process with pid = %d\n", _pid);

      mtype federationState = NOTCREATED;
      bool result = false;
      int fedpid =0;
      int i=0;
      int j = 0;
      int m = 0;
      int attributeCounter = 0;
      int counter = 0;
      int n = 0;
      int k = 0;
      Message msg;

      FederateInfo federateInfoArray[FEDERENUM];
      SaveInfo mSaveArray[SAVENUM];
      int mSaveCounter;
      bool mSaveInProgress = false;
      bool mRestoreInProgress = false;
      int mLastRequestedRestoreSaveLabel = -1;
      ObjectClassInfo objectClassFDDArray[OBJECTCLASSNUM];
```

```
InteractionClassInfo interactionFDDArray[INTERACTIONNUM];
byte federeArray[FEDERENUM];
mtype mDimensions[DIMENSIONNUM];
RegionInfo mRegionArray[REGIONNUM];
int mRegionCounter;
bool mPreConditionResult = true;


fillObjectInteractionInfo();


end:
do::
entry?msg,fedpid;
printf("\n\n Message = "); printm(msg.msgType);printf(" from
federate with  pid = %d\n",fedpid);

if
::( msg.msgType == CreateFederationExecution ) ->
d_step
{
 printf("RTI Service: Create Federation Execution\n");
 createFederationPreCondition();
 createFederationPostCondition();
}
::( msg.msgType == JoinFederationExecution ) ->
d_step
{
 printf("RTI Service: Join Federation Execution\n");
 mPreConditionResult = true;
 federationExecExistPreCondition();
 saveNotProgressPreCondition();
 restoreNotProgressPreCondition();
 joinPreCondition();
 if
 ::(mPreConditionResult == true ) ->
 joinFederationPostCondition();
 ::else -> assert(false);
 fi;
}
::( msg.msgType == PublishInteractionClass ) ->
d_step
{
 printf("RTI Service: Publish Interaction Class\n");
 mPreConditionResult = true;
 federationExecExistPreCondition();
 isJoinedPreCondition(fedpid);
 interactionClassInFDDPreCondition();
 saveNotProgressPreCondition();
 restoreNotProgressPreCondition();
 if
 ::(mPreConditionResult == true ) ->
 publishInteractionPostCondition();
 ::else -> assert(false);
 fi;
 }
 ::( msg.msgType == SubscribeInteractionClass ) ->
 d_step
 {
  printf("RTI Service: Subscribe Interaction Class\n");
  mPreConditionResult = true;
```

```
federationExecExistPreCondition();
isJoinedPreCondition(fedpid);
interactionClassInFDDPreCondition();
/*5.8.3d is left*/
saveNotProgressPreCondition();
restoreNotProgressPreCondition();

if
::(mPreConditionResult == true ) ->
subscribeInterClassPostCondition();
::else -> assert(false);
fi;
}
::( msg.msgType == UnpublishInteractionClass ) ->
d_step
{
 printf("RTI Service: Unpublish Interaction Class\n");
 mPreConditionResult = true;
 federationExecExistPreCondition();
 isJoinedPreCondition(fedpid);
 interactionClassInFDDPreCondition();
 saveNotProgressPreCondition();
 restoreNotProgressPreCondition();

 if
 ::(mPreConditionResult == true ) ->
 unpublishInterationPostCondition();
 ::else ->assert(false);
 fi;
 }
 ::(msg.msgType == PublishObjectClassAttributes) ->
 d_step
 {
  printf("RTI Service: Publish Object Class Attributes\n");
  mPreConditionResult = true;
  federationExecExistPreCondition();
  isJoinedPreCondition(fedpid);
 objectClassInFDDPreCondition();
 attributesAreAvailable();
 saveNotProgressPreCondition();
 restoreNotProgressPreCondition();
 if
 ::(mPreConditionResult == true ) ->
 publishObjectClassAttPostCondition();
 ::else -> assert(false);
 fi;
}
::( msg.msgType == PublishObjectClassAttributes) ->
d_step
{
 printf("RTI Service: Publish Object Class Attributes\n");
 mPreConditionResult = true;
 federationExecExistPreCondition();
 isJoinedPreCondition(fedpid);
 objectClassInFDDPreCondition();
 attributesAreAvailable();
 saveNotProgressPreCondition();
 restoreNotProgressPreCondition();
 if
 ::(mPreConditionResult == true ) ->
 publishObjectClassAttPostCondition();
```

```
          ::else -> assert(false);
          fi;
         }

        ..................

         ::(msg.msgType == ResignFederationExecution )->
         d_step
         {
          printf("RTI Service: Resign Federation Execution\n");
          mPreConditionResult = true;
          federationExecExistPreCondition();
          isJoinedPreCondition(fedpid);
          if
          ::(mPreConditionResult == true)->
          resignFederationExecutionPostCondition();
          ::else -> assert(false);
          fi;
         }
          ::else

     fi;
     od;

}/*FederationEx process*/


.................
```

APPENDIX A.2

Table A.1 : HLA methods implemented in the PROMELA model of the RTI

Federation Management	Create Federation Execution
	Join Federation Execution
	Resign Fderation Execution
	Request Federation Save
	Initiate Federate Save
	Federate Save Begun
	Federate Save Complete
	Federation Saved
	Request Federation Restore
	Federation Restore Begun
	Initiate Federate Restore
	Federate Restore Complete
	Federation Restored
Declaration Management	Publish Object Class Attributes
	PublishInteractionClass
	UnpublishInteractionClass
	Subscribe Interaction Class
Object Management	RegisterObjectInstance
	DiscoverObjectInstance
	UpdateAttributeValues
	ReflectAttributeValues
	SendInteraction
	ReceiveInteraction
	RequestAttributeValueUpdate
	ProvideAttributeValueUpdate
	DeleteObjectInstance
Data Distribution Management	CreateRegion
	CommitRegionModifications
	Subscribe Interaction Class With Regions
	Send Interaction With Regions

CURRICULUM VITA

Candidate's full name: Vijdan KIZILAY

Place and date of birth: Kırcali, 23 April 1984

Permanent Address: Güzeller mh. Şehit Mevlüt Duru cd. No = 7 Gebze/Kocaeli

Email: vkizilay@gmail.com

Universities and Colleges attended: Hacettepe University Department of Computer Engineering, 2006, BS, Bursa Anatolian High School, 2002

Work: TUBITAK MAM Research Center, Information Technologies Institute, Software Engineer, 2006 - Present

Publications:

- **Kızılay V.,** Topçu O., Oğuztüzün H., and Buzluca F., 2009: Verifying the Interface Compliance of Federates Using Pre- and Postconditions of RTI Services. 2009 Fall Simulation Interoperability Workshop (SIW), September 21-25, 2009 Orlando, Florida.

- **Kızılay V.,** Topçu O., Oğuztüzün H., 2009: HLA Federe Arayüz Servislerinin Ön ve Son Koşullarının Federasyon Mimari Metamodeline Eklenmesi. 3. Ulusal Savunma Uygulamaları Modelleme ve Simülasyon Konferansı (USMOS), June 17-18, 2009 Ankara, Turkey